Evaluación temporal del entrenamiento e inferencia de redes neuronales sobre plataformas hardware heterogéneas
Time evaluation of neural networks training and inference on heterogeneous hardware platforms

Trabajo de Fin de Máster
Curso 2018–2019

Autor
Lorenzo Martín López

Director
José Ignacio Gómez Pérez
Francisco Daniel Igual Peña

Evaluación temporal del entrenamiento e inferencia de redes neuronales sobre plataformas hardware heterogéneas
Time evaluation of neural networks training and inference on heterogeneous hardware platforms

Trabajo de Fin de Máster en Internet de las Cosas
Departamento de Arquitectura de Computadores y Automática

Autor
Lorenzo Martín López

Director
José Ignacio Gómez Pérez
Francisco Daniel Igual Peña

Convocatoria: *Septiembre 2019*
Calificación: *7,5*

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

27 de septiembre de 2019

A mi abuelo Valentín.

Agradecimientos

A mis tutores Nacho y Fran, por cooperar de tan buena manera y ayudarme a aprender. A Nico por soportar y resolver mis incidencias con los sistemas. A Victoria López y a Rafael Caballero por incitarme a aprender de los datos. A mi familia por alentarme en todo momento. Y a mi compañero y colega Esteban Izquierdo siempre tan honorable y dispuesto a recordarme mi valía.

Resumen

Evaluación temporal del entrenamiento e inferencia de redes neuronales sobre plataformas hardware heterogéneas

Este trabajo evalúa el rendimiento temporal de un conjunto de modelos de aprendizaje automático a partir de datos provenientes de un sistema de captación IoT compuesto por sensores de irradiación solar. Los algoritmos construidos se apoyan en redes neuronales artificiales, que serán implementadas mediante los frameworks para *deep learning* Tensorflow y Keras utiilzando el lenguaje de programación Python.

Se generará un grupo de modelos capaces de resolver un problema de regresión a partir de datos históricos de radiación solar consiguiendo predecir esta para un tiempo futuro. Para llegar a este objetivo será necesario acometer el entrenamiento e inferencia sobre las distintas redes neuronales. Estos procesos se realizarán sobre distintas arquitecturas hardware heterogéneas formadas por CPUs y GPUs capaces de acelerar los procesos computacionales.

Una vez realizada la computación necesaria, se evaluarán los tiempos requeridos para el procesado de cada uno de los modelos en las diferentes arquitecturas. Se realizará una comparativa para el proceso de entrenamiento y otra para el proceso de inferencia. De este modo se podrá vislumbrar cuándo es conveniente acelerar el cómputo mediante GPUs y cuándo es suficiente con utilizar la CPU en función de los requerimientos temporales de cada contexto.

Palabras clave

Deep learning, IoT, Tensorflow, Keras, Python, GPU, Nvidia, Neural network, regressión.

Abstract

Time evaluation of neural networks training and inference on heterogeneous hardware platforms

This work evaluates the temporal performance of a set of Machine learning models based on data from an IoT collection system located in a solar thermal power plant. The constructed algorithms are based on artificial neural networks, which will be implemented through the paradeep learningTensorflow and Kerasutiilil frameworks using the Python programming language.

A group of models capable of solving a regression problem will be generated based on historical solar radiation data, predicting this for a future time.To reach this goal it will be necessary to undertake training and inference about the different neural networks. These processes will be carried out on different heterogeneous hard-ware architectures formed by CPUs and GPUs capable of accelerating computational processes. Once the necessary computation has been carried out, the times required for the processing of each of the models in the different architectures will be evaluated.

A comparison will be made for the training process and another for the inference process. From this mode you can see when it is convenient to accelerate the computation through GPUs and when it is enough to use the CPU according to the temporal requirements of each context.

Keywords

Deep learning, IoT, Tensorflow, Keras, Python, GPU, Nvidia, Neural network, regression.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

"Ya no experimentaba cólera por las cosas ni por las personas."
— Pío Baroja. El árbol de la ciencia. 1911.

Actualmente y cada vez más, la toma de decisiones se delega a sistemas informáticos capaces de generar inteligencia y decidir por las personas. Esto es posible gracias a la existencia de dos factores fundamentales: los datos y su procesado. Debido al avance del Internet de las Cosas (IoT) la generación de datos ha sufrido una creciente evolución, dado que ahora somos capaces de monitorizar prácticamente cualquier acción por pequeña que sea. Para poder generar valor o inteligencia de estos datos, en los últimos años han surgido técnicas avanzadas de aprendizaje automático basadas en redes neuronales que son capaces de procesar inmensas cantidades de información y generar conclusiones a partir de dichos datos. Al igual que estas técnicas han avanzado considerablemente, han surgido nuevas arquitecturas hardware capaces de llevar a cabo las operaciones de cómputo de una forma mucho más eficiente que los dispositivos convencionales. Este trabajo aborda la tarea del procesado de datos a partir de mediciones tomadas por un sistema de captación IoT compuesto por sensores de irradiación solar. Para abordar la tarea se hará uso de herramientas software y hardware de última generación capaces de acometer la computación sobre este conjunto de datos. Así pues, el objetivo de este trabajo será evaluar, en términos de rendimiento temporal, cuál de las arquitecturas hardware es conveniente utilizar para la ejecución de los procesos de entrenamiento e inferencia de los algoritmos software desarrollados mediante redes neuronales.

1.1. Motivación

Debido a la creciente complejidad de los modelos de aprendizaje automático en el campo del *deep learning* es preciso conocer los recursos hardware que precisan para ejecutarse de forma eficiente. Ubicar computacionalmente los procesos de entrenamiento e inferencia de los modelos no es una cuestión baladí, ya que localizar estas tareas convenientemente permite prever las necesidades de aprovisionamiento de los sistemas de cómputo. Por tanto, en este trabajo se recurre a dos tecnologías de última generación, software y hardware, que se complementarán de la mejor forma posible para evaluar temporalmente la resolución de un problema concreto y permitir seleccionar cual es la arquitectura hardware propicia para acometer la computación.

1.2. Objetivos

1. Desarrollo software de un conjunto de modelos de aprendizaje automáticos capaces de acometer un problema de regresión y capaces de ejecutarse sobre plataformas hardware heterogéneas.
2. Creación de un marco comparativo en función de las características de los modelos.
3. Integración de herramientas de monitorización y perfilado que permitan vislumbrar las operativas de los procesos de entrenamiento e inferencia de los modelos y obtener métricas temporales.
4. Evaluación del rendimiento temporal de los procesos de entrenamiento e inferencia de los modelos en diferentes plataformas hardware.

1.3. Plan de trabajo

El plan de trabajo seguido para la consecución de los objetivos ha sido, en primer lugar, una investigación y estudio exhaustivos sobre el aprendizaje automático y el deep learning. Del mismo modo y al mismo tiempo, una familiarización con el framework Tensorflow para el lenguaje de programación Python.

Una vez avanzada la parte conceptual, se han ido implementando fragmentos de software con funcionalidades muy concretas para afianzar lo aprendido y para probar las distintas arquitecturas hardware. Acto seguido se ha ido conformando un entorno de desarrollo software capaz de abordar el entrenamiento y la inferencia en redes neuronales para posteriormente proceder a la investigación sobre herramientas capaces de obtener métricas de su ejecución y resultados útiles para la futura evaluación.

Finalmente, generado el conjunto de modelos, estudiadas las particularidades del hardware y con la destreza adquirida con las herramientas software tras un aprendizaje práctico, se ha realizado una comparativa que permite conocer los comportamientos de las distintas plataformas hardware según su naturaleza al ejecutar modelos basados en redes neuronales.

1.4. Estructura de la memoria

- Capítulo 1: informa de los objetivos del trabajó y cómo se desarollarán.
- Capítulo 2: incluye detalles sobre las tecnologías a las que recurre el proyecto y su implicación en cada tarea realizada.
- Capítulo 3: explica todo el proceso realizado para construir modelos de aprendizaje automático y la monitorización de sus procesos de entrenamiento e inferencia.
- Capítulo 4: describe el proceso comparativo realizado para conseguir una evaluación del rendimiento temporal de los procesos de entrenamiento e inferencia en las distintos plataformas hardware.
- Capítulo 5: muestra las conclusiones obtenidas y propuestas de continuación del desarrollo futuro de las mismas.
- Capítulo 6: traducción al inglés de la introducción.
- Capítulo 7: traducción al inglés de las conclusiones y los trabajos futuros.

Capítulo 2

Estado del Arte

En este capítulo se detallan las tecnologías utilizadas para el desarrollo de este trabajo así como los motivos por los cuales han sido seleccionadas. En primer lugar se tratará sobre el estado del arte del campo del aprendizaje automático y su variante basada en redes neuronales, el aprendizaje profundo o *deep learning*. Acto seguido se describirán las herramientas de software utilizadas para llevar a cabo estas técnicas y finalmente las ventajas que puede aportar la aceleración de las mismas mediante GPUs.

2.1. Introducción a la inteligencia artificial y al aprendizaje profundo

La inteligencia artificial se puede definir como la subdisciplina del campo de la Informática que busca la creación de máquinas que permitan imitar comportamientos inteligentes. Dentro de este gran abanico de posibilidades caben tareas como conducir un coche, predecir un valor numérico a largo plazo, clasificar objetos o generar contenido multimedia, entre otros. Las máquinas se programan para realizar estas acciones, llegando incluso a superar al ser humano en multitud de ellas, lo cual las convierte en herramientas totalmente funcionales y eficaces. Sin embargo, actualmente estas inteligencias artificiales no permiten consolidar una inteligencia holística como la que puede desarrollar la mente humana que sí puede realizar numerosas tareas de forma inteligente. Es decir, cada una de estas máquinas inteligentes están diseñadas para la realización de un fin concreto exclusivamente imitando comportamientos inteligentes. Por este último motivo, es por el que la inteligencia artificial se ha disgregado en campos concretos como la robótica, en el que la máquina tiene la capacidad de adaptarse al entorno que la rodea; o el procesado del lenguaje natural, en el que la máquina es capaz de interpretar el comportamiento humano, entre otros.

Para acometer todos este tipo de acciones de forma inteligente por parte de las máquinas, es esencial que ésta tenga la capacidad de aprender. Para ello, nace el aprendizaje automático, que se puede definir como una rama de la inteligencia artificial que busca dotar a las máquinas de capacidad de aprendizaje. Esto se consigue mediante la generalización del conocimiento a partir de un conjunto de experiencias por las que la máquina aprende cómo realizar una acción y no solo a ejecutarla de forma automática. Dentro de este campo, existen numerosas técnicas usadas para diversos fines como la regresión lineal, la regresión logística, los árboles de decisión y sus variantes, la clusterización, etc. Para este trabajo, la técnica de aprendizaje automático que se ha empleado es la basada en redes neuronales, que permiten generar la inteligencia de forma jerarquizada identificando patrones en los

datos. Esta generación de la inteligencia se consigue de forma secuencial y abstracta a medida que los datos van atravesando las diferentes capas que componen una red neuronal. Debido a que la tendencia es que estos modelos basados en redes neuronales aumenten en complejidad añadiendo más y más capas, se genera el concepto de aprendizaje profundo o *deep learning* para aglutinar este tipo de algoritmos.

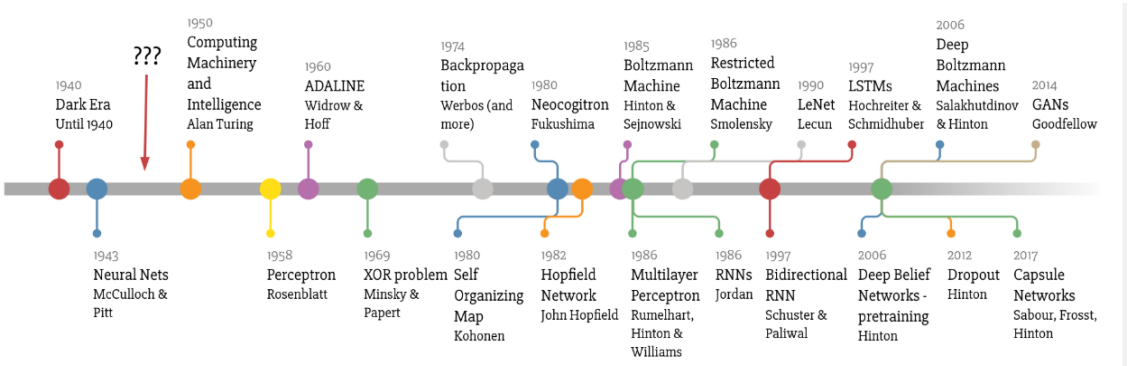


Figura 2.1: Evolución del *Deep learning*.
Fuente: Vazquez (2018)

Debido a el desarrollo de nuevas tecnologías como el IoT, capaces de generar ingentes cantidades de datos (comúnmente conocido como *Big Data*), se recurre cada vez más a algoritmos más y más complejos que son solo capaces de generarse mediante las redes neuronales y el aprendizaje profundo. En este trabajo se elaborarán distintos algoritmos basados en redes neuronales con topologías diversas para acometer un proceso de predicción de la radiación solar a partir de datos de sensores *IoT*. Para poder evaluar el diseño de estos modelos, será conveniente detallar cuales son sus elementos básicos y su forma de funcionar.

2.2. Introducción a las redes neuronales.

Para realizar esta sección se ha recurrido al capítulo 4.1 del trabajo fin de grado de Alberto Terceño Ortega de la Universidad Complutense de Madrid, Terceño Ortega (2017) adaptando la casuística a este trabajo en el que se aborda un problema de predicción de un valor real. Se empezará detallando la unidad básica de cálculo de una red neuronal, la neurona, hasta finalizar con los hiperparámetros que permiten tunear el comportamiento de una red neuronal profunda con numerosas capas ocultas y los optimizadores para el proceso de entrenamiento. Gracias al *framework* TensorFlow estas operativas se podrán codificar, pero primero es conveniente detallar cómo deben implementarse conociendo las peculiaridades de las redes neuronales.

2.2.1. La neurona

La neurona la unidad básica de procesamiento de una red neuronal que emula una función matemática. Ya fueron modeladas en el año 1958 por Frank Rosenblatt en forma del denominado perceptrón, capaz de construir cualquier proposición lógica según se puede ver en la figura 2.2.

Así pues, y basándose en esta estructura, una neurona realiza una suma ponderada de los valores numéricos de entrada en función de los pesos que tiene asignados a la neurona.

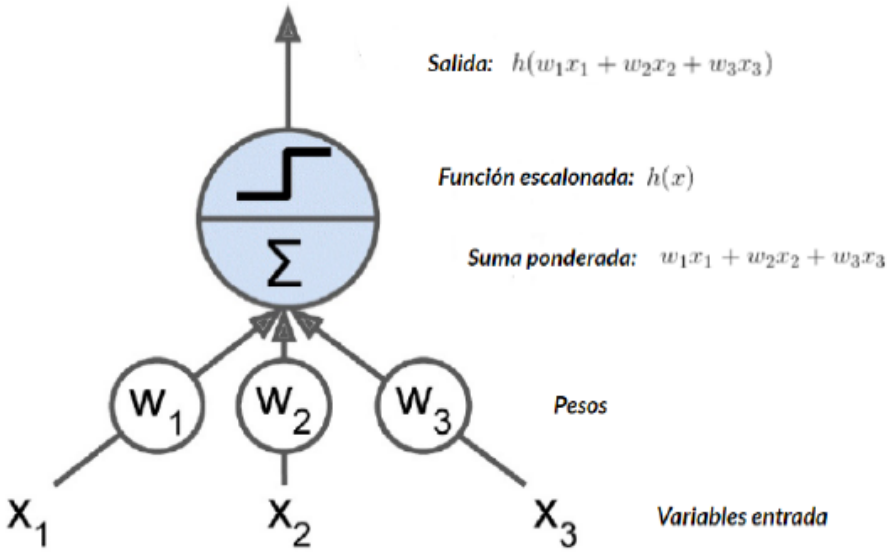


Figura 2.2: Perceptrón de Rosenblatt.
Fuente: Geron (2017)

Estos pesos, representan los parámetros del modelo matemático y generalmente se inicializan con un valor aleatorio. Además, a diferencia del perceptrón de Rosenblatt, la neurona también presenta un término independiente que se incorpora a esta suma ponderada siendo conocido como bias o sesgo. Este elemento siendo independiente de los valores de entrada, determinará también, según se puede ver en la figura 2.3, el valor de salida de la neurona. Finalmente, tras el proceso de aprendizaje, estos pesos y el sesgo se irán ajustando hasta quedar registrados diferenciando al modelo.

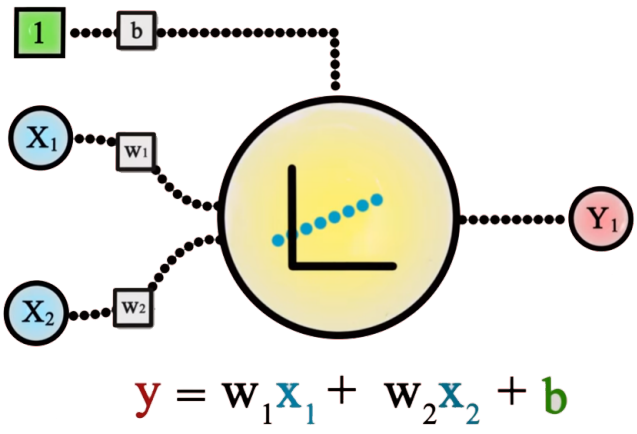


Figura 2.3: Neurona con dos variables de entrada.
Fuente: DotCSV (2018)

Gracias a una neurona, se podrá modelar una recta de forma similar a una regresión lineal y que permita diferenciar entre dos clases mediante un umbral, resultando en un modelo genérico de regresión logística binomial basada en una puerta lógica sencilla. La

figura 2.3 es el ejemplo más simple de clasificador, sin embargo y debido a su estructura, no permite la generación de umbrales que puedan emular puertas lógicas diferentes a las puertas AND, OR, NAND y NOR, como las XAND o XOR que requieren de dos rectas para establecer el umbral de decisión como se puede ver en la figura 2.4. Es por esto, por lo que a medida que se necesite establecer fronteras de decisión más complejas para cumplir con los objetivos deseados, se recurre a la adición de más neuronas para generar modelos más precisos.

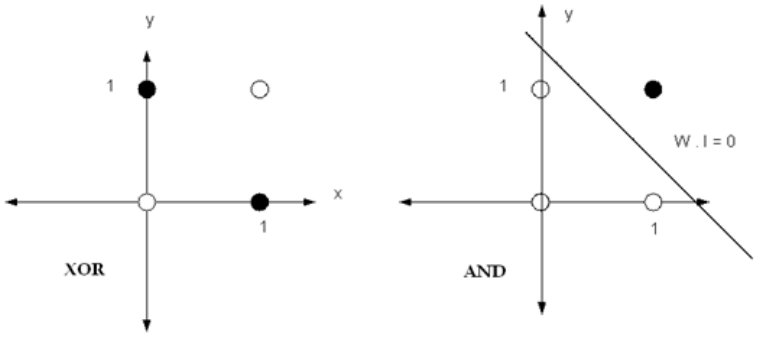


Figura 2.4: Puerta lógica XOR sin umbral definido y puerta lógica AND con umbral definido. Fuente Rossel (2018)

Sin embargo, el proceso de añadir neuronas a la estructura no es trivial, por tanto se debe hacer de manera estructurada comprendiendo las interrelaciones que se van a generar dentro de la red de la red neuronal. Además, y partiendo de la estructura del perceptrón de Rosenblatt, se podrán incorporar hiperparámetros similares a la función escalonada que permitan ganar en precisión como las funciones de activación, como actualmente estos son una característica que se aplica a capas de neuronas, se detallará primero como se construye una red neuronal.

2.2.2. La red neuronal y sus capas de neuronas

Como se ha mencionando anteriormente, para que un sistema aprenda a realizar una tarea mediante una red neuronal, debe adquirir el conocimiento de forma jerarquizada. Por tanto, la forma en la que se conecten las neuronas entre sí debe compartir este comportamiento. Existen numerosas formas de conectar las neuronas dentro de la red, sin embargo, en lo que a localización espacial se refiere, las neuronas se organizan en capas de neuronas situadas en paralelo. De forma secuencial, las neuronas de la capa se conectan a las neuronas de la capa siguiente permitiendo establecer una arquitectura de transmisión de la información de forma jerarquizada.

A la primera capa, se la denomina capa de entrada y presenta las dimensiones del conjunto de datos que se inyectará a la red neuronal. A partir de aquí, se pueden conectar capas de neuronas denominadas capas ocultas hasta que se alcanza la última capa, la denominada capa de salida. Las dimensiones de la capa de salida están determinadas por el resultado esperado, es decir, si se pretende generar una imagen, las dimensiones de la última capa deberán tener las dimensiones de la imagen deseada. Para el caso de este trabajo, en el que se pretende realizar la predicción de un valor numérico real, la capa de salida presentará una única neurona, generando valores entre 0 y 1 que se podrán interpretar como valores reales.

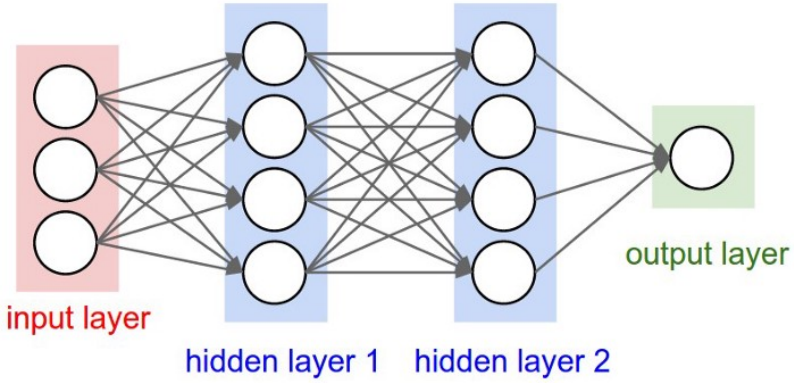


Figura 2.5: Red neuronal con dos capas ocultas.
Fuente Dabbura (2018)

La red neuronal con todas sus neuronas y capas, resultarán en un sistema final con una entrada y una salida, la cual es necesaria para poder generar una inteligencia funcional. Sin embargo, entre las capas del sistema sí que se pueden realizar conexiones cíclicas con un número determinado de iteraciones. Para la generación de los modelos de este trabajo no se recurrirá a redes con comportamientos cíclicos, si no que se utilizarán únicamente las redes *feed forward*, que no los presentan. Por otro lado, las capas más simples son aquellas en las que todas sus neuronas se conectan a cada una de las neuronas de la capa siguiente, son denominadas *fully-connected* y se usarán en este trabajo de forma reiterada. También se utilizarán capas de neuronas de tipo convolucional, más propicias para operaciones sobre imágenes. En el capítulo 3 se verá con mayor detalle.

2.2.3. Las funciones de activación y los hiperparámetros de las redes neuronales

Como se ha explicado anteriormente, la salida de una neurona se puede representar como una suma ponderada de sus entradas. La concatenación de numerosas capas de neuronas que generan salidas con comportamiento lineal, puede comportarse como una única regresión lineal, es decir, lleva al sistema de una red neuronal a colapsar en una única neurona. Por tanto, y para que este suceso no ocurra, se somete a la salida de las neuronas a una variación de comportamiento no lineal; de este modo, la salida del conjunto final no puede representarse como un sistema lineal. Estas deformaciones de tipo no lineal se conocen como funciones de activación y afectan directamente las salidas de cada una de las neuronas según se muestra en la figura 2.6.

En el Perceptrón generado por Rosenblatt la función de activación que se aplica sobre la salida es la función Heaviside o función signo, en la que para una salida de valor menor a un umbral, el valor se convierte en 0, y para un valor superior al umbral, la salida se convierte en 1 según lo siguiente:

$$Heaviside(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \qquad sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

Sin embargo, estas funciones, al no ser derivables, presentan grandes complicaciones para ser computadas por el algoritmo de descenso de gradiente, imprescindible para conse-

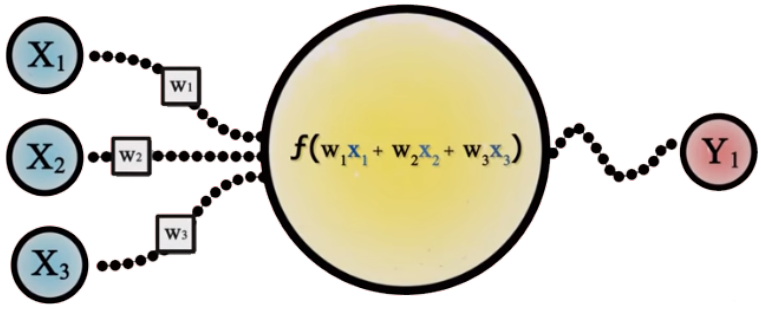


Figura 2.6: Activación aplicada a la salida de la neurona
Fuente DotCSV (2018).

guir el aprendizaje automático de la red neuronal como se vera en el apartado siguiente. Es por esto por lo que las funciones de activación que se aplican a las neuronas presentan la cualidad de ser derivables, algunas de ellas se muestran en la figura 2.7. Para este trabajo las función de activación que se empleará será la ReLU.

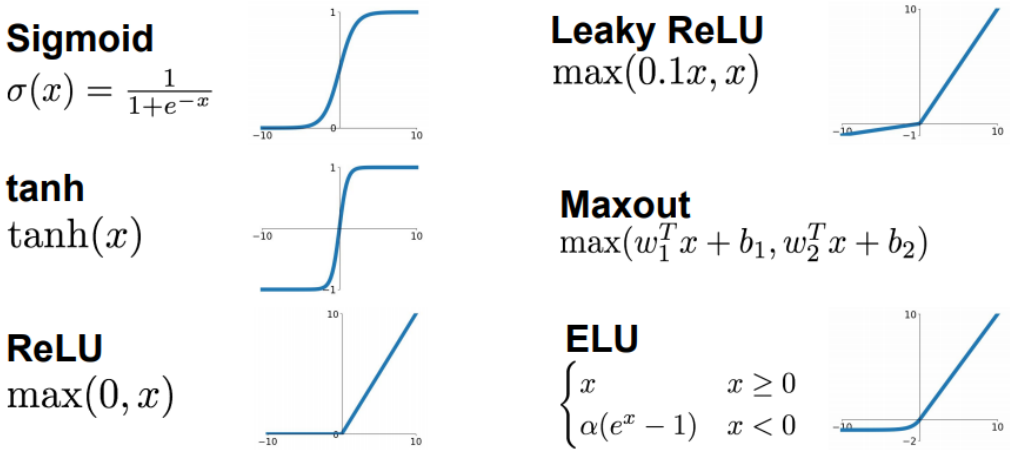


Figura 2.7: Funciones de activación.
Fuente: CS231 (2017)

Gracias a estas deformaciones y a la concatenación de neuronas colocadas en capas se consigue poder generar una gran diversidad de umbrales, o fronteras de decisión, mediante estos algoritmos que ya no presentan un comportamiento lineal. De este modo sí se podrán abordar problemas de naturaleza difusa en los que el número de variables sea considerable y la necesidad sea de regresión, clasificación, *clusterización*, etc. Además de las funciones de activación, las redes neuronales presentan multitud de parámetros ajustables denominados hiperparámetros para ser distinguidos de los parámetros o pesos de la red neuronal; cada uno con una funcionalidad concreta. Algunos de ellos son:

- La tasa de aprendizaje (*learning rate*): con valores entre 0 y 1, determina con qué frecuencia se actualizan los pesos de las neuronas en el proceso de entrenamiento, siendo 1 la mayor velocidad de actualización y 0 el no aprendizaje. Este parámetro puede ajustar de forma dinámica según se va realizando el proceso de entrenamiento y generalmente se establece que su comportamiento sea decreciente, así la red irá

aprendiendo rápidamente al principio y seguirá aprendiendo a partir de esta experiencia más lentamente en pasos sucesivos. Esto permite que eventos extraños que pudieran aparecer en las fases finales del aprendizaje resulten igual de válidos que todos aquellos anteriores que sí seguían un comportamiento similar.

- Número de neuronas por capa oculta: perteneciente al conjunto de números naturales positivos hasta infinito, determina la anchura de la red y la topología de la red. Permitirá crear multitud de estructuras de redes y será un valor que se estudiará en este trabajo para determinar su influencia en los tiempos de entrenamiento e inferencia.

- El número de capas ocultas: perteneciente al conjunto de números naturales positivos hasta infinito, será un valor que se estudiará en este trabajo para determinar su influencia en los tiempos de entrenamiento e inferencia.

- El número de iteraciones sobre el conjunto de datos para entrenamiento o *epochs*: perteneciente al conjunto de números naturales positivos hasta infinito, establece las veces que el conjunto de datos se introduce en la red neuronal. Este parámetro influye de forma directamente proporcional en el tiempo de entrenamiento de una red neuronal, por tanto, tras demostrar esto bajo diferentes condiciones, se establecerá a 1 para poder dilucidar cómo otros parámetros influyen sobre un único entrenamiento de los datos.

- El tamaño del lote o *batch size*: con un valor máximo determinado por el ancho de banda del dispositivo sobre el que se procesa la red neuronal, es el número de muestras del conjunto de datos de entrenamiento o inferencia que se introducen en la red a la vez. Este parámetro define el aprendizaje por lotes, ya que tras la computación de cada uno de estos es cuando se produce el reajuste de los pesos de las neuronas de la red. En este trabajo se probará con diferentes tamaños para determinar su infuencia en los tiempos de entrenamiento e inferencia.

- El tipo de optimizador, es el algoritmo que determina cómo aprende la red en el proceso de entrenamiento. Existen multitud de optimizadores pero el utilizado para este trabajo será el ADAM debido a su versatilidad. Puede ampliarse la información consultando en Kingma y Ba (2017).

Con estos atributos se podrá configurar cómo es la red neuronal y cómo debe aprender, gracias al *framework* TensorFlow y al lenguaje Python se podrán implementar modelos de diversa naturaleza y procesarlos para dotarlos de inteligencia. Este punto se tratará en la sección 2.3 detallando las sinergias que existen entre esta librería software y las redes neuronales. En la figura 2.8 se muestra una estructura equiparable a una neurona con una función de activación ReLU generada con Tensorflow. Sin embargo hasta ahora no se ha mencionado nada de qué es lo que determina que se elija un valor u otro para los pesos de las neuronas, es decir, que la red se ajuste y se adquiera la inteligencia. Esto es lo que se define como proceso de entrenamiento.

2.2.4. El proceso de entrenamiento

El proceso de entrenamiento de las redes neuronales es el paso clave, una vez definida la red, que permitirá a esta ajustar sus pesos y aprender a partir de los datos que procesa generando una interpretación. Pero antes de comenzar a detallar el proceso de entrenamiento de las redes neuronales es importante diferenciar los tipos que existen, ya que según el

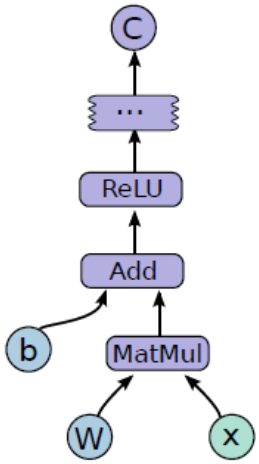


Figura 2.8: Red neuronal generada mediante Tensorflow.
Fuente Abadi et al. (2016)

aprendizaje que se quiera realizar a partir de los datos, se utilizarán unos algoritmos u otros. En el aprendizaje supervisado, la red neuronal aprende basándose en una referencia a la salida de la red, que le permite comparar esta con las etiquetas. Estas etiquetas son los resultados esperados y permiten indicar a la red si está aprendiendo correctamente o no. Este es el tipo de aprendizaje que se ha llevado a cabo con los algoritmos diseñados en este trabajo ya que se conocían los resultados que debía devolver el modelo para cada muestra de entrada. En contrapartida existe el aprendizaje no supervisado, destinado generalmente a problemas de clusterización y para deducciones que un ser humano no puede interpretar a partir de un conjunto de datos pero una red neuronal sí. Por último encontramos también el aprendizaje por refuerzo, en el que mediante una función de recompensa, se va premiando al modelo cuando consigue un objetivo determinado, este tipo de aprendizaje se ha utilizado para generar inteligencias que han llegado a batir a seres humanos en diversos juegos como el ajedrez o el go y hasta en juegos cooperativos, véase Shooter (2019)

En este apartado se tratará sobre el aprendizaje supervisado. Para entrenar este tipo de algoritmos ajustando los pesos de las neuronas se recurre al algoritmo del descenso de gradiente, que es capaz de determinar el error que ha cometido una neurona respecto a la salida deseada en cualquier punto del modelo y minimizar ese error mediante la función de coste o *loss function*. Esto se consigue calculando las derivadas parciales de la función de la neurona e identificando así la dirección en la cual se propaga el error, de este modo, la propia neurona sabrá que tendrá que ajustar sus pesos en el sentido contrario a este gradiente.

Sin embargo, este algoritmo no es suficiente para una concatenación de funciones como son las neuronas de una red neuronal en las que muchas de ellas vienen condicionadas por numerosas conexiones previas. En este caso, se recurre también al algoritmo de *backpropagation*, que facilita al algoritmo de descenso de gradiente la propagación hacia atrás del error, capa a capa, una vez se haya realizado un paso de entrenamiento y la comparación final con la salida deseada. Ver figura 2.9.

Realizando una retropropagación del error desde la capa de salida hasta las capas de entrada se permite generar una cadena de responsabilidades de modo que se puede evaluar la importancia que tiene cada neurona en la decisión final y si este este error proviene de

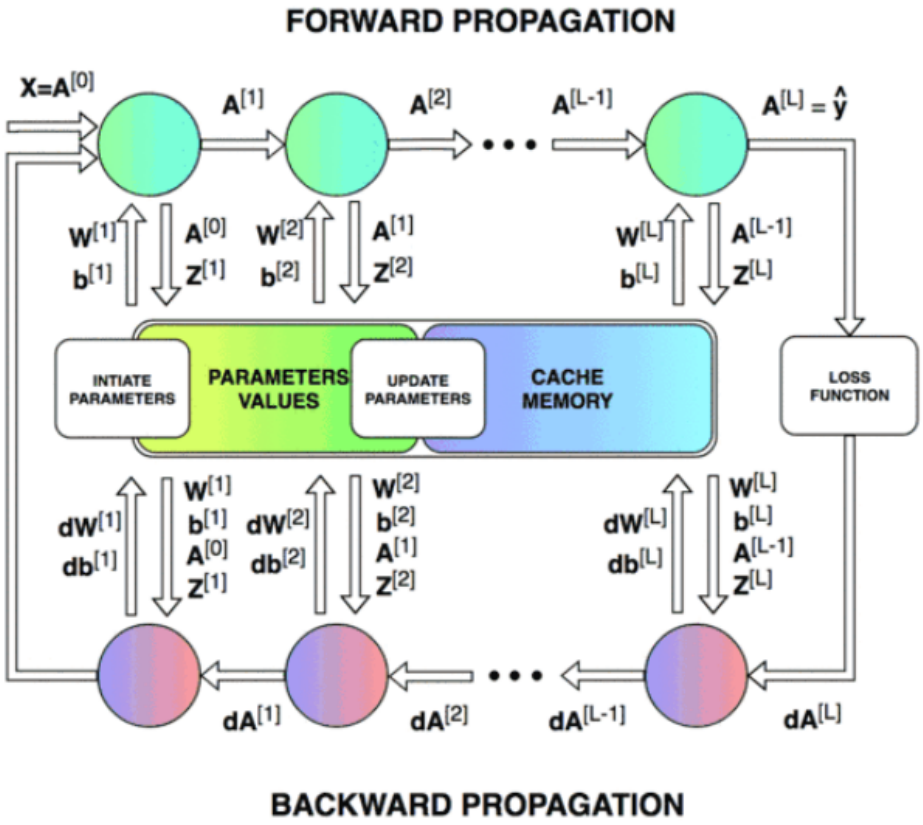


Figura 2.9: Ajuste de los parámetros de la red (W, b, A, Z)
Fuente: Moawad (2018)

las capas previas o no. Haciendo referencia a la estructura de una neurona y a su salida determinada por la suma ponderada de sus entradas, si se conoce cual de las entradas es la que más repercute en el error se deberá ajustar el valor del peso ligado a esa entrada para que tenga menor repercusión en la salida.

Como se ha mencionado anteriormente, el aprendizaje se realiza mediante lotes de datos o *batches* que se propagan por la red hasta resultar en una salida para que esta sea comparada con unas etiquetas, propagar el error cometido hacia detrás y así permitir el ajuste de los pesos de las neuronas. Todos estos procesos implican un alto coste computacional, sin embargo a día de hoy es la metodología más eficiente. En el capítulo siguiente se verá cómo repercuten estos procesos en el tiempo total requerido por el entrenamiento en función del número de operaciones que se requiera realizar. Por ejemplo, en la figura 2.10 se pueden apreciar el flujo operaciones en los cores de una CPU al realizar un proceso de entrenamiento. En la sección 3.6.3 se mostrarán con mayor detalle incluyendo información más detallada de los tiempos requeridos por cada una de ellas.

2.2.5. El proceso de inferencia.

El proceso de inferencia consiste en generar una conclusión o resultado recurriendo a un modelo ya entrenado al que se le aplican unos datos. Como se mencionó al principio del capítulo, los modelos no suelen generar inteligencias generalistas, por tanto, es conveniente

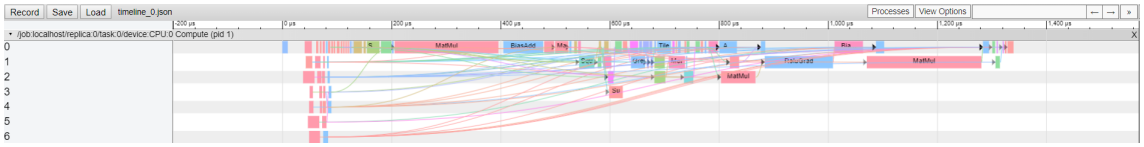


Figura 2.10: Trazas de entrenamiento generadas por Tensorflow

.

aplicar datos de naturaleza similar a los que se han utilizado para entrenar la red si se quiere obtener un resultado razonable. El proceso de inferencia es el que finalmente determina la precisión de un modelo ya que es el que se aplica en entornos de producción.

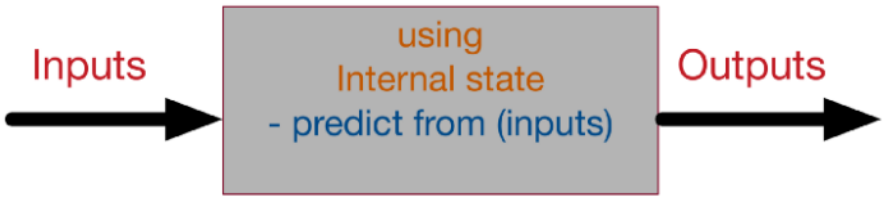


Figura 2.11: Modelo pre-entrenado como sistema simple y cerrado
Fuente: Moawad (2018)

Este proceso es mucho más rápido que el proceso de entrenamiento ya que aquí no se realiza el reajuste de los pesos, sin embargo, es conveniente analizar cuanto más puede tardar la inferencia en un dispositivo hardware u otro para determinar cual es el más conveniente para obtener el resultado en escenarios donde el tiempo sea un factor crítico. En este trabajo también se compararán los tiempos de inferencia en plataformas con aceleración por GPUs y sin ellas.

2.3. Lenguajes de programación

En este trabajo se ha usado el lenguaje de programación Python y sus librerías para los procesos de preprocesado de datos y para la generación, entrenamiento, inferencia y perfilado de las diferentes redes neuronales. Para el proceso de análisis de métricas se ha usado el lenguaje de programación R. A continuación se indican los porqués de cada uno de ellos.

Python: lenguaje de programación de propósito general interpretado en gran auge en la actualidad debido su fácil legibilidad, el renacimiento del aprendizaje automático y su multitudinaria comunidad. Sus características principales aplicadas a este trabajo son:

- Es software libre.
- Permite programación funcional.
- Posee estructuras de datos que facilitan el tratamiento de datos estructurados y no estructurados.
- Permite ser ejecutado en plataformas hardware heterogéneas.

- Cuenta con numerosas librerías para aprendizaje automático. Debido a su reiterada utilización durante el trabajo, cabe destacar en mayor medida las librerías utilizadas en este lenguaje para ganar en funcionalidad:
  - Pandas (Python Data Analysis Library): librería de código abierto para el tratamiento de datos estructurados, desde su importación hasta su generación. Permite trabajar cómodamente con datos en forma de tabla gracias a su estructura principal, el *dataframe*. Particularmente se ha utilizado para traer los datos al entorno de trabajo, hacer indexación y selección de los mismos para un preprocesado previo a las fases de inyección en los procesos de entrenamiento e inferencia en las redes neuronales. También se ha utilizado para generar las tablas que contienen las métricas de los perfilados.
  - Numpy (Numerical Python): librería de código abierto para el tratamiento de datos numéricos. Su estructura principal de datos es el *array* multidimensional. Permite ganar en agilidad al trabajar con datos numéricos y ganar en eficiencia computacional gracias a las operaciones vectorizadas, además, los *array*s son plenamente compatibles con TensorFlow para poder inyectarlos a las redes neuronales. Se ha utilizado para generar las diferentes topologías de redes neuronales y también para transformar y dimensionar los datos en tablas a formato *array* y así poder introducirlos a las redes neuronales.
  - *Scikit-learn*: librería de Python para aprendizaje automático. Cuenta con diversas funciones para generar diversidad de modelos y para obtener métricas de forma sencilla de cara a su evaluación. Los modelos iniciales de los que parte este trabajo se encontraban escritos mediante esta librería. Se ha utilizado para obtener algunas métricas de forma sencilla.

R: lenguaje de programación para análisis estadístico y visualización de datos. Al igual que Python es un lenguaje interpretado y de software libre. Cuenta con R-studio, un IDE que permite el ágil tratamiento de los datos en el entorno de trabajo, el manejo de librerías, una correcta escritura y la gestión de proyectos. Las librerías utilizadas para el análisis han sido:

- readr y data.table: para la importación de datos.
- tidyr: para la correcta ordenación de los datos.
- stringr: manipulación de datos tipo string.
- dplyr: transformación de datos.
- ggplot2: generación de gráficas.

A continuación se describirán más detalladamente las librerías utilizadas en Python para la generación de redes neuronales, su entrenamiento, inferencia y perfilado: TensorFlow y Keras.

2.4. TensorFlow

TensorFlow es una librería de software libre liberada por el equipo Google Brain, de la empresa multinacional Google, en el año 2015 bajo licencia Apache 2.0. Su última versión estable, a fecha de emisión de este trabajo, es la 1.14 con fecha 19 de junio de 2019; también

se ha lanzado la versión beta 2.0.0-beta1. Está escrita en los lenguajes C++, Python y CUDA y es multiplataforma.

Se liberó como un *framework* para cálculo numérico multidimensional. Aporta herramientas para la construcción de modelos matemáticos mediante grafos direccionales, lo que implica que de este modo, cada modelo matemático se puede segmentar en grafos más pequeños. Esto ofrece serias ventajas de cara a la hora de computar las operaciones, ganando en versatilidad para calcular el modelo. Esta característica es la que la diferencia sustancialmente de la librería de cálculo numérico más utilizada en el lenguaje Python, Numpy, con la cual comparte numerosas otras como el cálculo N-dimensional, las operaciones vectorizadas y los tipos de dato. Sin embargo, y aunque partiendo de la misma base, TensorFlow también permite más formatos de datos numéricos que Numpy (F64, ...), permite crear funciones y el cálculo automático de funciones primitivas como se describe en Ramsundar (2019) que es un tutorial sobre Tensorflow. Además, TensorFlow sí puede ejecutarse en plataformas hardware heterogéneas con GPUs y TPUs.

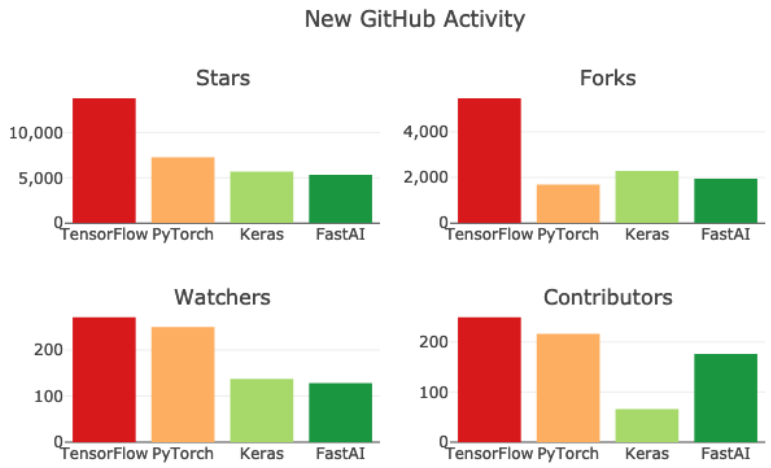


Figura 2.12: Actividad Git Hub

Con el tiempo, la tecnología ha ido evolucionando hasta establecerse como la librería de aprendizaje profundo más popular actualmente, véase Hale (2019). Gracias a las cualidades anteriormente mencionadas, su aplicación para la resolución de problemáticas en el campo del aprendizaje automático resulta muy eficaz, permitiendo un entorno de diseño abierto de este tipo de estructuras computacionales como son los modelos numéricos.

TensorFlow permite generar modelos matemáticos mediante un modelo de cómputo basándose en un grafo dirigido compuesto por nodos unidos entre si, donde los nodos representan las funciones que se aplican sobre los datos (números, matrices o tensores) representados como las uniones entre estos nodos. De este modo, se permite generar un flujo de cómputo dirigido a través de los grafos de la figura 2.13 emulando el modelo matemático.

Como su propio nombre indica, TensorFlow lo que pretende es un flujo de tensores, siendo el tensor el dato que fluye a través del grafo dirigido. Un tensor se puede interpretar como un conjunto perteneciente a un espacio multidimensional. Por tanto, un dato numérico real, ya sea un escalar, un vector o una matriz multidimensional, puede ser representado como un tensor según lo siguiente:

- Los tensores son representaciones multilineales de espacios vectoriales al conjunto de números reales(V espacio vectorial y V^* espacio dual)

$$f : \underbrace{V^* \times \cdots \times V^*}_{p \text{ copias}} \times \underbrace{V \times \cdots \times V}_{q \text{ copias}} \longrightarrow \mathbb{R}$$

- Un escalar es un tensor ($f : \mathbb{R} \longmapsto \mathbb{R}, f(e_1) = c$))
- Un vector es un tensor ($f : \mathbb{R}^n \longmapsto \mathbb{R}, f(e_i) = v_i$))
- Una matriz es un tensor ($f : \mathbb{R}^n \times \mathbb{R}^n \longmapsto \mathbb{R}, f(e_i, e_j) = A_{ij}$))

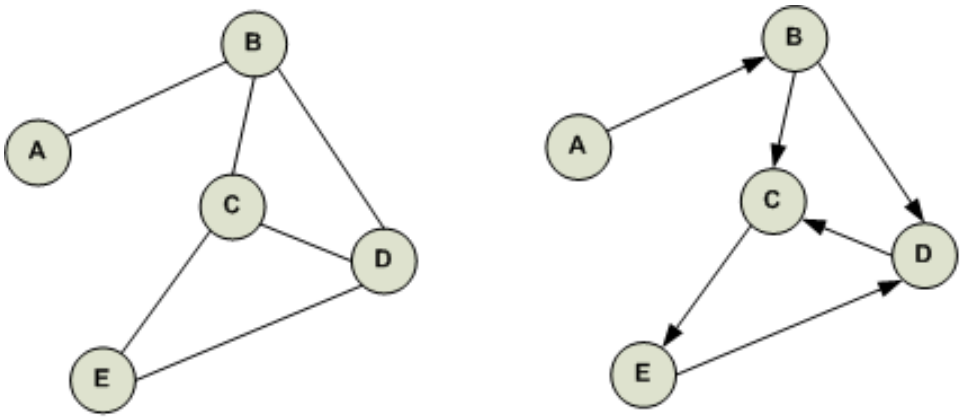


Figura 2.13: Grafos no dirigido (izq.) y dirigido (der.)
Fuente AlgorithmsInside (2019)

Así pues, para este trabajo, se interpretará al tensor como un *array* numérico n-dimensional, lo cual se puede generar de forma sencilla gracias al lenguaje de programación Python dadas sus estructuras de datos y tipos. Tensorflow, realizando cálculos de coma flotante, podrá interpretar estos *array* con longitudes de 8-bit, 16-bit, 32-bit y 64-bit. La integración de este tipo de datos con los *array*s de la librería de Python, Numpy, permite agregar versatilidad al trabajar con ambas librerías como se ilustra en Abrahams et al. (2016).

En cuanto al flujo operativo, en el que las propias operaciones se alojan en los nodos, Tensorflow permite aplicar numerosas funciones en estos nodos y poder crear diversidad de flujos. Para el desarrollo de este trabajo, cabe destacar que TensorFlow es capaz de calcular funciones primitivas y gradientes, lo cual resulta necesario para un proceso de aprendizaje automático eficaz y eficiente. Por otro lado, TensorFlow no permite la generación de grafos circulares, lo que implicará que siempre se genere un sistema con una entrada y una salida. Se podrán generar grafos que cumplan esta condición, interconectados de diversas formas, pero finalmente, el grafo se deberá *desenrollar* de forma que se pueda generar una ejecución secuencial de operaciones matemáticas.

2.4.1. TensorFlow y la ejecución del grafo dirigido

Una vez se haya generado un grafo dirigido Tensorflow se deberá establecer un conjunto de datos de entrada numérico para que sea operado al pasar a través de él. La ejecución del

grafo se realizará con estos datos dentro de una sesión de TensorFlow una vez se lance la orden de ejecución del grafo dentro de la sesión. Esto implica que hasta que no se dé esta orden, no es posible realizar la ejecución del modelo matemático. Este tipo de ejecución se denomina ejecución **perezosa**.

Así pues, con esta sesión Tensorflow se está estableciendo un entorno cerrado de cómputo con unas variables (tensores) que van a ser modificados hasta generar un resultado a la salida del grafo dirigido definido. En este trabajo se generarán grafos dirigidos de diversa naturaleza gracias a TensorFlow para cumplir con unos fines específicos y obtener unos resultados convenientes, en este caso, generando algoritmos de aprendizaje automático como son las redes neuronales.

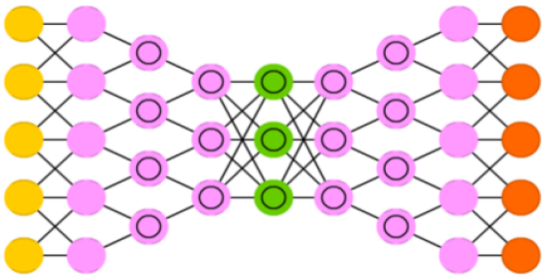


Figura 2.14: Red generativa adversaria (GAN)

2.4.2. TensorFlow y las redes neuronales

Como se ha mencionado anteriormente, los grafos computan de forma secuencial, generando una salida a partir de una entrada. Este procedimiento es equiparable al que realiza una red neuronal ya que se puede exigir al grafo que determinadas partes operen de una manera y otras de otra manera esperando una entrada para generar una salida. En una red neuronal cada una de estas partes es una capa formada por un numero determinado de nodos (neuronas) que operan de forma idéntica para generar una salida similar en función de los datos de entrada de cada nodo. Por tanto, generando una dependencia directa entre estas capas, o subgrafos, se podrá crear una estructura en la cual se ejecuten operaciones hasta generar un resultado cuando no existan más dependencias, lo cual, con TensorFlow es posible generar. Para finalmente poder generar una red neuronal es preciso que cada uno de los resultados de estas dependencias sean almacenados, es decir, se realice un seguimiento de los valores de cada uno de los nodos del grafo, lo cual, TensorFlow también permite, véase Abrahams et al. (2016).

2.4.3. TensorFlow multiplataforma

Una característica diferencial frente a otras librerías de Machine Learning, es que TensorFlow se puede ejecutar en diversos dispositivos hardware; mientras el resto de librerías hasta la fecha se ejecutaban únicamente en CPUs, esta librería permite ejecutarse en CPUs, GPUs y TPUs (Tensor Processing Unit) además de en dispositivos móviles gracias a su versión TensorFlow Lite. Esta última variante permite realizar procesos de inferencia sobre sistemas con poca capacidad computacional como *smartphones*, mini-computadores como la RaspberryPi o dispositivos IoT como se describe en la página web TensorFlow (2019); esto se conoce como escalabilidad hacia abajo. Pero para el proceso de entrenamiento, que conlleva mayor exigencia computacional, ha sido esencial permitir compilar el código a lenguaje de GPUs. Esto se realiza mediante el lenguaje CUDA y permite aprovechar las

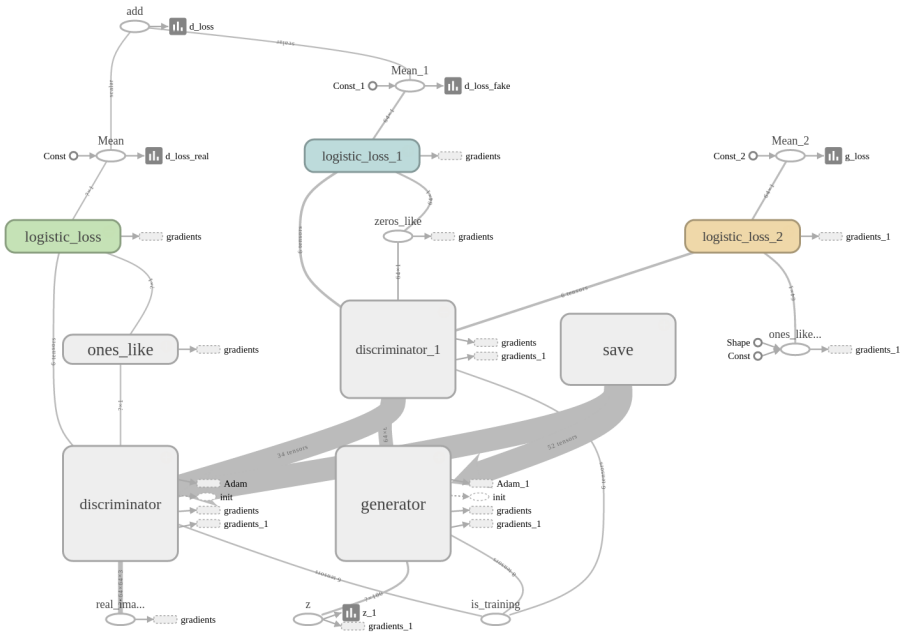


Figura 2.15: GAN visualizada mediante Tensorboard
Fuente Goodfellow (2017)

características estructurales de estos dispositivos para acelerar los procesos más exigentes computacionalmente. Esta característica es clave para la realización de este trabajo y se estudiará y evaluará en profundidad en la sección 3.5.

Figura 2.16: Aplicaciones móviles que utilizan Tensorflow
Fuente TensorFlow (2019)

2.4.4. TensorFlow y la ejecución distribuida

Una vez explicada la forma de operar de TensorFlow mediante grafos dirigidos, cabe destacar la versatilidad y escalabilidad que aporta poder partir el grafo. Gracias a esto, se podrá llevar a cabo la estrategia "divide y vencerás" de forma que el ahorro de tiempo en realizar las operaciones sea considerable cuando se pueda distribuir el cómputo entre varios dispositivos hardware. En lo que respecta a las redes neuronales y su proceso de entrenamiento particularmente, destacar que es posible segmentar la red sobre distintos nodos de cómputo para que operen su parte del grafo siguiendo unas directrices globales. Así pues, una vez terminada la ejecución en cada uno de ellos, se podrá reconstruir el grafo determinando los valores de los nodos de forma conjunta gracias a TensorFlow. Esta característica ha permitido que modelos matemáticos con grandes cantidades de parámetros puedan ser computados en un tiempo razonable hasta la fecha, ya que antes de que apareciera TensorFlow procesar estos modelos era plenamente ineficiente. Para ello, generalmente, se aprovisiona el entorno de TensorFlow con numerosas GPUs o TPUs que de forma síncrona permiten realizar entrenamientos de redes neuronales profundas. En

el capítulo 3 se hará una descripción más profunda de las técnicas que aporta la API *tf.distribute()* de esta librería.

2.4.5. Tensorboard y el perfilado

TensorFlow cuenta con una herramienta de visualización interactiva mediante un navegador web que permite analizar los modelos generados y cómo realizan sus operaciones. A lo largo del trabajo se mostrarán imágenes de diferentes topologías de redes neuronales extraídas de esta herramienta. Tensorboard también permite visualizar parámetros matemáticos que monitorizan el proceso de entrenamiento de un modelo, como la precisión, el MSE, etc. Gracias a esto podremos dilucidar cómo está aprendiendo una red neuronal, y ver qué recursos de la plataforma de cómputo se están consumiendo en ese proceso: memoria, procesador ...

Para la realización del perfilado de redes neuronales en este trabajo se ha recurrido a extraer trazas de cálculo del grafo para poder visualizar con alta precisión el tiempo que implica cada operación. Gracias a el navegador Chrome y su herramienta Chrome Tracing se pueden cargar y visualizar ficheros *json* con las trazas generadas durante los procesos de entrenamiento o inferencia de las redes neuronales en TensorFlow.

Estas dos herramientas se utilizarán para interpretar dónde y cómo se realizan las operaciones del software sobre el hardware y serán esenciales en el desarrollo de los procesos de evaluación temporal así como de la comprensión del funcionamiento de una red neuronal.

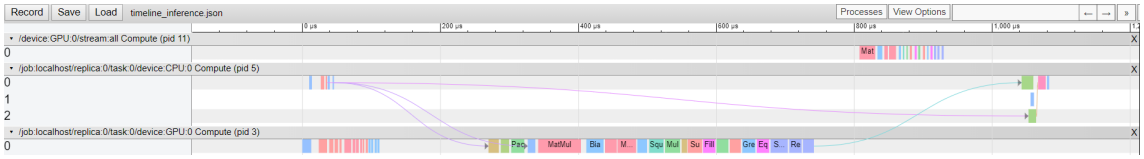


Figura 2.17: Trazas de un proceso de inferencia con aceleración por GPU

2.4.6. TensorFlow y este trabajo

Como se ha descrito anteriormente, este software aporta numerosas herramientas y funcionalidades para poder acometer un proceso de aprendizaje automático con alto rendimiento. Ha sido el instrumento más utilizado en este proyecto y sobre el que se han generado mayor número de aplicaciones junto con Python. Sin embargo, debido a la efervescencia del entorno del *Machine Learning* en el mundo globalizado donde el software libre juega un papel fundamental en el desarrollo de nuevas tecnologías, ha surgido una librería que permite acelerar la generación de modelos, permitiendo agilizar las operativas que en TensorFlow pueden presentarse complejas. Si bien este trabajo ha utilizado TensorFlow nativo para numerosas tareas, también recurre esta librería, Keras, para realizar procesos similares. Actualmente la versión 2.0.0-beta1 de TensorFlow incorpora Keras de forma nativa.

2.5. Keras

Keras es una librería de software libre para redes neuronales escrita en Python por François Chollet, del equipo de Google Brain, y liberada en el año 2015. Su principal funcionalidad es agilizar la creación de estructuras de redes neuronales; sin embargo, requiere

de un *backend* (motor) para realizar cómputo sobre ellas. Siempre ha permitido integración por encima de Tensorflow, permitiendo facilitar la programación de los modelos con estructuras complejas para la sintaxis de Tensorflow. Así pues, en el año 2017, el equipo de Google Brain decide incorporar esta interfaz en Tensorflow. En la versión 2.0 de TensorFlow se ha integrado en el core permitiendo simplificar numerosas operativas.

Keras diferencia de forma sencilla para el usuario los distintos tipos de capas que se pueden incluir en una red neuronal, su conexionado y sus dimensiones. Permitirá crear de forma sencilla concatenaciones de diversos tipos de capas como las *fully connected* o las convolucionales; y parámetros como las funciones de activación, los optimizadores, o las funciones de *callback*. Se puede ver un ejemplo del resumen *summary* de un modelo en la figura 2.18. Es por esto por lo que para la parte de generación de un modelo basado en redes convolucionales se decidió utilizar Keras y realizar los procesos de entrenamiento e inferencia mediante esta API corriendo sobre Tensorflow.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_14 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

Figura 2.18: Keras con capas Dense y optimización Dropout
Fuente Stackoverlow (2018)

Sin embargo, debido a que la versión de Keras utilizada para este trabajo no incorpora funcionalidades para el perfilado exhaustivo de cada iteración del entrenamiento, los modelos de perceptrón se implementarán en Tensorflow nativo para poder monitorizar y visualizar de mejor manera las operaciones internas de los procesos de entrenamiento e inferencia de los modelos.

2.6. Computación acelerada por GPU

En los entornos de trabajo de la Inteligencia Artificial se requiere un soporte Hardware con una capacidad de cómputo considerable para poder acometer de forma eficaz los procesos de desarrollo de modelos. Es por este motivo principalmente por el que, desde hace poco más de una década, se han incorporado arquitecturas complementarias a las CPUs (*Central Proccessing Unit*) para poder acelerar los procesos de entrenamiento de las redes neuronales. Aunque las GPUs (*Graphic Proccessing Unit*) no nacieron con esta finalidad, sino para procesar señales de contenido visual, su arquitectura es muy eficiente para computar sobre tensores o matrices multidimensionales lo cual aportará mayor velocidad

a los procesos de entrenamiento de las redes neuronales.

Actualmente las CPUs han evolucionado a arquitecturas *multi-core* o multi-nucleo en las que se pueden distribuir tareas en paralelo sobre estos núcleos. Así pues, Intel, desarrollador global de procesadores, presenta a día de hoy como su tope de gama, Intel Xeon, procesadores que cuentan con hasta 28 núcleos. Sin embargo, esta arquitectura se presenta algo escasa cuando comparamos con los cores con los que cuenta uno de los modelos más modernos de las GPUs Nvidia: la Tesla V100 incorpora 5.120 *CUDA cores*. Si bien es cierto que estas arquitecturas pueden combinarse y ganar en capacidad de cómputo de diversas formas, es conveniente aclarar que **las GPUs se emplean como aceleradoras de los procesos y no como distribuidor de procesos, lo cual es tarea de la CPU**. Este concepto es esencial para el cómputo distribuido, aquí la CPU será la encargada de monitorizar las variables de entorno que se generan durante el entrenamiento de una red neuronal para finalmente poder conformar los valores de los pesos finales y la red entrenada en la GPU.

Una vez establecida esta operativa de cómputo en la que la CPU destaca por su procesamiento secuencial y la GPU por su procesamiento en paralelo, se pueden conformar plataformas heterogéneas entre ambas para ganar en eficiencia, siendo la CPU la que solicite acciones a la GPU según lo precise la aplicación software. Cada uno de los dispositivos cuenta con su propia memoria, por tanto, estas tareas también deberán adecuarse de forma conveniente a las limitaciones de almacenamiento del dispositivo. Sobre estas limitaciones se tratará en el capítulo 4 relacionándose con los modelos desarrollados y sus características.

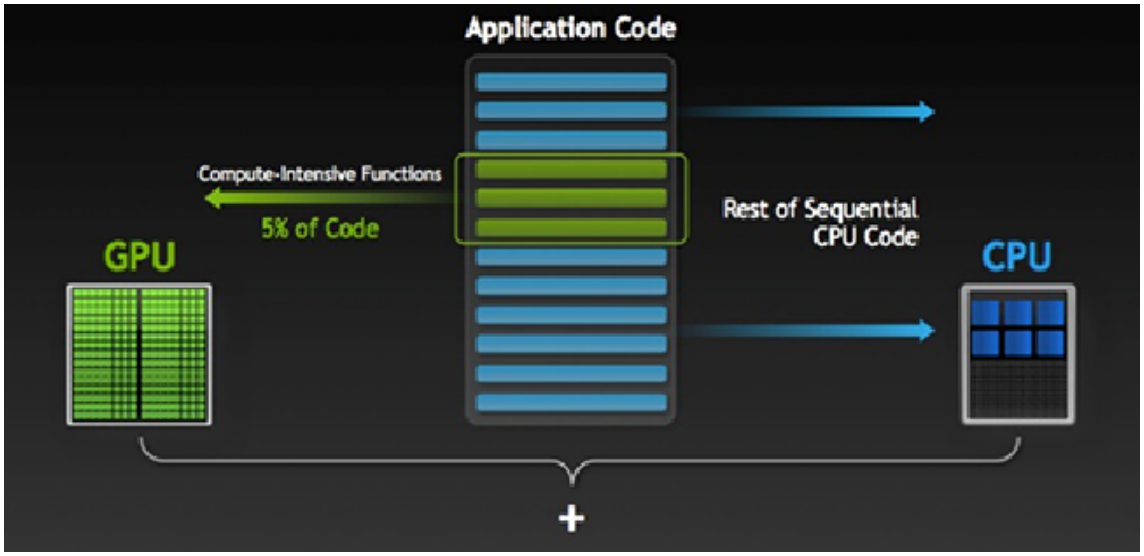


Figura 2.19: Tareas delegadas de la CPU a la GPU
Fuente Nvidia (2019a)

Ahora bien, para poder procesar tareas indicadas a través de un programa software, se debe poder compilar la aplicación a un lenguaje que sea comprensible por la GPU. A continuación se detallarán algunos aspectos relacionados con esta tarea y su principal desarrollador.

2.6.1. Nvidia CUDA

La empresa multinacional Nvidia ha puesto especial atención desarrollando un numeroso conjunto de herramientas y librerías que permiten compilar el código de forma eficaz

para que sea ejecutado sobre sus GPUs. Nvidia es el mayor vendedor de GPUs en el mundo con una cuota de mercado de en torno al 80 %, según se afirma en MarketRealist (2019), además, ha desarrollado una estrategia comercial centrada en la Inteligencia Artificial siendo partícipe de su crecimiento en diversos campos como la conducción autónoma, Volvo (2019) o la visión por ordenador. Para este tipo de fines comercializa la Nvidia DGX-1 para procesos de altos requerimientos, computacionales como los procesos de entrenamiento de grandes redes neuronales, y dispositivos optimizados para inferencia como los modelos Nvidia Jetson de los que hay más información en Nvidia (2019c).

En cuanto a lo que software se refiere, Nvidia desarrolla la plataforma CUDA (*Computed Unified Device Architecture*) Nvidia (2019b), entorno que ofrece diversas funcionalidades pensando en la computación de propósito general mediante la aceleración con GPUs. Entre ellas están la química computacional, la ciencia de datos o la bioinformática; pero en este trabajo se usarán las funcionalidades que aporta para el aprendizaje automático. Así pues, es conveniente destacar algunos de los complementos que aporta este kit de herramientas:

- cuDNN (*CUDA Deep Neural Network*): para la aceleración del cálculo de funciones primitivas para redes neuronales profunda y otras rutinas similares.
- cuBLAS (*CUDA Basic Linear Algebra Subroutine*): para álgebra lineal sobre dispositivos con CUDA.
- cuPTI (*CUDA Performance Tool Interface*) : para incorporar herramientas de perfilado y trazas sobre objetivos CUDA.
- NVCC (*Nvidia CUDA Compiler*): compilador a lenguaje C o a Microsoft Visual C para ser interpretado por CPUs o GPUs respectivamente.

Gracias a sus librerías CUDA permite ventajas como permitir consultar cualquier dirección de memoria, memoria compartida, velocidad en tiempos de lectura por parte de la GPU y operaciones a nivel de bit. Su modelo se basa en aprovechar el paralelismo operacional de las GPUs y el gran ancho de banda de las memorias. De este modo, en situaciones de gran coste aritmético, se evita realizar numerosos accesos a la memoria principal y evitar cuellos de botella. El componente esencial es el hilo de ejecución, que se computa sobre bloques de hilos con las mismas dimensiones que el hilo con un preciso sistema de sincronización. Todos los hilos pueden realizar distintos comportamientos de acceso a las distintas memorias que incorporan las GPUs, como la propia del hilo, las compartidas o la global.

A continuación, en la figura 2.20 se muestra el flujo de operación sobre un dispositivo CUDA.

1. Copia de los datos a procesar de la memoria principal.
2. Orden de la CPU.
3. Ejecución en paralelo por parte de la GPU.
4. Copia del resultado de la memoria de GPU a la memoria principal.

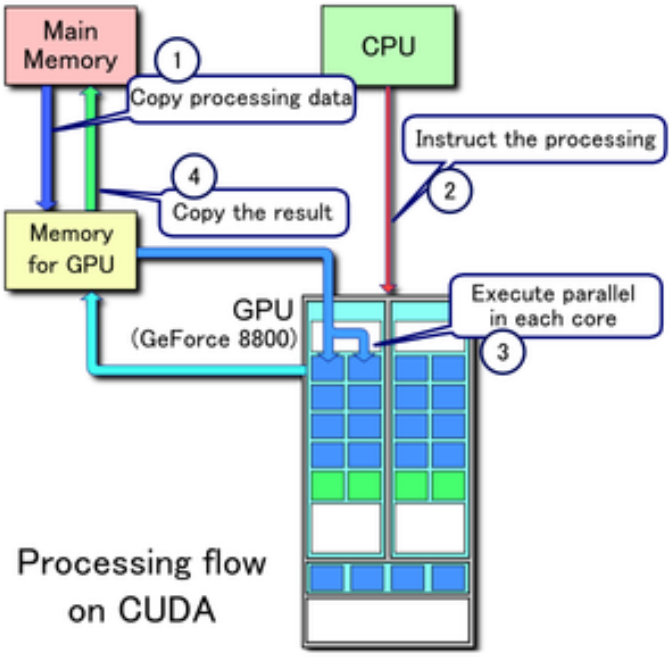


Figura 2.20: Flujo de procesado en CUDA
Fuente Wikipedia (CUDA)

Capítulo 3

Desarrollo de modelos basados en redes neuronales

En este capítulo se comienzan a describir los pasos que se han llevado a cabo para conseguir la evaluación temporal de redes neuronales sobre plataformas hardware heterogéneas. A partir de aquí se describirán las metodologías seguidas para generar modelos de *deep learning* basados en redes neuronales y las técnicas para monitorizar los procesos de entrenamiento e inferencia sobre ellas para finalmente poder realizar una evaluación conveniente. Para esto se ha realizado un diseño de software basado en el lenguaje de programación Python, sección 2.3, y las librerías TensorFlow, sección 2.4, y Keras, sección 2.5, que permitirá ejecutarse sobre plataformas hardware como CPUs y GPUs.

Este capítulo empieza detallando el entorno de trabajo y la tecnología utilizados para la realización de los desarrollos. Acto seguido detalla los pasos que se han seguido hasta poder realizar los procesos de entrenamiento e inferencia, empezando por una previa manipulación de los datos obtenidos mediante sensores IoT para adecuarlos a los requisitos de un aprendizaje correcto. Posteriormente describe la generación de redes neuronales mediante TensorFlow y Keras con diferentes hiperparámetros, véase la sección 2.2.3, para poder generar un entorno comparativo entre ellas. Una vez conseguidos los grafos de ejecución de esta manera, se detallarán los procesos de entrenamiento e inferencia prestando especial atención a sus requerimientos de procesamiento, según el tipo de topología de red neuronal. Gracias a esto se conseguirá generar un conjunto de datos que permitan, mediante las herramientas apropiadas, ser evaluados y decidir si la aceleración por GPU sería apropiada o no. El proceso de evaluación temporal se tratará de forma separada en el capítulo 4.

Todos los códigos, ficheros de configuración, ficheros de trazas y logs generados se encuentran en el repositorio del proyecto y se encuentran estructurados según se indica en el Apéndice D.

3.1. La estrategia seguida y el entorno de desarrollo

Este apartado detalla cómo se realizó el plan de desarrollo establecido al comienzo del proyecto y sobre qué tecnologías se llevó a cabo.

3.1.1. La estrategia seguida

La metodología de trabajo seguida puede resumirse en estos pasos:

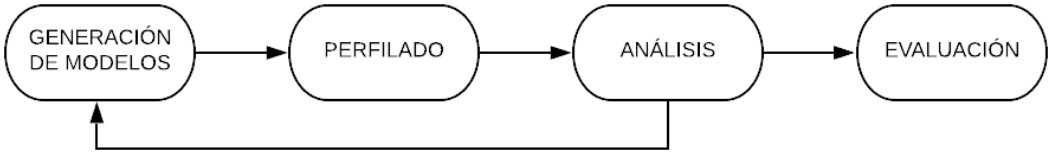


Figura 3.1: Estrategia utilizada

Para realizar este proyecto se parte de unos desarrollos software Python pertenecientes al Departamento de Arquitectura de Computadores y Automática, DACYA, y que permiten la predicción de valores de radiación solar en un punto de un área conociendo el valor en los puntos circundantes. Estos programas utilizan técnicas de aprendizaje automático que no son ejecutables en GPUs, por tanto, el primer paso de desarrollo de este trabajo ha sido codificar las mismas funcionalidades sobre un entorno que sí permita ejecuciones sobre GPUs.

Una vez se consigue la adaptación funcional, se inicia un nuevo proceso de desarrollo específico para permitir a TensorFlow y Keras generar información de la ejecución de los algoritmos de aprendizaje automático. Este proceso es el más largo en el tiempo del proyecto debido a la escasa experiencia del alumno con estas tecnologías; por tanto se precisó de una comprensión profunda de todo el proceso de aprendizaje automático, desde su naturaleza hasta su **implementación** con las mencionadas librerías. Una vez se consigue generar los modelos, entrenarlos y probarlos se extraen métricas temporales y computacionales mediante un proceso de **perfilado**.

Acto seguido se realiza un análisis exploratorio para ver la influencia de los hiperparámetros de los modelos en los tiempos de entrenamiento e inferencia. De este modo se generan nuevas topologías de redes neuronales hasta conseguir un conjunto de datos de perfilado conveniente que pueda ser evaluado mediante los conocimientos adquiridos durante el paso anterior. Para obtener las conclusiones se recurre al lenguaje de programación R para generar gráficas que permitan dilucidar de forma sencilla conclusiones acerca de los datos obtenidos.

3.1.2. El entorno de desarrollo

En este apartado se nombran los escenarios y herramientas con las que se han acometido los fines mencionados.

- Ordenador personal: en el dispositivo MSI PX60 2QD con sistema operativo Windows 10 se han desarrollado todos los programas Python y R sobre Visual Studio Code y RStudio respectivamente. Ha servido también como entorno de prueba de los programas TensorFlow sobre Python en el entorno Anacoda sobre su CPU IntelCorei7 y su GPU Nvidia GTX950M mediante los drivers CUDA. Además ha permitido la conexión remota, mediante SSH y con el cliente Linux para Windows 10, con los servidores del laboratorio Dacya para ejecutar sobre ellos los programas. También desde Windows y mediante un cliente Putty para SCP se ha permitido la transferencia de conjuntos de datos entre estos ordenadores.

- Laboratorio del DACYA: en este entorno es en el que se encuentran los dispositivos hardware con mayor capacidad de cómputo utilizados en este trabajo. Son la CPU IntelXeonE5 y las tarjetas gráficas Nvidia GTX1080 y Nvidia GTX980, contando

con dos de esta última. Sobre los servidores de este laboratorio se encuentra instalado CUDA. Se ha accedido a estos dispositivos por medio de los servidores Linux instalados con SSH habilitado y se han creado entornos Python para TensorFlow de forma genérica mediante pip y mediante Conda.

Para poder ampliar sobre las características de los componentes hardware mencionados en esta sección se puede acceder al Apéndice A. Para las herramientas software y su instalación al Apéndice B. Para ver la lista de comandos usados durante la realización del trabajo se puede acceder al Apéndice C.

3.2. Preprocesado de datos

Aunque este proyecto dista de la ejecución de la tarea del procesado de datos antes de su inyección al modelo, es imprescindible comprender los datos y su naturaleza. Para este trabajo se han utilizado conjuntos de datos facilitados en el laboratorio. Estos conjuntos han sido generados mediante el conjunto de herramientas Python creadas para dicho fin; estos programas se encuentran en el repositorio PythonToolchain de Piotr Parysek y sirven para filtrar, analizar, interpolar y extrapolar de forma básica cualquier conjunto de datos.

Los datos en crudo que se han pasado por este conjunto de programas de Piotr Parysek fueron datos provenientes de Centro de Medida e instrumentación de datos (MIDC) de Oahu, Hawai, Sengupta y Andreas (2010). Con esto resulta un conjunto de datos en formato tabla que cuenta con 50 variables (o columnas) de datos de radiación solar en diferentes puntos tomados cada segundo. Para este trabajo se generó una tabla con numerosas observaciones en formato .csv con la que poder llevar a cabo las tareas futuras.

Por otro lado, debido a las exigencias de TensorFlow de solo trabajar con datos en formato de *array* multidimensional, en este trabajo se ha requerido de una **adaptación del layout o forma de las tablas a matrices** que puedan ser computadas por las redes neuronales. Para ello se ha generado un *script* de Python *prepro_ convert_ to_ npy_ resize.py*, cuya localización se detalla en el Apéndice C, que permite seccionar y transformar estas tablas a formato .npy y así generar *arrays* de longitud deseada. Este programa puede actuar sobre los conjuntos de entrenamiento e inferencia, adaptando sus dimensiones convenientemente. Para el conjunto de etiquetas, se realiza un aplanamiento que no establezca la segunda dimensión al igual que ocurre en el conjunto de entrenamiento. Esto es transcendental para que durante el entrenamiento supervisado de las redes neuronales se pueda realizar una comparativa con los resultados del entrenamiento y calcular las métricas de error para el algoritmo de backpropagation. Este requisito dimensional es debido a las operaciones vectorizadas sobre *arrays* multidimensionales que realizan Numpy y TensorFlow para ganar en rendimiento computacional, véase McKinney (2012). Por tanto, gracias a este *script* se podrán generar ficheros más ligeros y preparados para su inyección directa en el modelo.

Debido a que también se ha trabajado con datos de imágenes que representan mapas de radiación solar generadas mediante la PythonToolchain en los que las dimensiones son de 50×50 variables, se ha generado otro *script* it Python nota-scriptIMAGENES para seccionar datos con estas dimensiones de forma similar al *script* anterior pero mediante un indexado en dos dimensiones.

A parte de estos procesos, se han extraído los códigos generados por Guillermo Yepes en su trabajo Infraestructura para la predicción de radiación solar a corto plazo localizados en el repositorio solarNodeCasting para aleatorizar la selección de muestras y *features* que se utilizan en el entrenamiento. Estas partes se han mantenido incorporadas en los *scripts*

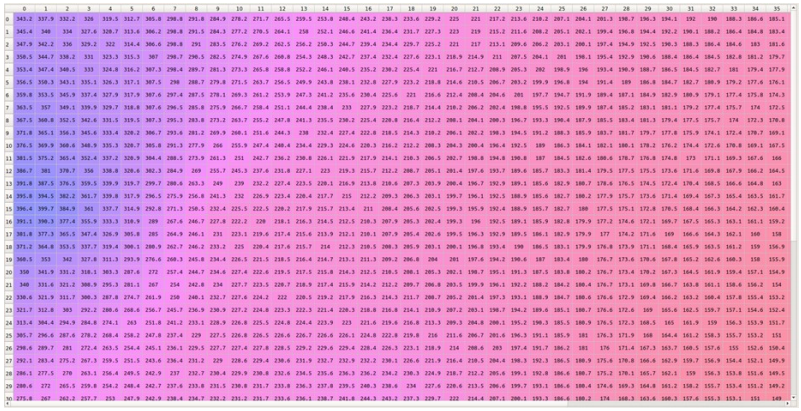


Figura 3.2: Mapa de radiación
Fuente PythonToolchain

principales que engloban todas las operativas TensorFlow y Keras localizables en el directorio /MAIN del repositorio del proyecto. Para los datos en forma de tabla se realiza una selección mediante columna o estación y para los de formato de mapa se realiza mediante imagen. También intervienen parámetros como el horizonte de predicción o la granularidad de las muestras seleccionados mediante los ficheros de configuración. Este proceso se conoce como regularización del conjunto de entrada de forma que el algoritmo sea capaz de generalizar y no se entrene demasiado para el conjunto de entrenamiento, lo que es conocido como *Overfitting*. Debido a que el objetivo de este trabajo no es evaluar la precisión del modelo o sus capacidades predictivas para otros conjuntos de datos, no se recurrirá a más herramientas de preprocesado. Únicamente se utilizarán las mencionadas para generar conjuntos de distinta longitud con sus dimensiones pertinentes y la aleatorización.

3.3. Generación de modelos con redes neuronales profundas

En esta sección se explica la generación de modelos de aprendizaje automático mediante los *frameworks* TensorFlow y Keras en base a los códigos que se puede pueden encontrar en el directorio MAIN del repositorio del proyecto, véase el Apéndice C. Para ello, primero habrá que describir la topología de red neuronal deseada para afrontar la problemática y posteriormente cómo implementarla. Como se han mencionado al principio del capítulo, la naturaleza de los datos es un factor esencial al construir la red, ya que son estos los que determinarán las dimensiones de la primera capa y, según el resultado esperado, las dimensiones de la capa de salida.

Para este caso se pretende resolver un problema de regresión o predicción de un valor real, por tanto, la capa de salida de las redes neuronales generadas será de una única neurona. Con esto se generará un único valor entre 0 y 1 que podremos identificar con un valor de radiación solar tras la desnormalización[1] de este valor resultante. En cuanto a la capa de entrada habrá dos posibilidades: para tratar con datos en formato de tabla la capa tendrá 50 neuronas y para tratar con datos en formato de imagen tendrá 50×50 neuronas. Estas dimensiones son las mismas que las de cada una de las muestras que se inyectarán a la red neuronal en los procesos de entrenamiento e inferencia. Debido a esta

[1]La normalización de los datos de entrada es un proceso de preprocesado que se aplica a los datos para establecerlos dentro de un conjunto reducido con la finalidad de agilizar y facilitar el cálculo en la red neuronal. En este caso se realiza mediante la PythonToolchain.

diferencia no será posible generar un único modelo que pueda tratar con ambos formatos de datos aunque la finalidad sea la misma. Por tanto, en el repositorio se cuenta con dos programas Python diferentes: *3fullyconnected_ nets_ profiler.py* con redes en TensorFlow para datos en formato tabla y *4keras_ CNN_ prof.py* con una red convolucional en Keras para el tratamiento de las imágenes.

La estrategia seguida para la conseguir la futura evaluación de los modelos en distintas plataformas hardware ha sido generar un *benchmark* de modelos para datos en formato tabla mediante TensorFlow modificando parámetros inherentes a todas las redes neuronales. Por otro lado, se ha adaptado el código de un modelo generado con Keras para poder realizar la comparativa temporal entre los distintos dispositivos. Gracias a esto se podrán diferenciar las funcionalidades de ambas librerías y sus herramientas para realizar una evaluación temporal de las redes neuronales. A continuación se amplían y relacionan los conceptos detallados en el capítulo 2 entre las redes neuronales y TensorFlow y Keras.

3.3.1. Generación de redes fully-connected con Tensorflow

En este apartado se detalla cómo se ha utilizado TensorFlow sobre Python para generar redes neuronales en las que todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente. Esto es así para que las redes generadas se comporten como el Perceptrón Multicapa implementado por Guillermo Yepes y permitan acometer un problema de regresión. El principal motivo por el que se realiza esta migración es debido a que este Perceptrón Multicapa no puede ejecutarse sobre GPUs al estar programado con la librería de Python scikit-learn que no lo permite.

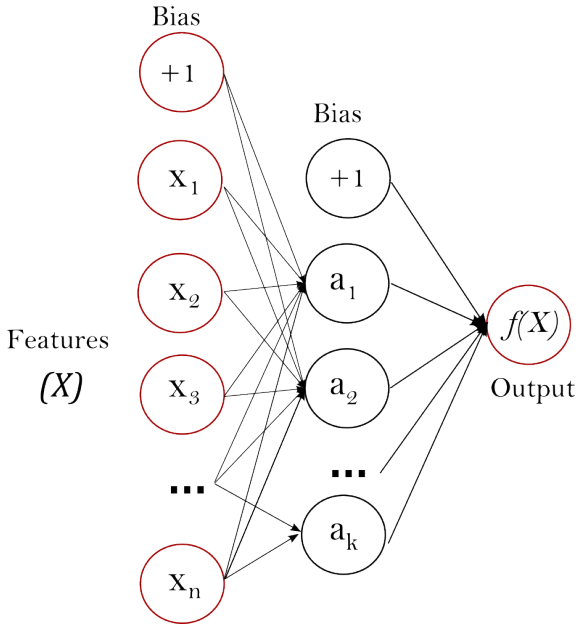


Figura 3.3: Perceptrón multicapa (MLP) con una capa oculta
Fuente Scikit-Learn

Puede apreciarse en la Figura 3.3, que el algoritmo *MLP* aprende una función no lineal de la forma $\mathbb{R}^m \longmapsto \mathbb{R}^o$ al entrenar sobre un conjunto de datos donde m es el número de dimensiones de la entrada y o es el número de dimensiones de la salida. La capa de la izquierda, la capa de entrada, representa las variables del conjunto de datos (en este trabajo serán 50) $\{x_i : x_1, x_2, ..., x_m\}$. En la capa siguiente, cada neurona aplica una una

suma ponderada a estos valores de la forma $w_1x_1 + w_2x_2 + ... + w_mx_m$, tras esto se aplica una función de activación que provoca una perturbación no lineal sobre esta salida de la forma $g : \mathbb{R} \longmapsto \mathbb{R}$. Finalmente la última capa recibe estos valores de la capa oculta y los transforma en valores de salida.

Ya se vio en la sección 2.2, esta estructura representa una red neuronal en la que las neuronas se establecen por capas de forma secuencial, por tanto, puede representarse mediante un grafo dirigido mediante Tensorflow, sección 2.4, y así ejecutarse sobre una GPU .

La API de TensorFlow con la que se han construido los modelos es la 1.12 y permite hacerlo de la manera ilustrada en la figura 3.4.

```
def fully_connected_model(input_tensor, hidden_layers):
       x = tf.layers.flatten(input_tensor)
       for layer in hidden_layers:
            x = tf.layers.dense(x, layer)
            x = tf.nn.relu(x)
       # Finally, connect the output to the right shape (1 output):
       x = tf.layers.dense(x, 1)
       return x
```

Figura 3.4: Código generador de red neuronal

Se observa inmediatamente en el extracto de código de la figura 3.4, que se está generando una estructura de red neuronal a partir de la variable x gracias a la función *tf.layers*. Primero se crea una capa con las dimensiones del conjunto de datos de entrada *input_ tensor* mediante la función *tf.layers.flatten* que aplana los datos manteniendo su eje. Luego se van añadiendo capas ocultas de tipo *fully connected* con el número de neuronas que presente el *array hidden_ layers* mediante la función *tf.layer.dense*. Tras cada capa oculta se aplica la función ReLU con *tf.nn.relu* hasta finalmente conectar con la capa de salida que presenta una única neurona.

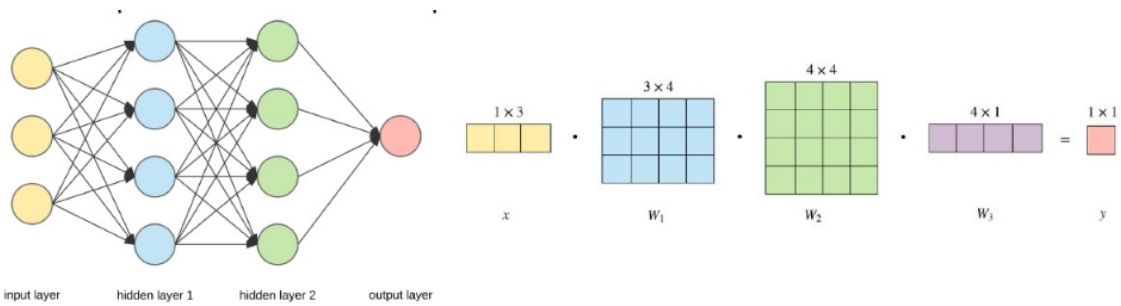


Figura 3.5: Ejemplo de red neuronal fully connected y transformaciones

Por ejemplo, generando una red neuronal para un *input_ tensor* de 3 *features*, dos capas ocultas de 4 neuronas cada una (*hidden_ layers* = [4,4]) y una única neurona en la capa de salida se obtendría lo que se muestra en la figura 3.5 de forma que los valores provocarán unas transformaciones del *array* de entrada de la siguiente manera.

Esta implementación replica la funcionalidad del *MLP* añadiendo capas con todas las neuronas conectadas entre sí y además permitiendo añadir el número de neuronas, el número de neuronas por capa y una perturbación no lineal mediante la función de activación ReLU, véase la sección 2.2.3. Dado que las funciones que aplican cada una de las neuronas a los datos que reciben como entrada es una suma ponderada de sus valores de entrada al

igual que el *MLP* es preciso comentar la funcionalidad de la función de activación elegida, la ReLU.

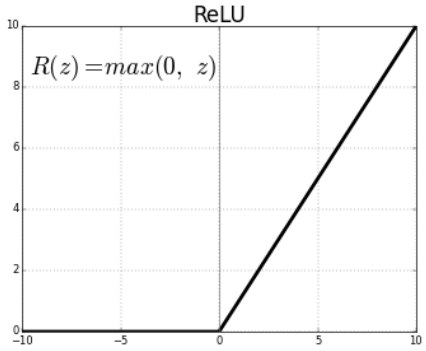


Figura 3.6: Función de activación ReLU
Fuente Google Imágenes

La figura 3.6 muestra la ReLU (*Rectified Linear Unit*) o Unidad lineal rectificada, que siendo una función no lineal, presenta un comportamiento lineal para valores de entrada positivos y 0 para los valores negativos. Es la más utilizada en redes neuronales convolucionales y en redes de aprendizaje profundo, puede consultarse Numerentur, debido a su ligereza computacional al ser su primitiva la función escalón. En contrapartida decir que aconseja su uso únicamente para la salida de las capas ocultas y que en ocasiones desactiva neuronas debido a que algunos gradientes pueden desaparecer durante el entrenamiento por su cercanía a la parte negativa. Para paliar esta última desventaja se puede utilizar su variante Leaky ReLU. Tras la aplicación de esta $f(x)$ al ejemplo de la red anterior se generaría la salida siguiente tras cada capa hasta llegar a la neurona de salida:

$$
\begin{aligned}
a_1 &= f(x \cdot W_1) \\
a_2 &= f(a_1 \cdot W_2) \\
y &= f(a_2 \cdot W_3) \\
&\Downarrow \\
y &= f\Big(f\big(f(x \cdot W_1) \cdot W_2\big) \cdot W_3\Big)
\end{aligned}
$$

Una vez definida la manera de generar una red neuronal con TensorFlow y sabiendo que las capas de entrada y salida son valores estáticos definidos por los datos y la salida esperada. Se ha generado una función Python que permita generar *arrays* de números de neuronas para las capas ocultas que está descrita en la figura 3.7.

```
def generador(red_1, n_redes = 1):
       n_1 = np.array(red_1)
       n_12 =np.copy(n_1)
       for x in range(0, n_redes-1):
            n_12 = n_12*np.array([2])
            n_1 = np.append(n_1, n_12)
       l = len(red_1)
       n_1 = np.reshape(n_1, (n_redes, l))
       return n_1
```

Figura 3.7: Función generadora de *arrays*

Con esto se consigue generar n redes de capas ocultas con el doble de dimensiones que las capas de la red introducida a la función en forma de *array*. Esta función ha sido utilizada para generar, de forma combinada con la función *fully-connected_ model*, distintos modelos de igual naturaleza pero con distintas dimensiones, es decir, presentan la misma funcionalidad pero difieren en el número de capas ocultas y en el número de neuronas por cada oculta.

Para este trabajo se ha decidido generar topologías de capas ocultas con estructura piramidal de forma que presente mayor número de neuronas la primera capa oculta que la última. Esta decisión es tomada para establecer un aprendizaje progresivo: según se está más próximo a la capa de salida el número de parámetros entrenables a computar por la red será menor. Esta es una estructura generalizada en la construcción de redes neuronales habilitando que las distintas capas identifiquen distintos patrones en los datos hasta obtener una salida perteneciente a un conjunto reducido de posibilidades, siendo para este caso una única salida. En contraste en una red generativa con la que se quiera producir una salida perteneciente a un conjunto de salida mayor que la entrada, las neuronas por capa irán aumentando progresivamente hasta las dimensiones de la salida esperada.

Con estas premisas y las herramientas implementadas, ya se pueden generar diversos modelos de aprendizaje automático mediante grafos Tensorflow. Lo que se ha construido en este trabajo es un *benchmark* o conjunto de 30 modelos (ver tabla 3.1), teniendo 10 con una capa oculta, 10 para dos capas ocultas y lo propio para tres; aquellos con más de una capa oculta presentando una estructura piramidal como se ha explicado. Gracias a este *benchmark* se podrá comparar cómo influyen la anchura y la profundidad de las redes neuronales en las exigencias computacionales y en los tiempos de entrenamiento e inferencia sobre CPUs y GPUs. En la sección 3.4.1 se detalla cómo y porqué influyen en el proceso de entrenamiento.

Una capa oculta	Dos capas ocultas		Tres capas ocultas		
50	50	25	50	25	13
100	100	50	100	50	26
200	200	100	200	100	52
400	400	200	400	200	104
800	800	400	800	400	208
1600	1600	800	1600	800	416
3200	3200	1600	3200	1600	832
6400	6400	3200	6400	3200	1664
12800	12800	6400	12800	6400	3328
25600	25600	12800	25600	12800	6656

Tabla 3.1: Distribución de neuronas por capa en las redes neuronales

3.3.2. Generación de una red convolucional con Keras

Según se detalló en la sección 2.5, Keras permite generar redes neuronales de forma más sencilla e intuitiva que Tensorflow. Para este trabajo se han modificado los códigos del trabajo Aplicaciones de *Deeplearning* a la predicción de radiación solar de Ahsin Rashid localizados en el repositorio solarcasting-ml para obtener los tiempos de entrenamiento e inferencia de la red convolucional que implementa. Para ello ha sido necesario estudiar la API de Keras y comprender los pasos necesarios para generar el modelo, entrenarlo y

guardarlo. En este apartado se explica cómo generar un modelo de red convolucional mediante Keras para posteriormente poder extraer los tiempos de entrenamiento e inferencia. También se detallan los parámetros a entrenar para el modelo. Como este modelo será el único que se utilizará se van a detallar aquí los parámetros que influirán en el proceso de entrenamiento, sin embargo, para las *fully connected* se explica en el apartado siguiente ya que son modificables.

La red convolucional consta de dos partes diferenciadas. En primer lugar, la red aplica un conjunto de herramientas para el tratamiento de imágenes representadas mediante filtros o capas convolucionales; gracias a esto consigue identificar las variables o píxeles del plano que resultan convenientes para representar la información inicial de una forma más compacta. Una vez extraídas las variables más representativas, se aplanan y se pasan por un perceptrón multicapa que permita realizar una clasificación, ver en la figura 3.8, o una regresión cuando la capa de salida presente una única neurona como es el caso de este trabajo. A continuación se detalla cómo funcionan estas primeras capas encargadas de seleccionar las *features* antes de pasar los datos a una red *fully connected*.

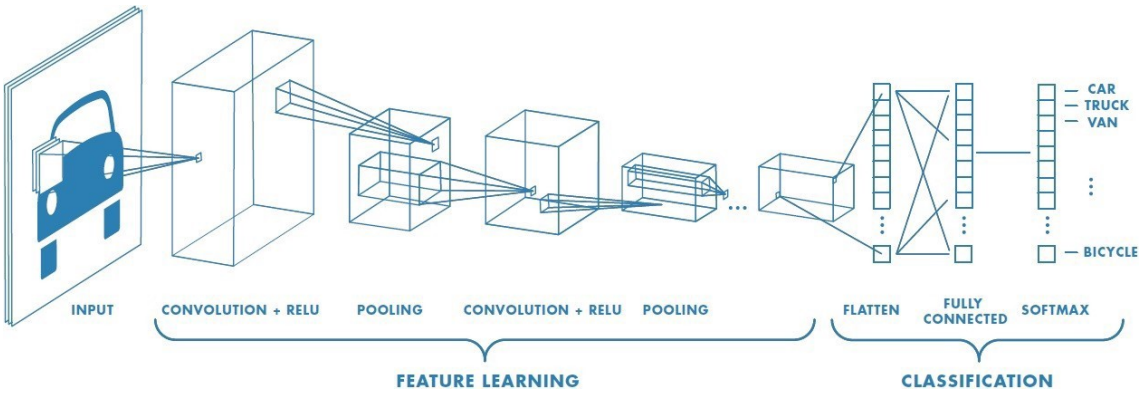


Figura 3.8: Red neuronal convolucional (CNN). Fuente Google Imágenes

La capa convolucional *Conv2D* se diferencia de la Dense en que mientras esta aprende patrones globales de todo el espacio de entrada, la convolucional aprende patrones localizados en ventanas del conjunto. De este modo la red recordará y reconocerá estos patrones cuando aparezcan de nuevo. Esto se consigue sin que todas las neuronas de la capa anterior estén conectadas a las neuronas de la capa siguiente, sino que serán las neuronas dentro de las dimensiones de la ventana las que se conectarán a una siguiente neurona. Esto es muy útil en el tratamiento de datos multidimensionales como las imágenes.

Como se cuenta con mapas de calor, imágenes en definitiva, se usará esta técnica para llevar a cabo la regresión sobre estos datos. El modelo está construido mediante la función *Sequential* de Keras que permite concatenar capas y funciones mediante el código de la figura 3.9.

Destacar que la mencionada ventana o *kernel* se interpreta como un filtro de dimensiones $kernel\_size$ que se aplicará *filters* veces sobre la imagen de entrada que en este caso es de 50×50. Por tanto el número de parámetros entrenables será

$$N = kernel\_size * filters + filters = 3 \times 3 \times 32 + 32 = 320. \tag{3.1}$$

Como se puede apreciar en la figura 3.10, las dimensiones de la matriz de salida vienen determinadas por las dimensiones del Kernel; en este caso, será de dos unidades menos en la horizontal y dos menos en la vertical ya que este nunca podría quedar fuera de la

```
nn_model = Sequential()
nn_model.add(Conv2D(filters = 32, kernel_size=(3, 3),
          activation='relu',input_shape=input_shape))
nn_model.add(Conv2D(64, (3, 3), activation='relu'))#
nn_model.add(MaxPooling2D(pool_size=(2, 2)))
nn_model.add(Dropout(0.25))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dropout(0.5))
nn_model.add(Dense(num_classes, activation='relu'))
```

Figura 3.9: Concatenación de capas

imagen. El resultado es una capa de 48×48 neuronas que puede apreciarse en el resumen del modelo en la tabla 3.2, tras esta operación sobre una capa de entrada de 50 valores.

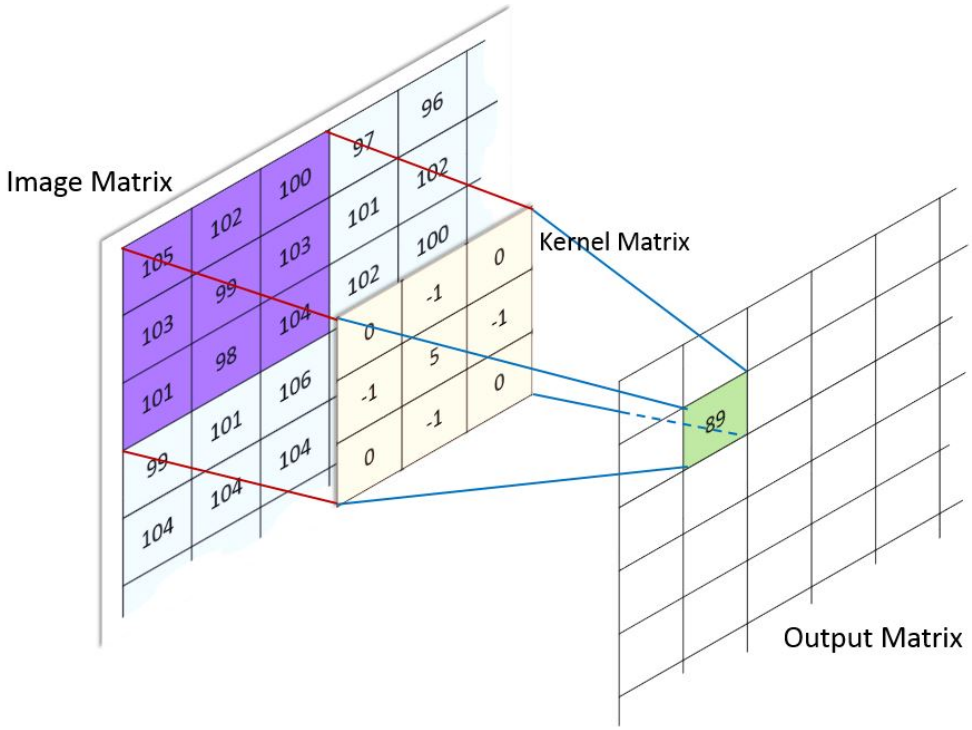


Figura 3.10: Filtrado mediante un Kernel 3× 3
Fuente Arsham

La fórmula matemática para generar el valor de salida es

$$(f * g)(i) = \sum_{j=1}^{m} g(j) f\left(i - j + \frac{m}{2}\right). \tag{3.2}$$

Para la siguiente capa convolucional se realiza la aplicación de 64 filtros del mismo tamaño sobre las 32 imágenes generadas por la aplicación de los filtros anteriores. Por

tanto, el número de parámetros entrenables será de:

$$\begin{aligned} Parametrosentrenables &= filtroscapaanterior \times filtros \times kernel\_size + filtroscapa \\ &= 32 \times 64 \times 9 + 64 = 18496. \end{aligned} \tag{3.3}$$

Acto seguido se aplica un *pooling* sobre 2x2 valores de forma que se selecciona el valor máximo de ese grupo de cuatro valores según se observa en la figura 3.11.

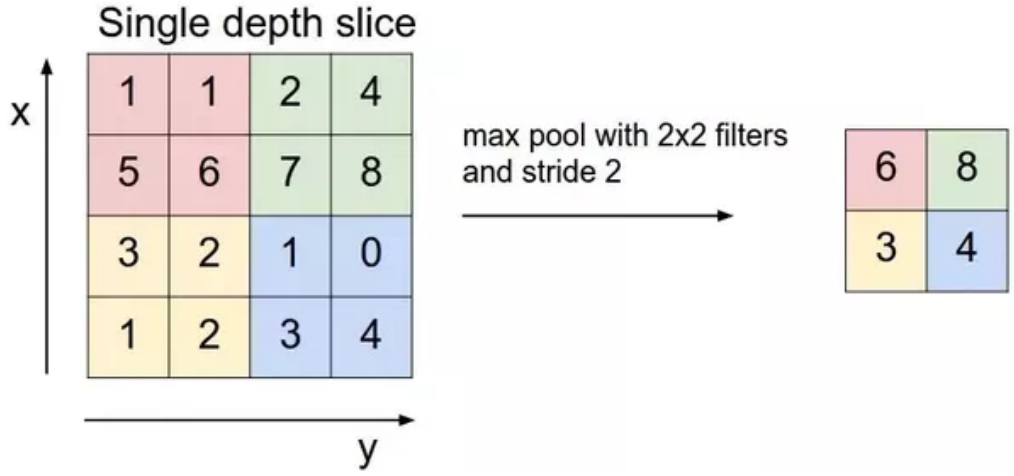


Figura 3.11: Ejemplo de MaxPooling2D
Fuente Quora

Tras esto se realiza un proceso de descarte del 25 % de las neuronas mediante un proceso de *dropout*[2], se aplana la salida mediante la función *Flatten* y se pasa por una capa *Dense* más una ReLU que implica que el número de parámetros sea:

$$Parametros = Ncapaanterior \times Ncapa + Ncapa = 33856 \times 128 + 128 = 4333696 \tag{3.4}$$

Finalmente se añade otro *dropout* del 50 % de las neuronas y se llega a la capa de salida que presenta como valor *num_classes* el valor 1. Aquí el número de parámetros a entrenar será de:

$$Parametros = Ncapaanterior \times Ncapa + Ncapa = 128 \times 1 + 1 = 129. \tag{3.5}$$

Con la librería de Keras es necesario realizar la compilación del objeto modelo *nn_model* tras su construcción con las métricas de precisión que se quieran aplicar sobre el mismo y el optimizador a utilizar en el entrenamiento. Una vez hecho esto el modelo estará generado y listo para aprender como se verá en la sección 3.4. Otra funcionalidad muy útil de Keras es que con su función *summary* se puede obtener la información del modelo compilado como se muestra en la tabla 3.2 que aparece a continuación.

3.4. Entrenamiento e inferencia de redes neuronales

Una vez generados los modelos se debe realizar el proceso de entrenamiento para ajustar los pesos de las neuronas que lo componen. Como se explica en la sección 2.2.4, esto

[2]El *Dropout* es una técnica de regularización patentada por Google para reducir el *overfitting* en redes neuronales.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 48, 48, 32)	320
conv2d_2 (Conv2D)	(None, 46, 46, 64)	18496
max_pooling2d_1 (MaxPooling2)	(None, 23, 23, 64)	0
dropout_1 (Dropout)	(None, 23, 23, 64)	0
flatten_1 (Flatten)	(None, 33856)	0
dense_1 (Dense)	(None, 128)	4333696
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 1)	129
Total params: 4,352,641 Trainable params: 4,352,641 Non-trainable params: 0		

Tabla 3.2: Información sobre el modelo compilado

es un proceso complejo que recurre al algoritmo de descenso de gradiente y al algoritmo de *backpropagation* para realizar un aprendizaje supervisado. En esta sección se detalla cómo repercuten estos reajustes de pesos en las operaciones computacionales que deberá hacer Tensorflow, se indicarán los hiperparámetros influyentes en el proceso y se aportarán ejemplos que lo demuestren. Al igual que en la sección anterior, se separará la parte implementada con TensorFlow de la parte de Keras. Finalmente se mostrarán las formas de realizar una inferencia con los modelos ya entrenados para realizar predicciones. Gracias a esto, ya se tendrán unos procesos que monitorizar para la obtención de métricas que nos permitan realizar la evaluación pertinente sobre las diferentes plataformas hardware.

3.4.1. Entrenamiento e inferencia de redes *fully connected* con Tensorflow

En este apartado se hace una breve explicación de cómo se puede realizar un proceso de entrenamiento por lotes de una red neuronal, o grafo dirigido, mediante Tensorflow. Una vez se dispone de los modelos con los pesos aleatoriamente inicializados, hay que propiciar que la propia red neuronal los ajuste de forma inteligente. Esto en TensorFlow se debe realizar dentro de una sesión como se indica en el apartado 2.4.1. En la figura 3.12 se muestra el grafo de un modelo de red neuronal con tres capas ocultas dense con todas las operaciones implícitas para el entrenamiento.

Para realizar el entrenamiento de los modelos generados se ha partido del tutorial Profiling TensorFlow de Corey Adams. Adaptado a la casuística de este trabajo, se va a generar una función denominada *trainer* que permita incluir las llamadas a las funciones implementadas para la generación de modelos y también los parámetros necesarios para el entrenamiento e inferencia elegidos en este trabajo. En esta función se generará una sesión TensorFlow, que se ejecutará sobre la red neuronal y permitirá generar un grafo, o modelo entrenado, con los pesos ya ajustados tras el paso de los datos por él. Además permitirá indicar dónde se realiza todo este proceso: sobre CPU o sobre GPU (argumento *FORCE_CPU*). La cabecera de la función es la siguiente:

```
def trainer(BATCH_SIZE, LEARNING_RATE, LOGDIR, OPT, MODEL, NN, EPOCHS,
      NAME, FORCE_CPU=False):
```

Para inyectar datos externos al grafo de TensorFlow en la versión con la que se realiza este trabajo es necesario definir unos nodos que actúen como puntos de entrada. Esto se consigue mediante las variables *tf.placeholder* que actuan como nodos *dummy* (emuladores)

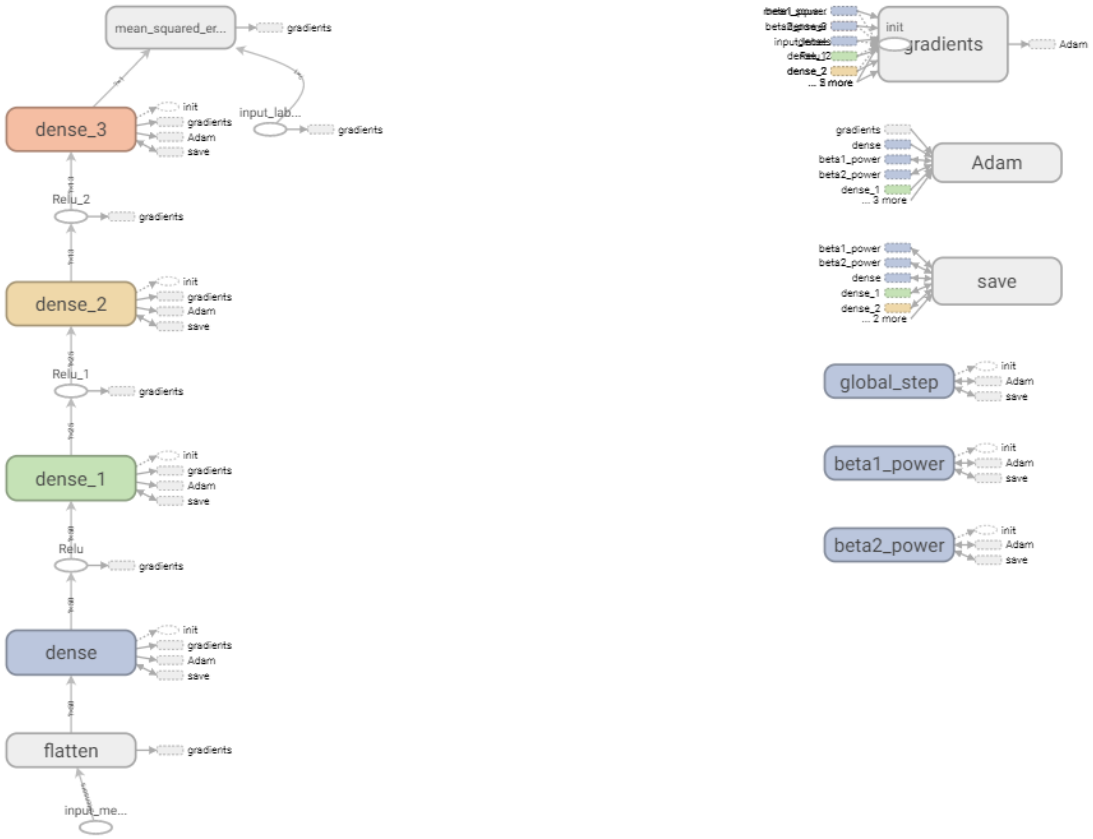


Figura 3.12: Grafo Tensorboard de una red neuronal *fully connected*

para introducir los conjuntos de entrenamiento e inferencia adecuadamente. Una vez generados estos nodos con los datos se inyectan al grafo mediante un *feed dictionary* cuando se desee ejecutar la sesión, y así realizar el entrenamiento o inferencia, mediante la llamada a `session.run()`. El *feed dictionary* es un diccionario python que simplemente mapea los valores de los *arrays numpy* recibidos. Como el proceso de entrenamiento e inferencia que se desean realizar es por lotes, se ha establecido que este *feed dictionary* se vaya generando según la longitud *BATCH_SIZE* de muestras del conjunto. Esto es un concepto circunstancial para entender cómo se van a realizar las operaciones de los procesos de entrenamiento e inferencia y qué implicación tiene en el tiempo de realización de las mismas.

Ya se ha mencionado anteriormente que en este trabajo no se pretende realizar varios recorridos sobre el conjunto de entrenamiento o *EPOCHS*, por tanto se va a fijar este argumento de la función *trainer* a 1. Sin embargo sí que se va a realizar un entrenamiento por lotes realizando un indexado de los conjuntos de la siguiente manera al construir el *feed dictionary*.

```
   # Construct a feed dictionary
fd = {
input_tensor : x_train[data_access_index:data_access_index+BATCH_SIZE],
input_labels : y_train[data_access_index:data_access_index+BATCH_SIZE]
}
```

El tamaño del *BATCH_SIZE* determina el número de muestras que tiene cada lote con el que se realiza el entrenamiento o inferencia. El número total de lotes es igual al número de pasos que deberán realizarse en ambos procesos hasta recorrer todo el conjunto

de entrada con la red neuronal, utilizando la función ceil que proporciona el menor entero mayor que el considerado.

$$\text{Pasos de entrenamiento} = \text{ceil}\left(\frac{\text{longitud de las muestras}}{\text{tamaño del lote}}\right) \tag{3.6}$$

Para poder realizar una comparativa de la influencia de este número de pasos en el entrenamiento se han seleccionado tamaños de lotes de 1000, 500 y 100 muestras. Resultando menor número de pasos cuando el tamaño del lote es menor.

Por otro lado, por cada uno de los lotes se realizarán un número de la transformaciones en función de la topología de la red neuronal (argumento *MODEL*) como se vio en la sección 3.3.1. Es decir, se realizarán tantas sumas ponderadas en función de las neuronas que presente cada red. En este aspecto es donde interviene el argumento de la función *trainer* NN, que utilizado para generar las topologías de red deseadas mediante *arrays* de capas ocultas condiciona el número de parámetros entrenables, pesos, que existirán al ejecutar el grafo. En la figura 3.13 se muestra un ejemplo de las transformaciones que se realizan sobre una red con 3 neuronas de entrada (equivalentes a tres *features* del conjunto de entrada), 4 neuronas en las dos capas ocultas y una única neurona en la capa de salida. Este proceso es idéntico en todas las redes neuronales *fully connected* implementadas variando en función de las neuronas y capas que determina *NN*.

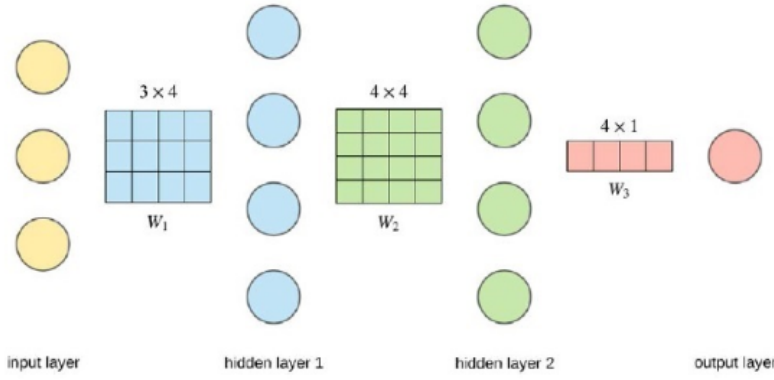


Figura 3.13: Transformaciones tras cada capa en una red neuronal

Anteriormente se ha indicado que la longitud de la capa de entrada viene determinada por el número de *features* del conjunto de datos de entrada y será 50 para todas las redes neuronales. También la capa de salida es de una neurona en todos los casos. Por tanto, para una misma topología de red el número de parámetros entrenables o pesos es el mismo. De este modo, variando la topología de red y con cuantas variables se alimenta a esta red se podrá valorar cómo se condiciona el número de operaciones a realizar, lo cual influye directamente en los tiempos. Teniendo en cuenta que el número de parámetros a entrenar es la suma de los productos de las neuronas de cada capa por las de la siguiente incrementados en el número de términos independientes, si c_i representa el número de neuronas de la capa i, se puede establecer la siguiente fórmula:

$$\text{Parámetros entrenables} = \sum_{i=1}^{n-1} c_i c_{i+1} + c_{i+1} = \sum_{i=1}^{n-1} (c_i + 1) c_{i+1} \tag{3.7}$$

que, aplicada al ejemplo de modelo de red neuronal considerada, establece que el número de parámetros a entrenar sería:

$$\text{Parámetros entrenables} = (3 \cdot 4) + 4 + (4 \cdot 4) + 4 + (4 \cdot 1) + 1 = 41.$$

Para el caso de este trabajo en el que las capas de entrada y salida siempre tienen 50 y 1 neuronas, la fórmula (3.7) se concreta en

$$\text{Parámetros entrenables} = 51c_2 + \sum_{i=2}^{n-2}(c_i + 1)c_{i+1} + (c_{n-1} + 1)$$

pudiéndose establecer que el número de parámetros a entrenar depende de la anchura de la red, debido a la cantidad de neuronas que pueda tener una capa; y también a la profundidad de la red, por el número de capas que pueda tener esta.

Para el proceso de entrenamiento por lotes es conveniente pasar como argumento un optimizador (argumento *OPT*), para ese trabajo solo se ha utilizado el *ADAM*. Para realizar el entrenamiento se ha generado una variable *tf.losses()* que permite incorporar este optimizador y así, durante la ejecución de la sesión mediante *session.run()*, minimizar el valor de la variable indicada tras cada lote. La variable elegida ha sido la métrica MSE que se irá intentando reducir tras la realización de las predicciones y la comparación con las etiquetas condicionando el reajuste de los pesos. Este optimizador no se utiliza para el proceso de inferencia ya que no se realiza la *backpropagation* ni el reajuste de los pesos. Sin embargo, la inferencia sí se realizará por lotes.

Finalmente los argumentos que podemos pasar a la función de entrenamiento *trainer* son el propio nombre del modelo deseado (*NAME*) y el directorio donde queremos que se guarde el grafo generado tras el entrenamiento (*LOGDIR*). También en este directorio se guardarán las métricas de monitorización generadas tras los procesos, los *logs* de Tensorboard y las trazas de ejecución. Sobre esto se hablará en la sección 3.6 ya que es el último paso necesario para realizar la evaluación.

Esta función *trainer* también realiza el proceso de inferencia sobre el conjunto de datos destinado para ello. Esto se realiza solo con el número de pasos *forward propagation* resultantes de aplicar la fórmula (3.6) sobre conjunto de datos denominado *test*. En este caso las comparaciones con las etiquetas de salida son determinantes en el resultado final del error cuadrático medio (MSE) final de un modelo llevado a producción. El modelo nunca ha recibido esas muestras de datos y debe poder realizar con ellas la predicción más cercana posible al valor esperado. En la figura 3.14 se muestran los valores predichos mediante un modelo de perceptrón y los valores esperados, o etiquetas, para un lote de 100 muestras donde se puede apreciar la precisión de la red neuronal para cada una de ellas.

El proceso de inferencia se realiza mediante otra llamada a `session.run()` en la que se aplica el modelo con los pesos ya ajustados en el grafo de la sesión sobre el conjunto de inferencia. Esto generará una predicción para cada una de las muestras que se pueden extraer de la sesión como un *array* Numpy para manejarse fuera de ella. Este proceso de extracción se ha realizado con el código *5checkeador.py* del repositorio del trabajo que realiza lo mismo que *3fullyconnected_nets_profiler.py* generando un fichero .csv con las predicciones realizadas. Además este proceso se ha separado del entrenamiento en los códigos *6inferencia_redes.py* de modo que se pueda hacer un proceso único de inferencia y extraer los tiempos directamente de cada uno de las redes con los pesos iniciados aleatoriamente. También, y de cara a poder realizar predicciones únicamente con un modelo ya entrenado, se ha implementado el código *7inferencia_grafo_importado.py*. Con este último *script* se cargan los valores de los pesos y la topología de la red a partir de los ficheros generados mediante la función *tf.train.Saver()* en el entrenamiento previo del modelo y almacenados en el directorio *LOGDIR/NAME/train/checkpoints*.

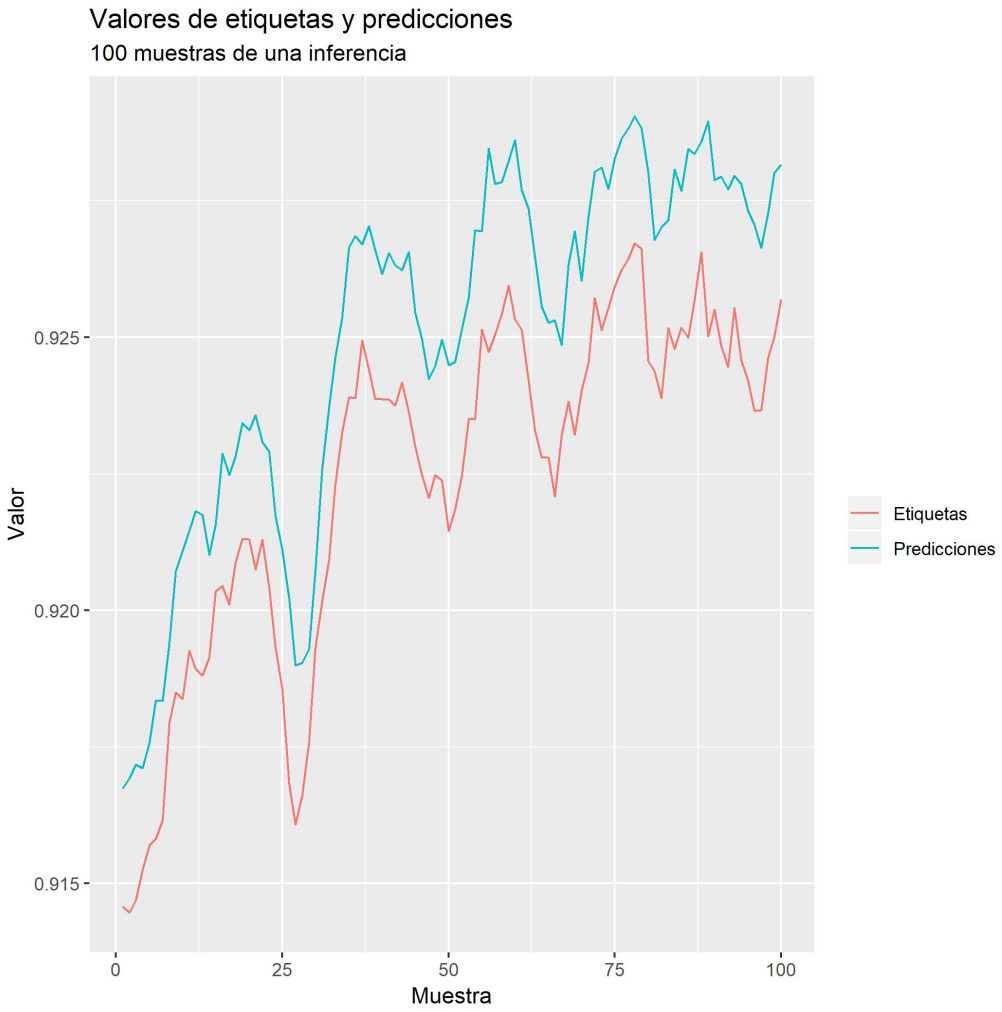


Figura 3.14: Calidad de la predicción de un modelo *fully connected*

3.4.2. Entrenamiento e inferencia de redes convolucionales con Keras

Como ya se vio en el apartado anterior, para el modelo Keras el número de parámetros entrenables es fijo, 4,352,641; por tanto, solo se modificará el tamaño de muestras con las que se realizan los procesos por lotes ya que, al igual que para las redes *fully connected*, se ha utilizado el optimizador ADAM. En la figura 3.15 se puede apreciar la topología del modelo utilizado con todas las operaciones implícitas del proceso de entrenamiento.

A continuación se muestra la función *fit()* de Keras que permite realizar un aprendizaje supervisado del objeto red neuronal generado anteriormente *nn_model* con la estructura definida para este trabajo:

```
history=nn_model.fit(x_train, y_train,batch_size=bs,epochs=epochs,
          verbose=1,validation_split = 0.2, callbacks = [tb])
```

A diferencia de TensorFlow no hay necesidad de crear una sesión ni un *feed dictionary* para inyectar los datos al modelo. Simplemente basta con indicar el conjunto de datos para entrenamiento, sus etiquetas, el porcentaje del tamaño conjunto de validación (*validation_split*), el número de *epochs* (que está fijado a 1) y el tamaño del lote o *batch_size*. Los tamaños que se han elegido para analizar el comportamiento del modelo han sido de 2, 8, 32, 64, 128, 256, 512 y 1024 muestras. Keras también permite incorporar funciones de

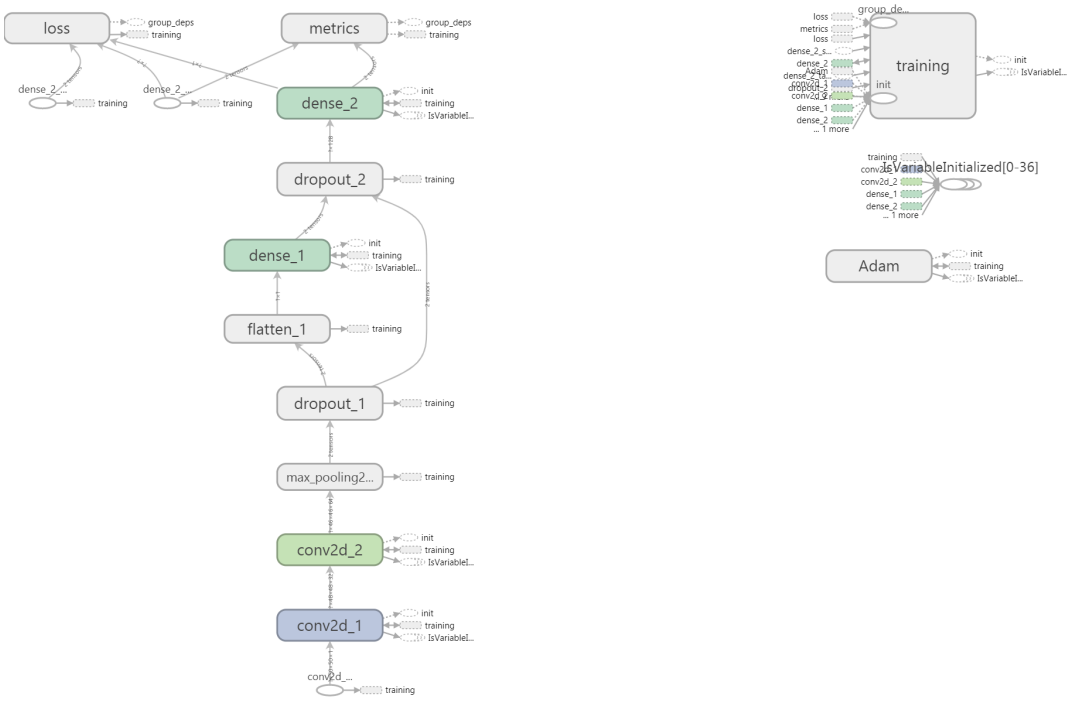


Figura 3.15: Grafo Tensorboard la red convolucional utilizada

callback para realizar diversas funcionalidades durante el entrenamiento como Tensorboard, la cual se detallará en la siguiente sección debido a su importancia para el perfilado.

Una vez se realiza el entrenamiento, se genera un diccionario Python de valores denominado historial *history* que contiene la información relevante del modelo. Una vez el modelo *nn_model* se ha entrenado se puede llamar a la función *save()* de Keras sobre él para almacenarlo íntegramente en un fichero *.h5* a diferencia de TensorFlow que almacenaba los datos en varios ficheros. Esto permite mayor facilidad al trabajar con modelos permitiendo un almacenado y carga de forma más cómoda para el proceso de inferencia.

Para realizar el proceso de inferencia se puede llamar a la función de Keras *evaluate()* sobre un nuevo conjunto de entrada y sus respectivas etiquetas para generar las métricas con las que el modelo fue compilado. También se puede llamar a la función *predict()* para obtener los valores de las predicciones para un conjunto de entrada. La primera forma de hacerlo está incorporada en *4keras_CNN_prof.py*; la segunda se ha implementado en *8inferencia_conv.py*, que mediante la función *load_model* de Keras, permite cargar el modelo y realizar la inferencia sobre él. En el repositorio se ha añadido el directorio */8h5* con un modelo entrenado en formato *.h5* sobre el que se puede hacer la inferencia directamente.

Como se puede comprobar en los códigos del trabajo, trabajar con Keras es más cómodo que con TensorFlow. Sin embargo hay alguna funcionalidad que no está integrada completamente y no permite realizarse sobre modelos Keras. A continuación se tratará de cómo se han monitorizado los procesos de entrenamiento e inferencia en las redes neuronales según el *framework* utilizado.

3.5. Entrenamiento distribuido con Keras y Tensorflow 2

Debido al gran tiempo que implica el proceso de entrenamiento de modelos, actualmente y cuando es posible, se recurre a la distribución del proceso entre varios dispositivos en pos de acelerar la tarea. Mediante la versión de Tensorflow 2.0 se pueden realizar entrenamientos de redes neuronales de forma distribuida gracias a un API (*Application Programming Interface*) que establece una estrategia de distribución preestablecida entre las plataformas locales mapeadas, *tf.distribute()*. En este trabajo, se ha recurrido a integrar este API en el programa que cuentan con la red neuronal convolucional Keras debido a que en los programas implementados con Tensorflow nativo el proceso de distribución del cómputo no es tan sencillo.

Actualmente, la versión 2.0 de Tensorflow cuenta con varias estrategias de distribución de cómputo, todas ellas de forma síncrona, lo que implica que el entrenamiento en cada uno de los nodos se realiza con **paralelización de los datos**. Por tanto, cada nodo contará con una réplica del modelo a entrenar. De este modo, el conjunto de datos de entrada se secciona en partes que se distribuyen por todos los nodos y estos irán actualizando las variables y añadiendo gradientes de forma conjunta. En contraposición encontramos la distribución asíncrona, en la que cada nodo realiza partes del entrenamiento de forma independiente; a día de hoy no existen referencias a este tipo de estrategias de distribución para la Api que se ha utilizado. Por tanto, se han realizado entrenamientos distribuidos mediante dos estrategias síncronas.

Mirrored strategy Replica el modelo en cada uno de los dispositivos que van a ser utilizados para el cómputo y establece unas variables de entorno a las que se les aplicarán actualizaciones de forma idéntica mediante algoritmos *all-reduce* hasta finalmente conformar la variable contextual *MirroredVariable*. Por defecto el algoritmo que se ejecuta es el Nvidia NCCL aunque se pueden aplicar otros. En la figura 3.16 se puede comprobar la operatividad realizada.

Esta estrategia permite indicar sobre qué dispositivos se desea hacer la distribución. En este trabajo se ha ejecutado en GPUs idénticas y GPUs diferentes y se ha calculado el tiempo que implica realizar el entrenamiento con diferentes tamaños de *batch*. Cabe destacar, que aquí el parámetro que se pasa a la red será el **tamaño del batch para cada una de las réplicas** y no el tamaño del *batch* total, que se calcula a partir del producto entre el de cada réplica y el número de réplicas en sincronización con la estrategia.

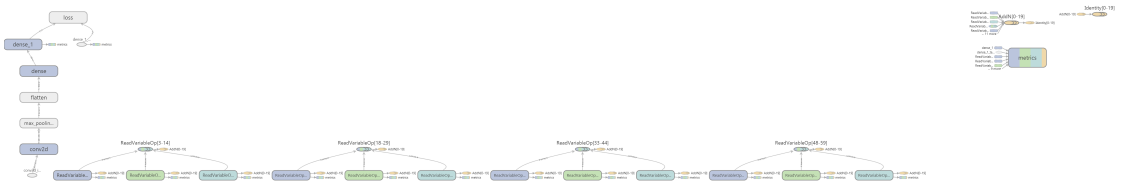


Figura 3.16: Grafo de estrategia replicada

Para cada tipo de topología de dispositivos para el entrenamiento distribuido, se establecerá un número de iteraciones en la *epoch* determinado por Tensorflow en función de las memorias de los dispositivos mapeados y el *batch* por réplica indicado. Las combinaciones de dispositivos GPUs utilizadas han sido las siguientes:

- GTX980 y GTX980.

- GTX1080 y GTX980.
- GTX1080 y dos GTX980.

A continuación, en la figura 3.17 se muestra una captura de Tensorboard para el entrenamiento del modelo en las tres GPUs del laboratorio.Se puede apreciar mediante el color cómo se distribuyen las operaciones sobre las GPUs.

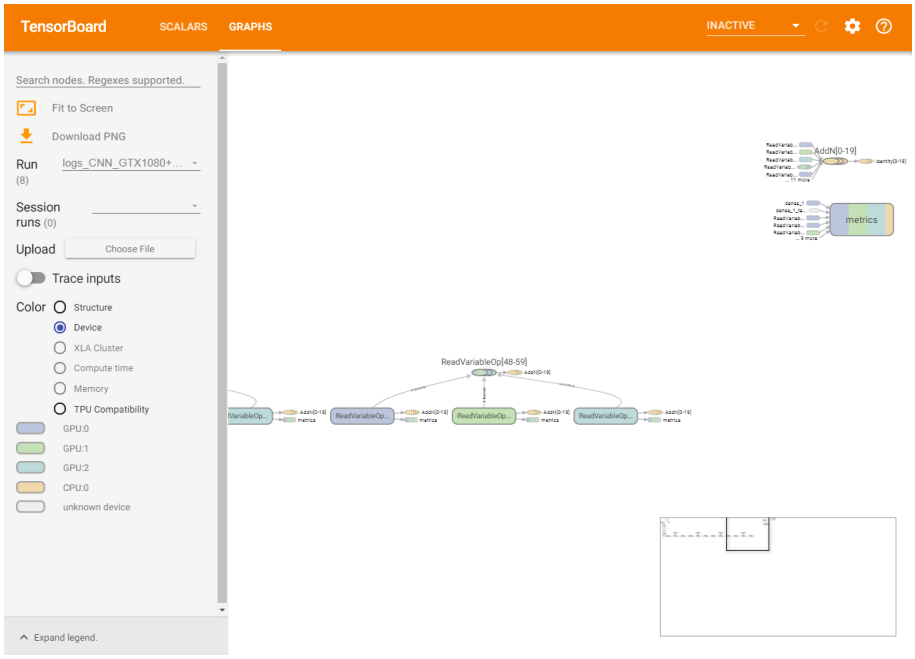


Figura 3.17: Operaciones para una actualización de la variable de entorno

El proceso de distribución de cómputo se realiza tras el descubrimiento por parte de Tensorflow de los dispositivos disponibles para el entrenamiento. Una vez terminado, se calculan el número de iteraciones que se realizarán durante cada una de las *epochs* en función del tamaño del *batch* que se envía a cada réplica. Tras esto, se procede al reajuste de los pesos de la red aplicando *backpropagation* en cada uno de los nodos y finalmente se recogen las variables de entorno generadas por cada uno de ellos. Será este parámetro, el tamaño de *batch* por réplica, lo que limite el entrenamiento; en definitiva, su máximo valor aplicable viene determinado por la memoria del dispositivo con menor memoria.

Central strategy A diferencia de la estrategia anterior, aquí las variables no están replicadas en cada nodo, sino que se almacenan en la CPU, que será la encargada de distribuir las operaciones entre las GPUs locales mientras alberga el modelo, véase la figura 3.18. En este trabajo se ha utilizado esta estrategia sobre todas las GPUs disponibles ya que no se permite realizar un mapeo sobre dispositivos a diferencia de la estrategia anterior.

En el directorio */distribuido* del repositorio del proyecto se pueden encontrar los códigos Python para su ejecución.

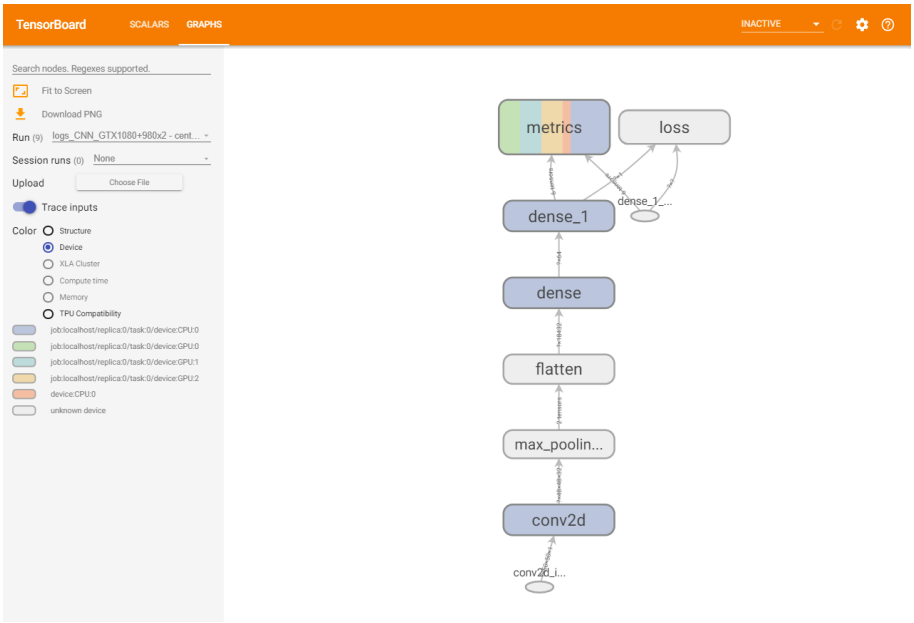


Figura 3.18: Grafo de estrategia centralizada

3.6. Perfilado de redes neuronales

En esta sección se detallan las herramientas utilizadas en la monitorización de los procesos de entrenamiento en inferencia de las redes neuronales. El objetivo de este perfilado es extraer métricas temporales que permitan un análisis exploratorio sobre los requerimientos computacionales que exigen los modelos en cualquier dispositivo hardware. De este modo se habilitará un conjunto de datos que se podrá analizar y evaluar de forma conveniente para poder decidir en qué procesos sería conveniente una aceleración mediante GPU.

Todas estas herramientas son funcionalidades de Python, Tensorflow y Keras, cada una con sus peculiaridades. A continuación se listarán las tareas a cubrir y cómo se han incorporado en los distintos programas. Para que todos estas tareas se puedan realizar de forma transparente al dispositivo sobre el que se realizan, es necesario que cuando se trabaje sobre una GPU Nvidia se habilite la librería de perfilado CUPTI en el entorno para poder perfilar las operaciones realizadas por este dispositivo según se indica en el Apéndice C.

3.6.1. Extracción de tiempo del proceso

Para acometer esta tarea se ha recurrido a la librería *time* de Python que permite iniciar un contador cuando se llama a su función *time()*. Con esto se puede realizar la diferencia entre el marcador de tiempo inicial y el final para calcular el tiempo total del proceso. Se ha utilizado de la misma forma en los códigos de Tensorflow y Keras para extraer los tiempos de entrenamiento e inferencia. Para las redes neuronales en Tensorflow también se ha incorporado para monitorizar el tiempo de los pasos en el entrenamiento para cada lote. Esto es posible gracias a que, como se ha visto en la sección 3.4.1, se realiza una ejecución de la sesión para cada uno de estos lotes. Sin embargo para Keras, que realiza todo el entrenamiento bajo la función *fit()* no es posible extraer el tiempo de cada procesado del lote. Tras este proceso de extracción, se generan unas tablas con los tiempos de operación sobre cada dispositivo de forma que se pueda realizar la evaluación pertinente. Sin embar-

go, para poder fundamentar las comparativas temporales con argumentos empíricos será también necesario realizar las tareas que se muestran a continuación.

3.6.2. Extracción de grafos Tensorboard

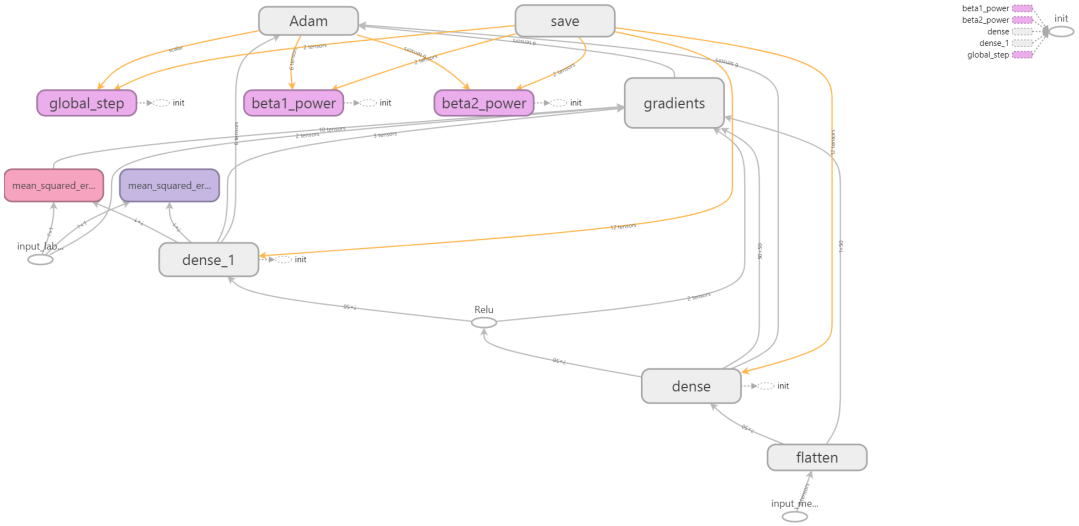


Figura 3.19: Grafo Tensorboard de un modelo *fully connected*

Esta tarea se realiza de forma totalmente diferente con Keras y con Tensorflow. Aunque la finalidad sea la misma, que es generar unos ficheros interpretables por Tensorboard (sección 2.4.5) para visualizarse en el navegador, la manera de generarlos no es trivial. Gracias a la evolución de Keras y su cada vez más completa integración con TensorFlow, se pueden generar estos ficheros *logs* mediante la llamada a la función *callback keras.callbacks.Tensorboard()* durante el proceso de entrenamiento de la función *fit()*. Esto permite generar de forma paralela ficheros para construir el grafo Tensorboard de manera predefinida, lo cual solo permite generar las métricas y la topología de la red neuronal sin monitorizar los tiempos de cómputo ni el uso de memoria de todas ellas como sí permite Tensorflow. En versiones posteriores de Keras a la utilizada en este trabajo se ha incorporado el argumento *profile_batch* para monitorizar el uso de recursos por cada batch.

Para la generación completa de un grafo con su monitorización del uso de recursos por cada proceso se ha utilizado Tensorflow nativo. Esto permitirá conocer qué operativas son las que implican mayor gasto computacional en el dispositivo. Para extraer estos valores de los procesos de entrenamiento e inferencia se debe generar un objeto *tf.summary.FileWriter()* que apunte a un directorio local donde almacenar el grafo, las métricas y los metadatos con la función *add()* sobre este objeto. De este modo de puede incorporar un marcador en cualquier punto de la ejecución para visualizarlo en Tensorboard. En el código *3fullyconnected_nets_profiler.py* se han generado *logs* para cada paso de entrenamiento y para la inferencia.

En la figura 3.19 se muestra una estructura completa de operativas a realizar en el entrenamiento de una red neuronal de una capa oculta de 50 neuronas. Gracias la monitorización de cada una de las ejecuciones de la sesión en la figura 3.20 se puede apreciar cómo para la inferencia no se realizan todas estas operativas.

Gracias a esta monitorización se puede apreciar cual es el dispositivo que se ha realizado

para ejecutar un proceso. En la figura 3.20 se muestra un proceso de inferencia realizado sobre GPU, que se representa mediante color azul. Los nodos que aparecen sin color, son aquellos que no se ejecutan para el proceso de inferencia.

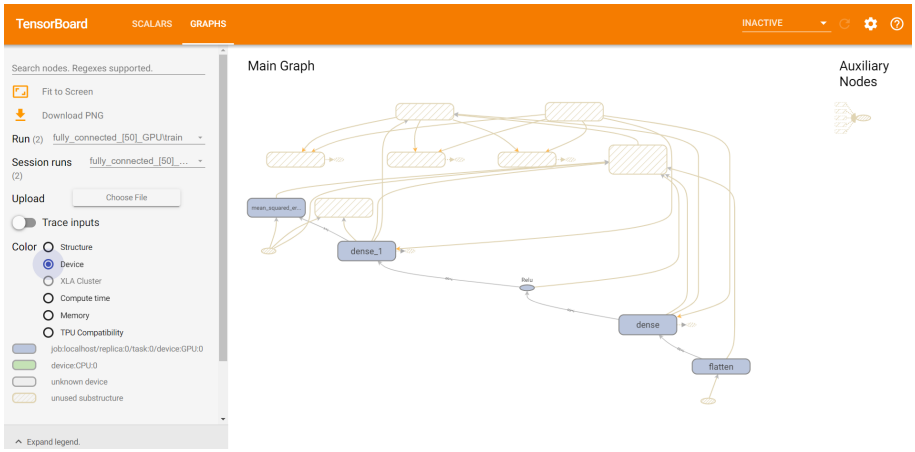


Figura 3.20: Menú de Tensorboard para la sesión de inferencia en GPU

Del mismo modo, mediante el menú interactivo de Tensorboard se pueden mostrar las exigencias de cómputo y memoria de cada operación como se muestra en las capturas de la figura 3.21.

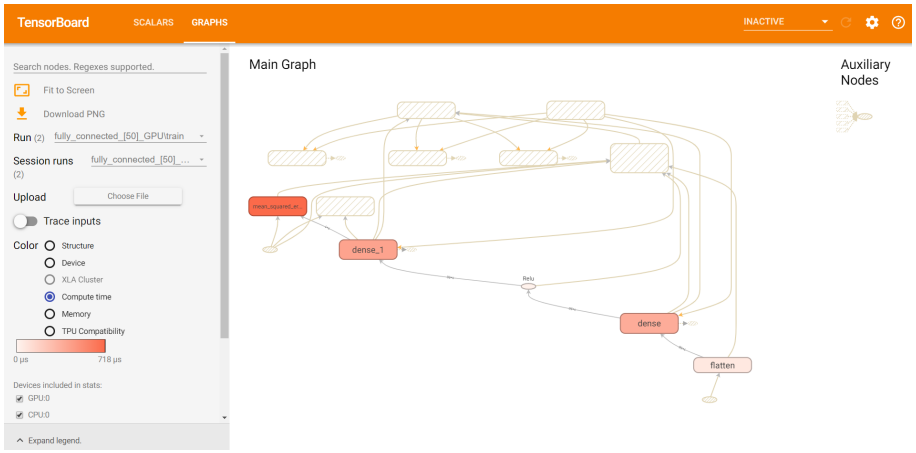
Gracias a Tensorboard se han podido dilucidar los comportamientos de los procesos de forma intuitiva. Ha sido una herramienta indispensable para la generación correcta de los modelos y para asegurar que los procesos se ejecuten de manera conveniente sobre los nodos específicos del grafo.

En el repositorio del proyecto se ha incluido una carpeta con ficheros *log* para visualizar algunos grafos que representan modelos de redes neuronales diferentes mediante los comandos propios del Apéndice C.

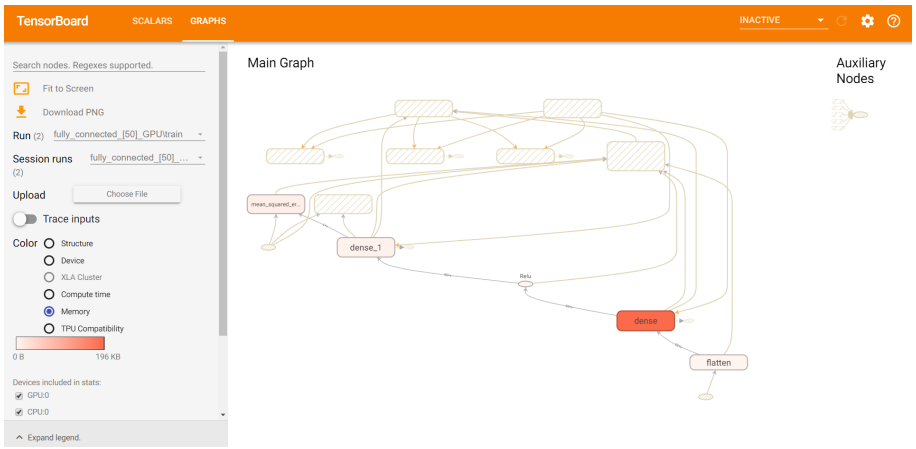
3.6.3. Extracción de trazas de ejecución

Para poder escudriñar las operaciones que se realizan en los distintos dispositivos y el tiempo que cada una de estas implica se han generado trazas de ejecución que se pueden visualizar mediante el navegador Chrome. Como se ha mencionado antes, Keras en su versión utilizada para este trabajo no permite monitorizar las operaciones internas, por tanto, esta tarea solo se ha realizado en Tensorflow nativo para las redes *fully connected* gracias a la función *timeline()* del cliente Python de Tensorflow. De forma similar a los ficheros *logs* de Tensorboard se generan unos ficheros de trazas *.json* que se podrán cargar en el navegador según se indica en el Apéndice C. Con esto se puede visualizar cómo se distribuyen las operaciones del grafo por los cores de los dispositivos según se muestra en la figura 3.22. Se puede ver cómo los *pid* indican los procesadores disponibles, lo cual variará entre CPUs y GPUs.

En la figura 3.22 se puede apreciar cómo la operación que implica mayor tiempo es MatMul con 574,719 milisegundos. Esto se corresponde con las operaciones a realizar tras el paso de los datos de inferencia a través de la red en la que existen numerosas neuronas y calcular el valor de salida. Gracias a esta herramienta, se puede realizar una comparativa ágil de cómo se realizan estas operaciones y el tiempo que implican en cada dispositivo. En la figura 3.23 se puede apreciar como, para el mismo modelo, los *pid* son mayores debido a la ejecución en paralelo de una tarjeta gráfica y el tiempo de operación es considerablemente



(a) Menú de Tensorboard



(b) Menú de Tensorboard

Figura 3.21: Menú de Tensorboard con las exigencias de cómputo de cada operación

menor.

En contraste con las trazas para el modelo anterior en distintos dispositivos, en la figura 3.24 se puede apreciar cómo para una red de menor tamaño, con solo 50 neuronas de capa oculta, el tiempo de las Matmul es considerablemente más pequeño que para la red anterior para el mismo lote y sobre el mismo dispositivo.

En el directorio /traces del repositorio del proyecto, véase Apéndice C, se encuentran algunos ejemplos de trazas extraídas de los entrenamientos e inferencias sobre distintos dispositivos. Como se puede comprobar, con estas trazas se pueden dilucidar las diferencias de cómputo y tiempo entre los dispositivos y los modelos. La herramienta de trazas será necesaria para poder fundamentar y contrastar los resultados del proceso de evaluación así como una mejor comprensión de las operativas implementadas.

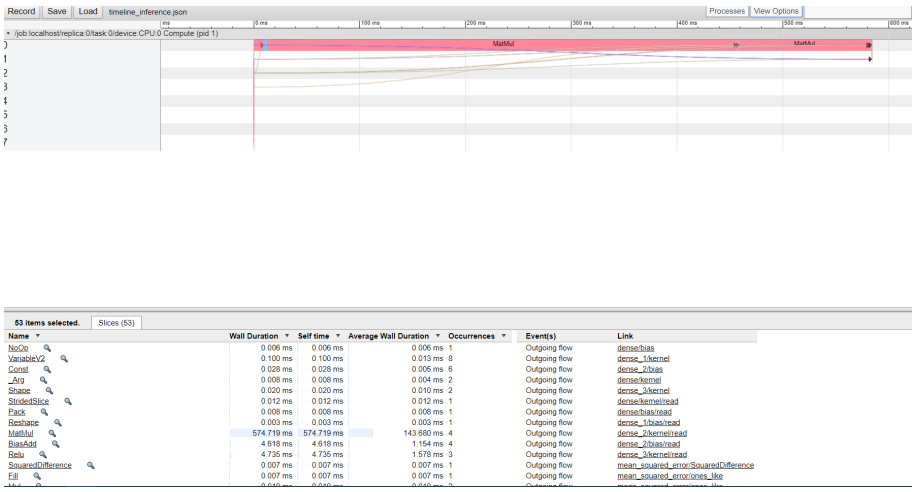


Figura 3.22: Trazas de un proceso de inferencia de una red con tres capas ocultas [6400, 3200, 1664] sobre un Intel Core i7.

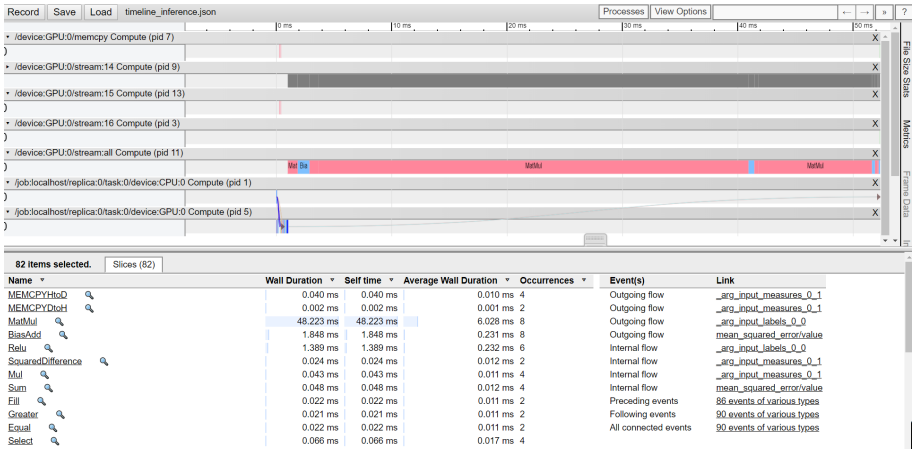


Figura 3.23: Trazas de un proceso de inferencia de una red con tres capas ocultas [6400, 3200, 1664] sobre una GTX950M.

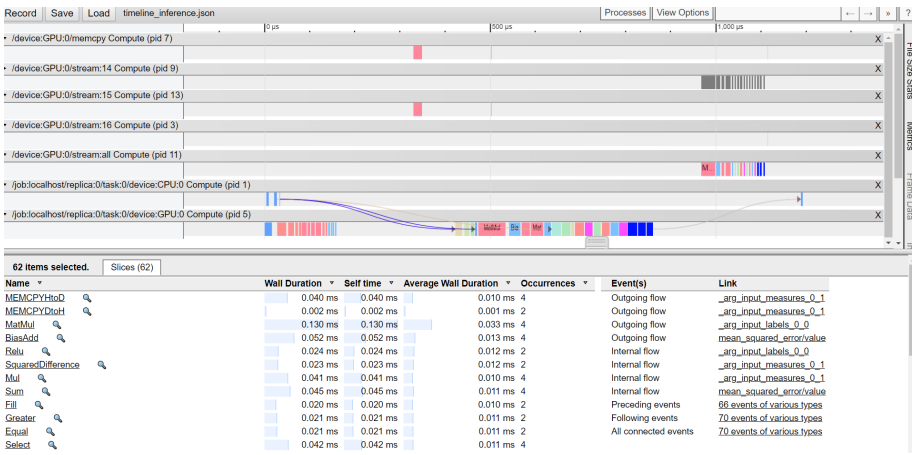


Figura 3.24: Trazas de un proceso de inferencia de una red con una capa oculta de 50 neuronas sobre una GTX950M. .

Capítulo 4

Evaluación temporal del entrenamiento y la inferencia

Gracias al desarrollo de un conjunto de modelos y a la extracción de métricas de la monitorización de los procesos de entrenamiento e inferencia, se ha conseguido contar con un conjunto de datos sobre el que se puede hacer una comparativa y averiguar si es necesaria una aceleración de los procesos mediante una GPU. Como se ha indicado en la figura 3.1, para realizar una evaluación consecuente es esencial la realización de un análisis exploratorio continuo para modificar parámetros de los procesos y así poder sacar conclusiones demostrables.

Este análisis exploratorio de los valores obtenidos tras la ejecución de los modelos sobre distintas plataformas hardware se ha realizado con el lenguaje de programación R, los códigos se encuentran en el directorio /analisisR del repositorio del proyecto. Como los datos generados se estructuran en formato de tabla, se han podido unificar de forma sencilla y visualizar para favorecer su comprensión. Todas las conclusiones generadas en este capítulo irán acompañadas de figuras o tablas comparativas obtenidas exclusivamente de los datos generados en el capítulo anterior. Estas tablas cuentan con las mediciones temporales de los procesos ejecutados en cada modelo y para cada dispositivo hardware.

Antes de empezar el proceso de evaluación temporal para este trabajo se parte de la premisa de que, generalmente, el uso de una aceleración por GPU reducirá el tiempo de cualquier operativa. Para poder corroborar esto es imprescindible conocer las características de los dispositivos utilizados y comprender su implicación en los procesos según se detalla en la sección 2.6.

En las tablas 4.1 y 4.2 se muestran las características relativas a la capacidad de procesamiento y de memoria de los dispositivos utilizados en el trabajo. Ya se ha especificado anteriormente que en el PC se encuentran el Corei7 y la GPU M (mobile) y en el servidor del laboratorio, el resto de componentes. Para ampliar esta información se puede acceder al Apéndice A.

CPU Intel	Arquitectura	Memoria [GB]	Cores	Frecuencia de reloj [Ghz]
Corei7	64 bit	16 (RAM)	8	2.30
XeonE5	64 bit	64 (RAM)	28	2.70

Tabla 4.1: Especificaciones de las CPUs utilizadas

Como se puede apreciar, el número de cores de cómputo es considerablemente mayor

GPU Nvidia	Arquitectura	Memoria	Memoria dedicada [GB]	CUDA Cores	Frecuencia de reloj [Ghz]	Compute capability
GTX950M	Maxwell	DDR3/GDDR5	2	640	>0.914	5.0
GTX980	Maxwell	GDDR5	4	2058	>1.064	5.2
GTX1080	Pascal	GDDR5X	8	2560	>1.607	6.1

Tabla 4.2: Especificaciones de las GPUs utilizadas

en las GPUs, lo cual se antoja más práctico para la ejecución de procesos en paralelo. En los siguientes apartados se podrá comprobar si esto influye en los procesos en los que se requieran gran número de operaciones, siendo este número de cores una limitación para la ejecución en paralelo de las mismas. Por otro lado, hay que destacar también las limitaciones de memoria: si bien la CPU cuenta con la memoria RAM del ordenador en el que se encuentra, la GPU cuenta con una memoria dedicada sobre la que se copian datos desde la memoria principal RAM. Por tanto, como se verá a continuación, serán estas limitaciones de espacio las que restrinjan el guardar los máximos resultados de las operaciones anteriormente computadas. Estos resultados serán los tensores generados por Tensorflow tras las transformaciones realizadas por las capas de neuronas según se muestra en la figura 3.12. Para el caso de transformaciones que generen tensores de gran tamaño, debido a las dimensiones de las capas ocultas, se podrá comprobar cómo Tensorflow parará su ejecución debido a las limitaciones de memoria.

A continuación se describirán las evaluaciones temporales realizadas tras el análisis exploratorio sobre las características de memoria y capacidad de cómputo de los dispositivos. Gracias a esto y a la comparación de los tiempos de entrenamiento e inferencia, se podrá averiguar cual es el dispositivo o dispositivos más convenientes para realizar la tarea.

4.1. La memoria del dispositivo y su influencia en el tiempo y la precisión

El entrenamiento y la inferencia en las redes neuronales se suele hacer mediante lotes de modo que no se inyecta todo el conjunto de datos de una sola vez. En este apartado se trata principalmente de la influencia del tamaño de este lote y su repercusión en la precisión del modelo y el tiempo de entrenamiento. El tamaño del lote está íntegramente relacionado con la cantidad de datos que puede albergar la memoria del dispositivo sobre el que se realiza el proceso, siendo la memoria RAM para las CPUs y la memoria dedicada para las GPUs. Es decir, en una GPU con una memoria dedicada de 2GB no se podrá procesar un tensor con dimensiones de datos mayor que esa capacidad.

Las dimensiones de estos tensores dependerán de la topología de las redes neuronales y, particularmente las del primer tensor generado, dependerán del tamaño del *batch* en su paso a la primera capa oculta. Es en este primer paso donde se va a hacer especial hincapié ya que según se ha podido comprobar en las redes *fully connected*, el tamaño de muestras introducidas a la red influye directamente sobre su precisión. Tras cada paso de este lote por la red se realiza un reajuste de los pesos a partir de las muestras, es decir, la red aprende de esas muestras calculando una nueva función de coste y un nuevo gradiente. En la figura 4.1 se muestran las operaciones realizadas en un paso de entrenamiento para la misma red neuronal para un *batch* de 100 y un *batch* de 1000 muestras sobre una CPU. Como se puede

comprobar, la implicación relativa (en tanto por ciento) del cálculo del gradiente (mostrado en color azul como *ApplyAdam*) en el tiempo total del paso es considerablemente mayor para el *batch* menor. De la misma manera, se puede comprobar cómo para un tamaño de lote de 100 muestras el tiempo total es menor como se puede ver en el eje de tiempo superior.

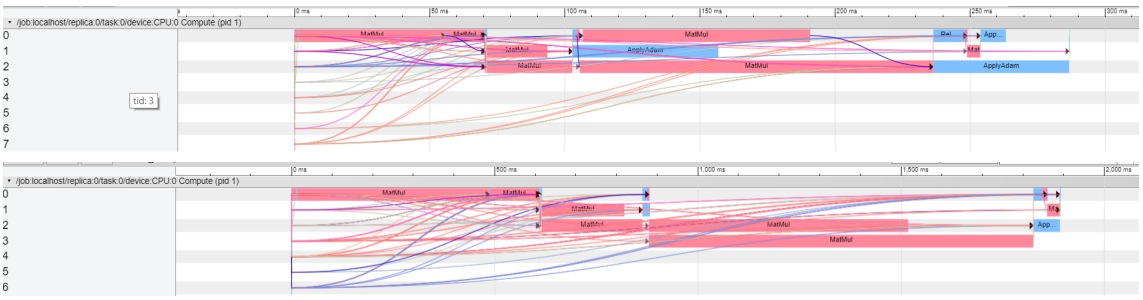


Figura 4.1: Operaciones en un paso de entrenamiento para 100 (arriba) y 1000 muestras (abajo)

Cuanto menor sea el número de muestras del lote, mayor número de iteraciones se deberán hacer hasta terminar de pasar todo el conjunto de datos por el modelo. En la figura 4.2 se puede apreciar la ínfima diferencia de precisión obtenida para los distintos conjuntos de *batch* seleccionados. En este resultado influye la regularización realizada sobre el conjunto de entrenamiento. De modo que si esta regularización (mediante aleatorización en este caso) es buena, aunque se introduzcan pocas muestras a la entrada, la red será capaz de generalizar correctamente y por tanto, ganar en precisión.

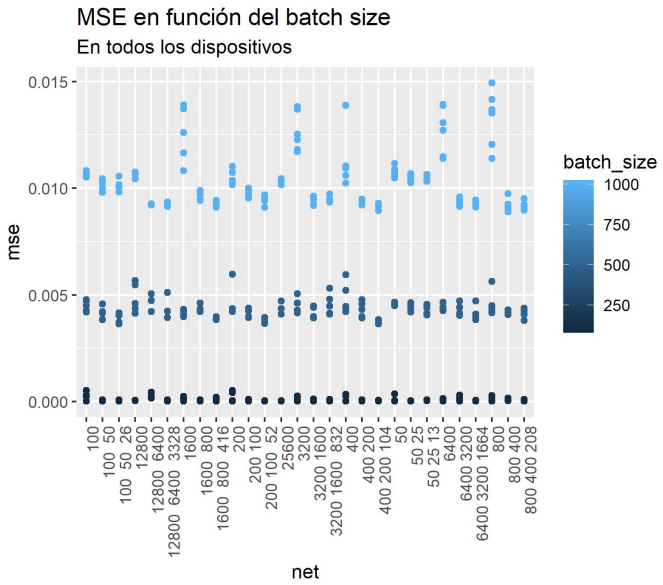


Figura 4.2: Influencia del *batch*_size en la precisión de los modelos

En lo que al tiempo respecta, realizar el entrenamiento por lotes es equiparable a realizar el entrenamiento sobre el conjunto de datos total repetidas veces o *epochs*; es decir, el tiempo de entrenamiento es directamente proporcional al número de iteraciones

que se ejecuten con cada lote y a las *epochs* que se ejecuten sobre el conjunto total de entrenamiento.

En las tablas 4.3 y 4.4 se pueden comparar las relaciones obtenidas entre el tiempo total y el número de *epochs*, y el tiempo total y el numero de pasos realizados.

device_name	epochs	t_t_o_t_a_l/ t_e_p_o_c_h
GTX1080	2	2.0027388416223335
GTX980	2	2.002600301725368
IntelXeonE5	2	2.001429023646041
GTX1080	1	1.0013275619020248
GTX980	1	1.0012223801833247
IntelXeonE5	1	1.0007315813839401

Tabla 4.3: Relación del tiempo total y el tiempo de *epoch*

device_name	t_t_o_t_a_l/(t_i_t*N_i_t)
IntelXeonE5	1.0001292492822769
GTX1080	1.0001237798028009
GTX980	1.0001223079239914
GTX950M	1.0000998327428676
IntelCorei7	1.0000915343455268

Tabla 4.4: Relación del tiempo total y el número de pasos por tiempo de *batch*

Por tanto, para un tamaño de *batch* pequeño, el número de pasos de entrenamiento será mayor y en consecuencia, el tiempo total de entrenamiento requerido será tantas veces mayor. Vista la mínima diferencia encontrada en la precisión y la gran diferencia encontrada en el tiempo de entrenamiento de los modelos en función del tamaño del *batch*, se ha decidido realizar la comparativa de los dispositivos con el tamaño máximo posible de *batch*. En la figura 4.3 se puede apreciar la evolución del tiempo de entrenamiento en función del tamaño del *batch* para una red de 50 neuronas, lo cual, muestra una diferencia considerable en lo que a tiempo se refiere. Como se puede apreciar, el tamaño pequeño del *batch* condiciona que no se aprovechen del todo las capacidades de procesamiento en paralelo de las tarjetas gráficas al introducir matrices de dimensiones más pequeñas.

Esta conclusión complementa la frase encontrada en el tutorial de *Tensorflow Distributed training with Keras* en el que se recomienda utilizar la máxima memoria del dispositivo para el tamaño del *batch*. Como se pudo comprobar en el análisis exploratorio, con la realización de un mayor número de iteraciones se deben realizar mayor número de transmisiones de datos entre memorias y un mayor número de *backpropagations*, lo cual implica un mayor tiempo de entrenamiento. Por tanto, se puede afirmar que es conveniente utilizar un lote del mayor tamaño posible para realizar el entrenamiento y en el caso de que se quiera ajustar la precisión, realizar un mayor número de *epochs* teniendo en cuenta que el mayor tamaño posible del lote lo determina la memoria del dispositivo.

Debido a que los perceptrones reciben en su primera capa oculta los datos aplanados (*flatten*), para calcular las dimensiones del primer tensor habrá que identificar el tamaño del *batch*, resultando sus dimensiones: D = (batch_size, neuronas_capa_uno). En el caso de que a lo largo de la red no exista una capa que supere en número de neuronas el *batch_ size* máximo seleccionado, se puede identificar de esta manera la capacidad máxima de la memoria para ese tipo de datos. Para esta tarea no se ha encontrado una operativa

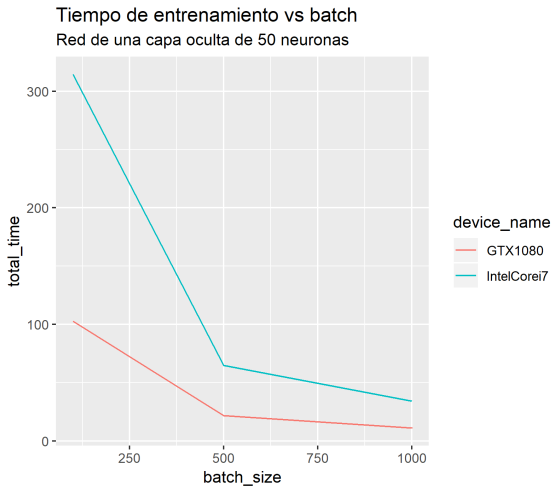


Figura 4.3: Tiempo de entrenamiento vs tamaño del lote

que permita determinar el tamaño máximo del *batch* de forma genérica. Es un proceso que se realiza probando hasta encontrar el valor óptimo en función de los datos. Por tanto, en este trabajo, se ha seleccionado un valor que permitiese adaptarse a las memorias de todos los dispositivos sin buscar el valor máximo. También hay que tener en cuenta, que en el proceso de entrenamiento, se deberán guardar en memoria los tensores para el proceso de *forward propagation* y para el de *backpropagation*, lo cual implica el doble de memoria necesaria.

En las redes convolucionales, la diferencia reside en que es la selección de *features* la que se hace a partir del número de imágenes que marca el *batch*, posteriormente, los datos se aplanan y se meten al perceptron para hacer la regresión. Por tanto, no se puede comparar con el número de muestras que se introducen en el perceptrón multicapa ya que al ser el mismo modelo, es el mismo número. Aquí solo varía la selección de las *features* a partir de más o menos muestras. Como se puede comprobar en la figura 4.4 el tiempo de entrenamiento de las redes convolucionales también disminuye considerablemente al aumentar el tamaño del lote hasta estabilizarse a partir de las 100 muestras.

Tras este análisis exploratorio de la implicación del tamaño del *batch* en la precisión y el tiempo, se ha concluido que, de forma genérica, será mejor utilizar el mayor tamaño de lote posible para aprovechar la cantidad máxima de memoria del dispositivo. De este modo, el número de operaciones a realizar en un paso será el máximo y se podrá aprovechar el paralelismo al máximo. Cuando se quiera ganar en precisión, será conveniente aplicar *epochs* que permitan reajustar los pesos con el máximo número de muestras y así generalizar mejor.

4.2. Evaluación del entrenamiento

Una vez se ha decidido el tamaño del lote a utilizar en pos de aprovechar la máxima memoria RAM y de GPU posibles y un ratio de aprendizaje constante, se puede establecer el marco comparativo bajo las mismas condiciones. También debido a la relación directa entre el número de iteraciones y *epochs* con el tiempo total, se van a realizar comparativas sobre un único recorrido del conjunto de entrada para un lote de 1000 muestras. Para realizar la evaluación temporal entre CPUs y GPUs se va a partir de la premisa de que a más

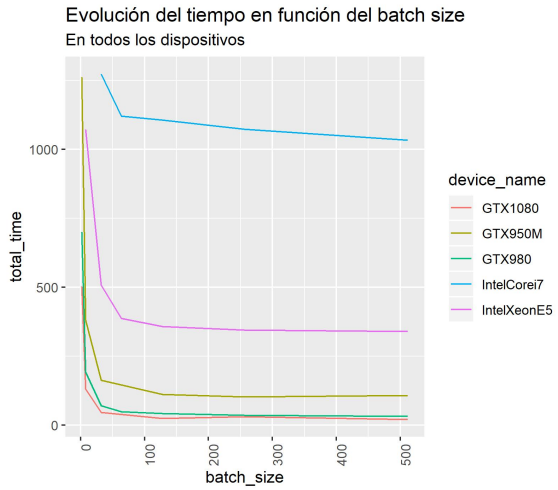


Figura 4.4: Tiempo de entrenamiento vs tamaño del lote

capacidad de computación en paralelo, mayor velocidad de operación. En primera instancia se va a realizar una comparativa entre las CPUs disponibles sin aceleración por GPU para todos los modelos de perceptrón. Acto seguido se comparará un entrenamiento realizado únicamente con CPU con otro acelerado por GPU para comprobar si, debido a la mayor cantidad de cores que pueden trabajar en paralelo, el tiempo es menor. Finalmente para las redes *fully connected*, se visualizará cómo influyen las dimensiones en el entrenamiento sobre los distintos dispositivos. En el último apartado se muestra una comparativa para el modelo de red convolucional.

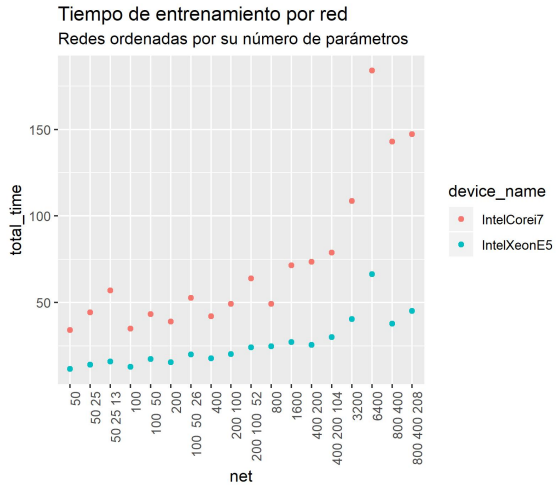


Figura 4.5: Comparativa entre CPUs multicore

Para las gráficas mostradas a continuación los perceptrones utilizados estarán ordenados en función de su número de parámetros (pesos) entrenables. De este modo, se podrá vislumbrar cómo este parámetro es el que influye en mayor medida en el tiempo de entrenamiento. Todos los tiempos están expresados en segundos.

4.2.1. Comparativa entre CPUs multicore

En la figura 4.5 se puede apreciar cómo el dispositivo con el mayor número de cores es más rápido que el de menos, en este caso, el XeonE5 cuenta con 28 cores y el Corei7 con 8. A medida que el número de parámetros aumenta, se ve que la diferencia temporal entre dispositivos aumenta de forma proporcional. También se puede apreciar que aunque una red tenga más parámetros, no tiene porqué ser más lenta de entrenar, esta circunstancia es debido a la topología de la propia red.

4.2.2. Comparativa entre CPU multicore y GPU

En la figura 4.6 se muestra el tiempo de entrenamiento en una CPU con 28 cores y entre la menor de las GPUs con 640 cores CUDA.

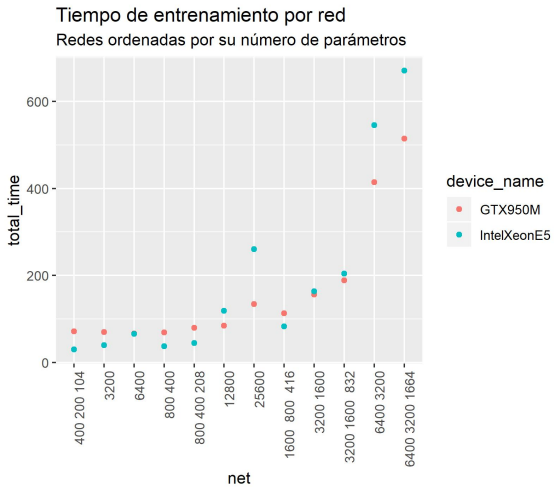


Figura 4.6: Comparativa entre XeonE5 y Nvidia GTX950M

Como se puede apreciar, para redes de pocos parámetros la CPU tarda menos que la GPU. Según se va ascendiendo este número, se aprecia que cuanto mayor número de cores para procesar operaciones en paralelo, mejor. De nuevo vuelven a aparecer circunstancias especiales debido a la magnitud de anchura de las redes. Se puede observar que al hacerse demasiado ancha la red (caso de 25000 neuronas) la GPU obtienen un mejor tiempo de entrenamiento. En contrapartida, en la siguiente red (1600 800 416), más profunda y con mayor número de parámetros, el tiempo de entrenamiento es menor en la CPU. Se puede, por tanto, afirmar que las CPUs presentan más complicaciones que las GPUs al entrenar redes de gran anchura.

En la figura 4.7 se puede apreciar la comparativa entre la CPU y la GPU más potentes disponibles. Se muestra el tiempo de entrenamiento en una CPU con 28 cores y entre la mayor de las GPUs con 2560 cores CUDA. Se ha separado en dos tramos para apreciar mejor las diferencias en el eje de ordenadas. La GTX1080 entrena más rápido que el XeonE5 desde las redes con menor número de parámetros. Para las redes más grandes, la diferencia de eficiencia es considerablemente mayor que en el caso anterior.

Esta comparativa entre los dispositivos más potentes denota la versatilidad que presentan las GPUs de última generación con arquitecturas como la Pascal, al ser capaces de superar en rendimiento a las CPUs más potentes para cualquier tipo de red neuronal.

En las gráficas anteriores se puede apreciar una evolución no lineal en los tiempos de

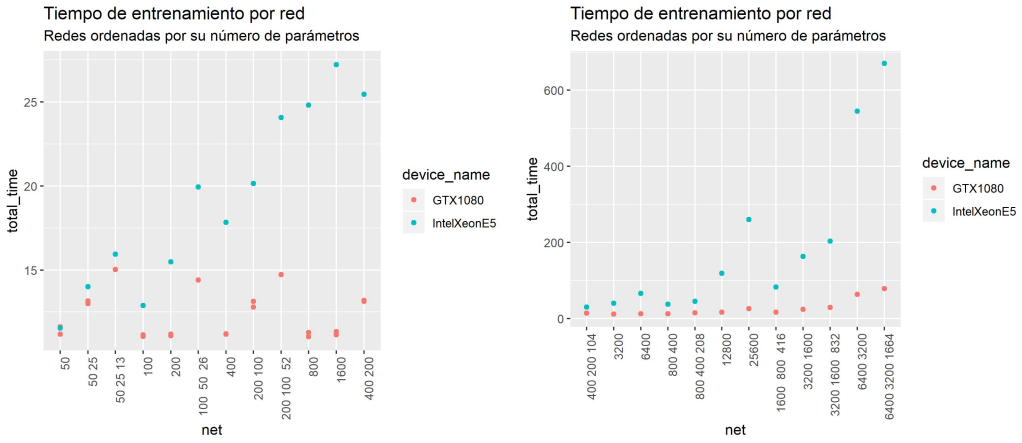


Figura 4.7: Comparativa 1 entre XeonE5 (der.) y GTX1080 (izq.)

entrenamiento en función del número de parámetros entrenables de las redes neuronales.

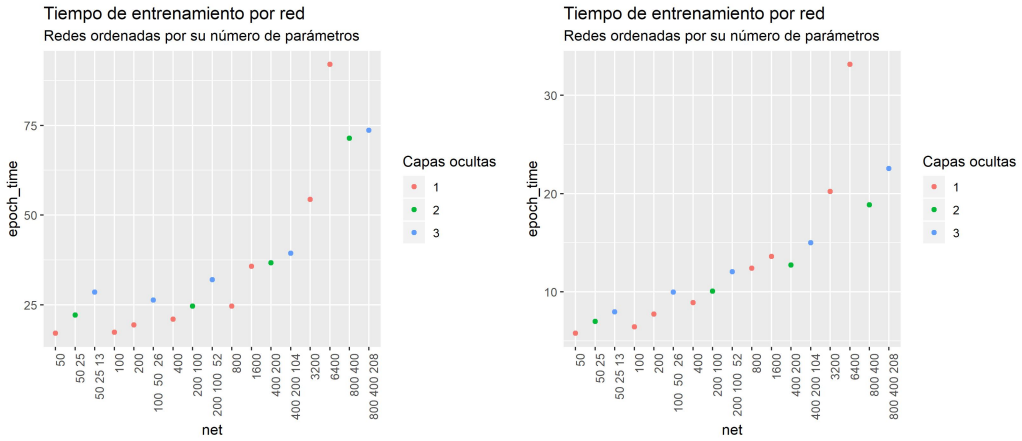


Figura 4.8: Topologías sobre Corei7 (izq.) y sobre XeonES(der.)

En esta sección se muestra el comportamiento por separado de los dispositivos en torno a esta circunstancia. De este modo, se podrá dilucidar la influencia de la profundidad y la anchura de las redes neuronales en el tiempo de entrenamiento. Gracias a esto, se podrá determinar un diseño de la topología de la red de forma que, en la medida de lo posible, no ralentice el proceso. Para realizar esta comparativa se han diferenciado las redes por capas ocultas de manera que se pueda apreciar cuando es más conveniente ensanchar la red o cuando hacerla más profunda en función del tiempo.

4.2.3. Influencia de la topología de la red en los dispositivos

En la parte izquierda de la figura 4.8, para el Corei7, se ve que a partir de una red de una sola capa con 3200 neuronas (166401 parámetros) tarda más en ser entrenada que una red más profunda de 800 400 y 208 neuronas y con mayor número de parámetros (444817). En la parte derecha de la figura 4.8, para el XeonE5, se aprecia un comportamiento similar a partir de la red ancha de 1600 neuronas (83201 parámetros), que ya es superada por una red más profunda de 400 y 200 neuronas con un mayor número de parámetros (100801).

Estas dos primeras figuras llevan a pensar que cuanto más potente es el dispositivo, más capaz es de procesar las redes profundas.

Para el caso de la GPU menos potente, la GTX950M, figura 4.9, se puede apreciar que procesa más rápido la red de 800 y 400 neuronas (361.601 parámetros) que una de 3200 neuronas (166.401 parámetros).

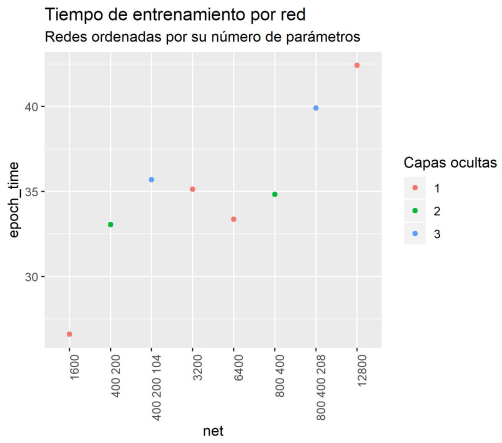


Figura 4.9: Topología sobre GTX950M

Para el caso de la GTX980, parte derecha de la figura 4.10, se puede apreciar que no presenta tanta dificultad al procesar redes con gran número de neuronas en una sola capa, redes anchas, y aún procesa más rápido la de 3200 neuronas que la de 800 y 400. Es a partir de la red de 6400 neuronas y 332.801 parámetros cuando empieza a tardar más que la red de 800 y 400. También la diferencia de tiempo se va reduciendo con la próxima red en número de parámetros. En este caso, la diferencia entre la siguiente red profunda de tres capas tarda menos de un segundo más en ser entrenada cuando en la 950M se aprecia una diferencia mayor a los 5 segundos.

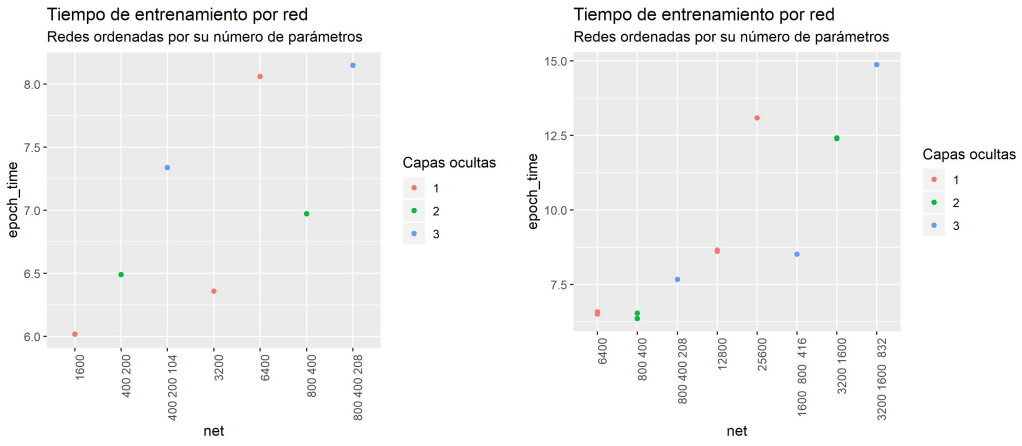


Figura 4.10: Topologías en GTX980 y GTW1080

Para el caso de la GTX1080, parte izquierda de la figura 4.10 se puede apreciar que es capaz de procesar redes anchas de forma más rápida que redes profundas hasta una red de 6400 neuronas. Sin embargo, a partir de esta cifra, ya procesa más rápido redes de hasta incluso tres capas ocultas con mayor número de parámetros. Por ejemplo, se aprecia

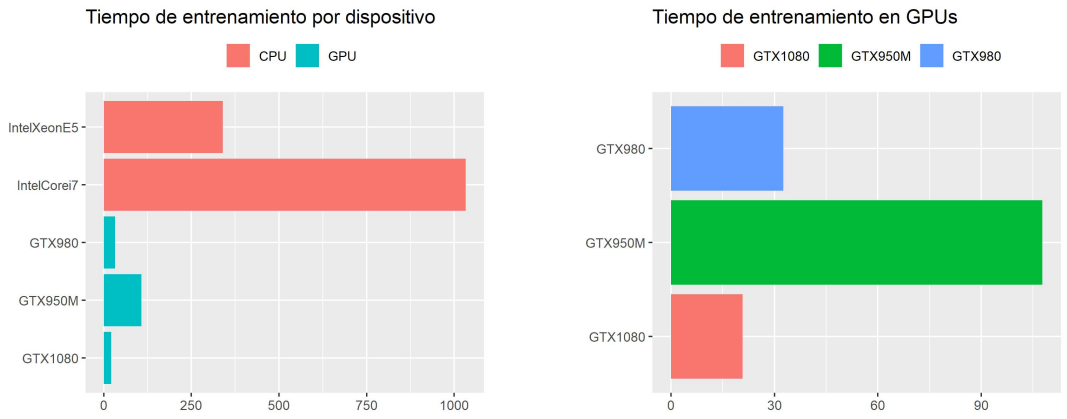
que procesa una red profunda de 1600, 800 y 416 neuronas con 1.696.033 parámetros más rápido que una red de 12.800 neuronas con 665.601 parámetros.

A la conclusión que llevan estos análisis exploratorios es que no es solo el número de parámetros entrenables lo que condiciona el tiempo de entrenamiento. También influye la topología de la red, es decir, su anchura y profundidad generan comportamientos diversos en función de las características del dispositivo. Como se ha podido comprobar, hasta que no se alcanza una capa con un número mayor al millar de neuronas, el comportamiento temporal sigue en función de número de parámetros entrenables sin que influya el número de capas ocultas.

Según se ha comprobado, las CPUs presentan mayor problema al computar redes anchas, con capas de un gran número de variables, que las GPUs, siendo esta característica más restrictiva que la profundidad de la red. Es decir, para modelos con un gran número de variables de entrada, las CPUs tendrán mayores restricciones temporales que las GPUs, sean cuales sean sus capas ocultas.

4.2.4. Comparativa para un modelo de red convolucional

Para esta comparativa se utiliza un modelo de red neuronal convolucional de 4,352,641 parámetros al que se le entrama con imágenes de de 50x50 en lote de 512 muestras.



(a) CPUs multicore y GPUs

(b) GPUs

Figura 4.11: Comparativa CPUs multicore y GPUs

Según se puede apreciar en la gráfica 4.11a, el tiempo que tarda en procesar la red convolucional la CPU más potente se sitúa en torno a 3 veces el tiempo que tarda la GPU menos potente. Para este tipo de redes neuronales, en las que su naturaleza implica un gran número de parámetros a entrenar, las GPUs son de gran utilidad ya que reducen el tiempo hasta 10 veces para estos modelos de dispositivos, por tanto, se prestan necesarias a la hora de acometer estos entrenamientos. De la misma manera se puede realizar una comparativa entre las GPUs como se aprecia en la gráfica 4.11b. Para este caso, la GTX1080 es casi tres veces más rápida que la GTX950M.

De nuevo, si se compara, para la GTX 1080, el tiempo de entrenamiento de una red perceptrón de un número de parámetros similar a esta red convolucional como una red con tres capas de 3200, 1600 y 832 neuronas respectivamente y 6.617.665 parámetros, se puede apreciar que el número de parámetros es una buena forma de intuir el tiempo de entrenamiento. Mientras que esta red convolucional de 4,352,641 parámetros tarda 21.93617

segundos en ser entrenada para un *batch* de 1024 muestras de imágenes, el perceptrón tarda 27.184643 segundos para un *batch* de 1000 observaciones.

4.2.5. Comparativa del entrenamiento distribuido

Tras definir las estrategias de distribución síncrona y establecer los dispositivos disponibles, aquí se muestra la comparativa del tiempo de entrenamiento para el modelo convolucional sobre diferentes plataformas. Como se puede observarse en la figura 4.12, aplicar un entrenamiento distribuido con cualquier estrategia aporta mayor velocidad que procesar en un único dispositivo.

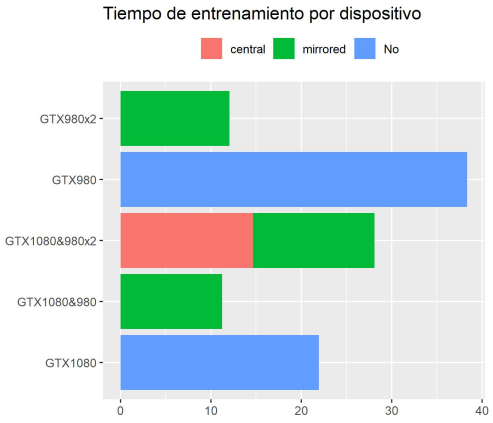


Figura 4.12: Comparativa según la estrategia de distribución

Se puede apreciar también que la estrategia de distribución *mirrored* es más rápida en todas sus combinaciones que la estrategia *centralized*. La diferencia se puede ver en la figura 4.13 con mayor detalle en la que se comparan las estrategias con una plataforma con los mismos dispositivos.

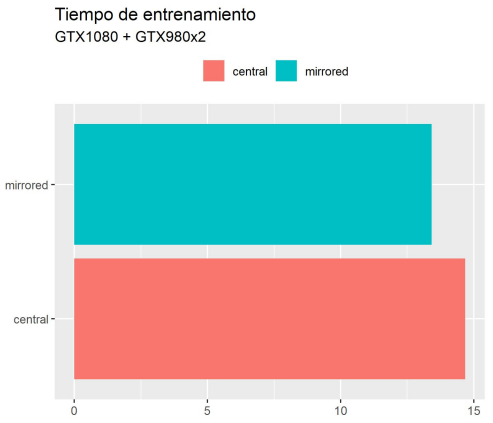


Figura 4.13: Comparativa de estrategias

Sin embargo contar con un mayor número de dispositivos a utilizar no tiene implicación directa en el tiempo de entrenamiento para la estrategia *mirrored*, al menos para este caso de uso. Para la estrategia central no se pudo comprobar debido a que no se pueden

seleccionar ni ocultar los dispositivos para realizarla debido a que se encuentra en versión beta a fecha de realización de este trabajo.

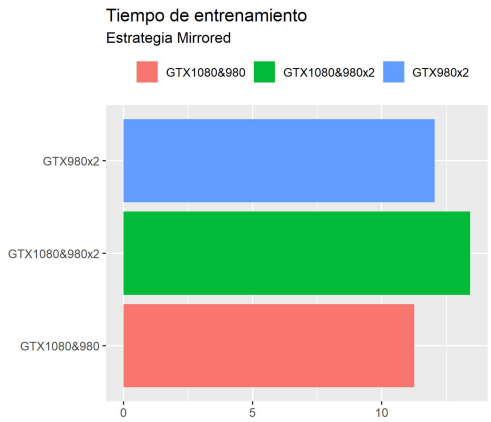


Figura 4.14: Comparativa para la estrategia replica

La estrategia *mirrored* replica el modelo en todos los dispositivos, según se puede observar para este caso en la figura 4.14, resulta computacionalmente más costoso establecer el modelo en tres GPUs que en dos. Sin embargo, que las GPUs tengan distintas arquitecturas, capacidades de cómputo o memoria, no resulta contraproducente.

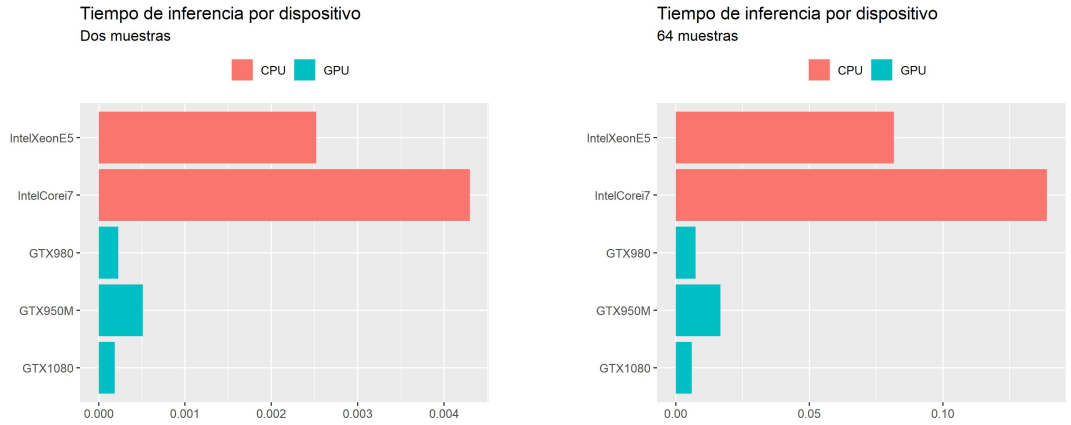
4.3. Evaluación de la inferencia

El proceso de inferencia consiste en realizar una predicción del valor de un sensor de radiación a partir de una observación tomada de los sensores colindantes en un instante de tiempo anterior con un modelo ya entrenado. A diferencia del entrenamiento, que busca ganar el precisión para la predicción, aquí el horizonte de predicción puede variar en función de las necesidades requeridas. Por tanto, conociendo las medidas de un grupo de sensores se podrá predecir cual será la radiación que habrá en otro dentro de x segundos. En el caso de la inferencia el aspecto temporal resulta más crítico ya que lleva una acción asociada tras la obtención del resultado. Es decir, si se desease, por ejemplo, orientar los espejos de una planta termosolar en base al resultado obtenido por el modelo, si este resultado tarda demasiado en llegar, implicará un menor tiempo para ejecutar la acción. Por tanto, aquí la necesidad es la de averiguar cuánto tarda el modelo en darme el resultado en función el dispositivo. Para el caso analizado, se ha entrenado a los modelos para que obtengan la mejor predicción en 10 segundos a futuro, sin embargo, esto no implica que no se pueda inferir para predecir en condiciones cercanas al tiempo real siempre que los dispositivos lo permitan.

Para la evaluación se parte de la premisa de que cuanto mayor número de parámetros o pesos tenga la red, mayor será el tiempo necesario para calcular la predicción. En esta parte se va a utilizar el modelo de red neuronal convolucional, ya que el objetivo buscado es poder decidir cuándo se requiere una aceleración por GPU y no analizar el comportamiento de los modelos, que para este caso, se presupone similar al del proceso de entrenamiento pero sin la *backpropagation*. Para empezar se partirá con conjuntos de pocas muestras o mapas de irradiación para obtener la predicción en un margen de tiempo menor al segundo, de este modo se podrá comparar a partir de cuantas muestras se tarda más de un segundo en

realizar la predicción. Para un mayor número de muestras de entrada, se incluirán también los resultados de algunos modelos de perceptrón.

En la inferencia se valora cuánto tarda una muestra o conjunto de muestras en atravesar la red neuronal una única vez. Como el objetivo de este trabajo es evaluar el comportamiento temporal de los dispositivos, se van a mostrar situaciones en las que se valore cuanto tardaría en predecir el modelo para distintos instantes de tiempo futuros, sin embargo, en el caso de que se quisieran introducir otras variables, el tiempo de paso por la red sería el mismo. Por ejemplo, en el caso de que se desee realizar la predicción del valor de radiación para dos instantes de tiempo, segundo i+1 y segundo i+2 para un único sensor será necesario introducir 2 muestras con los valores del resto de los sensores en los instantes i-1 e i-2, siendo i el instante actual.



(a) 2 muestras

(b) 64 muestras

Figura 4.15: Inferencias para 2 y 64 muestras

Como se puede apreciar en la gráfica 4.15a, el tiempo que tarda una en realizarse una predicción en cualquiera de los dispositivos es considerablemente menor, del orden de milésimas de segundo, al segundo que se quiere predecir. De modo que para un sensor se obtendrían los valores de radiación del segundo i+1 y del i+2 con un margen de tiempo considerable. Es en este punto donde, para elegir un dispositivo u otro, intervienen las necesidades de acción del sistema encargado de ejecutar la acción.

De la misma manera, si se desease conocer la radiación en dos sensores A y B para el mismo instante de tiempo i +1, sería necesario introducir 2 muestras, una con todos los valores menos los del sensor A y otra con todos los valores menos los del sensor B en el instante i-1. En el supuesto caso de que se quisiese conocer el valor de todos los sensores para el próximo instante de tiempo sería necesario introducir 50 muestras diferentes. En la gráfica 4.15b se muestra el tiempo de inferencia para un conjunto de 64 imágenes introducidas al modelo convolucional.

Es inmediato comprobar, aquí el margen que dejan las CPUs es de un orden de décimas de segundo para una posible actuación en tiempo real, considerablemente menor a cuando se introducían dos muestras. De nuevo, será el actuador del sistema el que requiera de un dispositivo u otro para obtener el resultado. Hasta este punto, se puede afirmar que las CPUs permiten realizar predicciones con un margen de tiempo aceptable.

A continuación se van a introducir muestras en grandes cantidades para evaluar supuestas situaciones de tiempo real en las que se requiriese la introducción de un gran número de valores de entrada, por ejemplo una central termosolar con un elevado número de sensores

requeriría tantos mapas de calor diferentes para adivinar cada uno de los valores de los sensores.

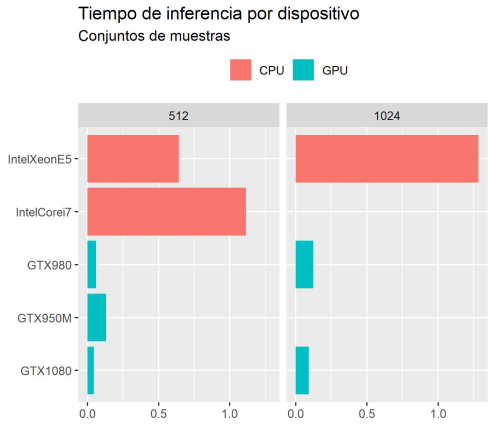
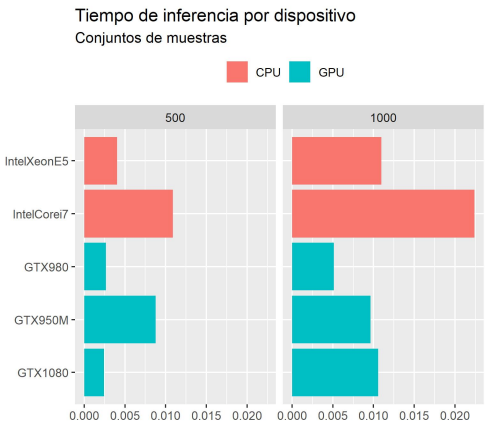


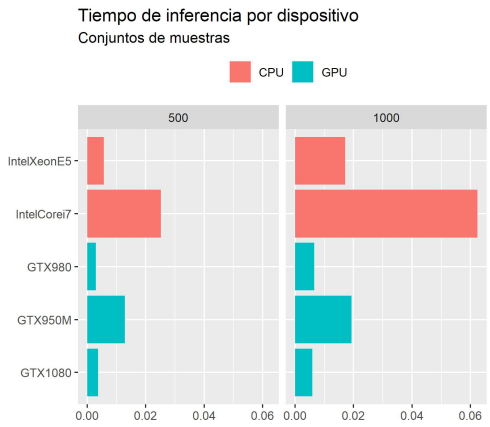
Figura 4.16: Inferencia para 512 y 1024 muestras

Con este número tan alto de variables, 1024, se puede comprobar en la parte derecha de la figura 4.16 cómo ambas CPUs no permiten situaciones de inferencia para tiempos menores de un segundo. Para el caso de 1024 muestras, no se puede realizar la inferencia en la GTX950M debido a limitaciones de memoria. Así pues, con esta comparativa se puede comprobar que la naturaleza de la red convolucional y el tipo de datos con los que trata, implican que a partir de un alto número de muestras para realizar la inferencia, sea conveniente aplicar la aceleración por GPU.

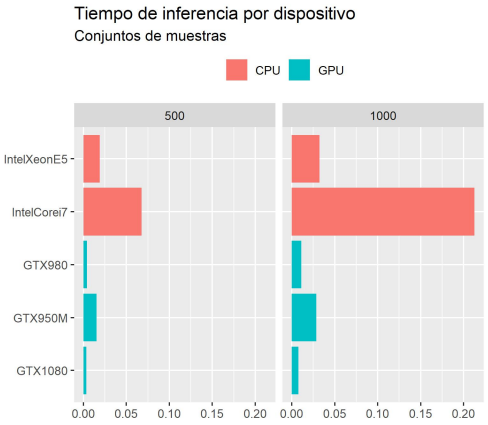
Para los modelos de perceptrón, no se ha encontrado ninguna situación en la que para cumplir con inferencias menores a un segundo sea necesaria la aplicación de una aceleración por GPU. Todos los modelos, particularmente los de un mayor número de parámetros, cumplen con creces para cualquier número de muestras en las CPUs utilizadas. En la figura 4.17 se presentan varias comparativas que lo demuestran.



(a) 12800 neuronas



(b) 3200, 1600 y 832 neuronas



(c) 6400, 3200 y 1664 neuronas

Figura 4.17: Inferencias para perceptrones

Capítulo 5

Conclusiones y Trabajo Futuro

La primera conclusión que se ha obtenido tras la realización del proyecto es que es primordial conocer las herramientas disponibles y las funcionalidades que aporta cada una de ellas para realizar un trabajo sobre aprendizaje automático. Con este proceso previo de selección se puede ahorrar un tiempo considerable. Con la plena integración de Keras en la versión 2.0 de Tensorflow muchas de las tareas que solo se podían realizar anteriormente en Tensorflow se podrán realizar de forma más sencilla sin necesidad de implementar de forma nativa. Como se ha podido ver, mediante unas pocas lineas de código este API ya permite una ejecución sencilla entre varias GPUs, por tanto sería conveniente el estudio de la nueva versión y la adaptación de todos los modelos para ganar en agilidad y funcionalidades.

En cuanto a lo que respecta a la evaluación temporal de los modelos, tras las comparativas realizadas en los distintos dispositivos, se puede afirmar que para los diferentes tipos de modelos de perceptrón, para un número pequeño de parámetros entrenables cercano al medio millón de parámetros, las CPUs multicore son una buena opción para el entrenamiento. Sin embargo, cuando el número de pesos es grande, las GPUs presentan mejores tiempos de entrenamiento; también para redes neuronales con un alto número de neuronas en una capa, generalmente condicionado por la capa de entrada, las GPUs presentan mayor velocidad. Del mismo modo, las comparativas sobre el modelo convolucional presentan gran ventaja para las tarjetas gráficas.

En cuanto a la inferencia, buscando condiciones cercanas al tiempo real, se puede decir que generalmente todos los dispositivos testeados presentan buenos comportamientos. Cuando el número de variables del conjunto de entrada aumenta, las GPUs presentan mejor desempeño. Sin embargo, para este caso de uso, se puede afirmar que se utilizar cualquier dispositivo sería válido.

Finalmente es conveniente destacar la versatilidad que presenta Tensorflow para ejecutarse en distintas plataformas. Con la ejecución del entrenamiento mediante múltiples GPUs se ha conseguido reducir el tiempo de entrenamiento considerablemente. De la misma manera, se puede aplicar sobre sistemas empotrados propicios para el proceso de inferencia, con propias librerías software para optimizar el rendimiento. Esto sería conveniente probarlo para situaciones en las que las condiciones de tiempo y consumo sean determinantes como en un contexto IoT donde se requiera la inferencia cercana al sensor.

Chapter 6

Introduction

Currently and increasingly, decision making is delegated to computer systems capable of generating intelligence and deciding on people. This is possible thanks to the existence of two fundamental factors: the data and its processing. Due to the advance of the Internet of Things, the generation of data has undergone a growing evolution, since we are now able to monitor virtually any action, however small it may be. To generate value or intelligence of this data, in recent years there have been advanced learning techniques automatic based on neural networks that are capable of processing immense amounts of information and generate conclusions from such data. Just as these techniques have advanced considerably, new hardware architectures have emerged that are capable of carrying out computing operations much more efficiently than conventional devices. This work deals with the processing of data from measurements taken by sensors of a solar thermal power plant that has a solar radiation acquisition system based on IoT. To address the task, the latest software and hardware tools will be used. Computing on this Big Data. Thus, the intention of this work will be to evaluate which of the hardware architectures it is convenient to use according to the developed software algorithms and their execution time on them to generate value.

6.1. Motivation

Due to the increasing complexity of machine learning models in the field of deep learning it is necessary to know the hardware resources that they need to run from efficient way. Computationally placing training and inference processes of the models is not a trivial matter, since locating these tasks conveniently allows forecasting the provisioning needs of computer systems. So, in this work we use two latest technologies, software and hardware, which will complement each other in the best possible way to temporarily evaluate the resolution of a specific problem and allow to select which hardware architecture is conducive to undertake computing.

6.2. Purposes

1. Software development of a set of capable deep learning models to tackle a regression problem and able to run on heterogeneous hardware platforms.

2. Creation of a comparative framework based on the characteristics of the models.

3. Integration of monitoring and profiling tools that allow you to glimpse the operational processes of training and inference of the models and obtain temporary metrics.

4. Evaluation of the temporal performance of the training and inference processes of the models on different hardware platforms.

6.3. Plan of work

The work plan followed to achieve the objectives has been, in the first place, a thorough investigation and study on machine learning and deep learning. In the same way and at the same time, a familiarization with the Tensorflow framework for the Python programming language.

Once the conceptual part has been advanced, software fragments with very specific functionalities have been implemented to strengthen what has been learned and to test the different hardware architectures. Then, a software development environment has been formed that is capable of addressing training and inference in neural networks to subsequently proceed to research on tools capable of obtaining metrics of their execution and useful results for future evaluation.

Finally, generated the benchmark of models, studied the particularities of the hardware and with the skill acquired with the software tools after a practical learning, a comparison has been made that allows to know the behaviors of the different hardware platforms according to their nature when executing models based on neural networks.

6.4. Structure of the project

- Chapter 1: informs about the objectives of the work and how they will be developed.
- Chapter 2: includes details on the technologies to which the project resorts and its involvement in each task performed.
- Chapter 3: explains the entire process to build machine learning models and the monitoring of their training and inference processes.
- Chapter 4: describes the comparative process performed to achieve an evaluation of the temporal performance of the training and inference processes in the different hardware platforms.
- Chapter 5: shows the conclusions obtained and proposals for the continuation of their future development.

Chapter 7

Conclusions and Future Work

The first conclusion that has been obtained after the completion of the project is that it is essential to know the available tools and the functionalities provided by each of them to perform a work on machine learning. With this prior deselection process, considerable time can be saved. With the full integration of Keras in Tensorflow version 2.0, many of the tasks that could only be done previously in Tensorflow can be performed more easily without the need to implement training. As we have seen, through a few lines of code this API already allows a simple execution between several GPUs, therefore it would be convenient to study the new version and adapt all the models to gain in agility and functionalities.

Regarding the temporary evaluation of the models, after the comparisons made in the different devices, it can be affirmed that for the different types of perceptron models, for a small number of trainable parameters close to a million parameters, the multicore CPUs are a Good option for training. However, when the number of weights is large, the GPUs have better training times; also for neural networks with a high number of neurons in a layer, generally conditioned by the input layer, GPUs have a higher speed. Similarly, comparisons on the convolutional model have great advantage for graphics cards.

Regarding the inference, looking for conditions close to real time, it can be said that generally all tested devices have good behaviors. When the number of variables in the set of input increases, GPUs have better performance. However, for this use case, it can be said that using any device would be valid.

Finally it is convenient to highlight the versatility that Tensorflow presents to take care of on different platforms. With the execution of training through multiple GPUs, the training time has been reduced considerably. In the same way, it can be applied on embedded systems conducive to the inference process, with own software libraries to optimize performance. This would be convenient to test for situations in which time and consumption conditions are decisive as in an IoT context where inference close to the sensor is required.

Bibliografía

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., 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., Mane, 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., Villegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. y Zheng, X. Tensorow: Large-scale machine learning on heterogeneous distributed systems. *arXiv.org*, Disponible en `https://arxiv.org/abs/1603.04467v2`.

Abrahams, S., Hafner, D., Erwitt, E. y Scarpinelli, A. *TensorFlow for Machine Inteligence. A Hand-On Introduction to Learning Algorithms*. Bleeding Edge press, 2016.

AlgorithmsInside. Graph theory. Disponible en `https://algorithmsinsight.wordpress.com/graph-theory-2/`.

CS231. Cs231n standford. lectura 6 diapositiva 15. Disponible en `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf`.

Dabbura, I. Coding neural network forward propagation and backpropagtion. Disponible en `https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76`.

DotCSV. ¿ qué es una red neuronal? parte 1: La neurona. 2018.

Geron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

Goodfellow, I. Generative adversarial network (gan) in tensorflow. Disponible en `https://mlnotebook.github.io/post/GAN5/`.

Hale, J. Which deep learning framework is growing fastest? Disponible en `https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318`.

Kingma, D. P. y Ba, J. L. Adam: A method for stochastic optimization. *ICLR 2015*, Disponible en `https://arxiv.org/pdf/1412.6980v9.pdf`.

MarketRealist. Tough times for nvidia's gaming business with amd in the rearview. Disponible en `https://articles2.marketrealist.com/2019/07/tough-times-for-nvidias-gaming-business-with-amd-in-the-rearview/`.

McKinney, W. *Python for Data Analysis*. O'Reilly Media, Inc., 2012.

Moawad, A. Neural networks and back-propagation explained in a simple way. *Medium Corporation (US)*, Disponible en https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e.

Nvidia. Computación acelerada. Disponible en https://la.nvidia.com/object/what-is-gpu-computing-la.html.

Nvidia. Cuida zone. Disponible en https://developer.nvidia.com/cuda-zone.

Nvidia. Nvidia jetson. Disponible en https://developer.nvidia.com/embedded/develop/hardware.

Ramsundar, B. Cs224d: Tensorflow tutorial. Disponible en https://cs224d.stanford.edu/lectures/CS224d-Lecture7.pdf.

Rossel, G. Arquitecturas de redes neuronales. el perceptrón. Disponible en http://smartcomputing.gerardorossel.org/arquitecturas-.aspx.

Sengupta, M. y Andreas, A. Oahu solar measurement grid (1-year archive): 1-second solar irradiance; oahu, hawaii (data); nrel report no. da-5500-56506. Disponible en http://dx.doi.org/10.5439/1052451.

Shooter. Rise of the machines: Ai beats humans in multiplayer shooter quake iii arena. 2019.

Stackoverlow. what does dense[number] mean in model summary in keras. Disponible en https://stackoverflow.com/questions/50899660/what-does-dense-number-mean-in-model-summary-in-keras/.

TensorFlow. Tensorflow for mobile and iot. Disponible en https://www.tensorflow.org/lite.

Terceño Ortega, A. Análisis de un modelo predictivo basado en google cloud y tensorflow. 2017.

Vazquez, F. A weird introduction to deep learning. 2018.

Volvo. Volvo group partners with nvidia to develop advanced ai platform for autonomous trucks. Disponible en https://www.volvogroup.com/en-en/news/2019/jun/news-3340185.html.

Wikipedia (CUDA). Entrada: "CUDA". Disponible en https://es.wikipedia.org/wiki/CUDA (último acceso, Agosto, 2019).

Apéndice A

Especificaciones hardware

A continuación, se detallan las características más relevantes de los componentes Hardware utilizados durante la realización del trabajo.

CPUs

- **Intel Xeon E5**

Arquitectura:	x86_64
modo(s) de operación de las CPUs:	32-bit, 64-bit
Orden de los bytes:	Little Endian
CPU(s):	28
Lista de la(s) CPU(s) en línea:	0-27
Hilo(s) de procesamiento por núcleo:	1
Núcleo(s) por «socket»:	14
«Socket(s)»	2
Modo(s) NUMA:	2
ID de fabricante:	GenuineIntel
Familia de CPU:	6
Modelo:	63
Nombre del modelo:	Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
Revisión:	2
CPU MHz:	2200.000
CPU MHz máx.:	2301,0000
CPU MHz mín.:	1200,0000
BogoMIPS:	4599.64
Virtualización:	VT-x
Caché L1d:	32K
Caché L1i:	32K
Caché L2:	256K
Caché L3:	35840K
CPU(s) del nodo NUMA 0:	0-13
CPU(s) del nodo NUMA 1:	14-27

- **Intel Core i7-5700HQ CPU @ 2.70Ghz**

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	71
Model name:	Intel(R) Core(TM) i7-5700HQ CPU @
Stepping:	1
CPU MHz:	2701.000
CPU max MHz:	2701.0000
BogoMIPS:	5402.00
Virtualization:	VT-x
Hypervisor vendor:	vertical
Virtualization type:	full

- **GPUs**
  - **Nvidia GeForce GTX 1080 pascal**
    - GPU Engine Specs:
      2560 NVIDIA CUDA® Cores
      1607 Base Clock (MHz)
      1733 Boost Clock (MHz)
    - Memory Specs:
      10 GbpsMemory Speed
      8 GB GDDR5XStandard Memory Config
      256-bitMemory Interface Width
      320 Memory Bandwidth (GB/sec)
    - Thermal and Power Specs:
      94 Maximum GPU Temperature (in C) 180 WGraphics Card Power (W)
      500 WRecommended System Power (W)4
      8-Pin Supplementary Power Connectors
      Compute capability: 6.1
      Página web geforce-gtx-1080
  - **Nvidia GeForce GTX 980 maxwell**
    - GPU Engine Specs:
      2048 CUDA Cores
      Minimum 1064 MHz + BoostBase Clock
    - Memory Specs:
      7.0 GbpsMemory Speed
      GDDR5 Memory Interface
      256-bit Memory Interface Width
      224 Memory Bandwidth (GB/sec)
      Compute capability: 5.2
      Página web geforce-gtx-980

- **GTX 950M Engine Specs**
  - GPU Engine Specs:
    640CUDA Cores, 914 + BoostBase Clock (MHz)*
  - GTX 950M Memory Specs:
    1000 or 2500 MHzMemory Clock
    DDR3 or GDDR5Memory Interface
    128-bitMemory Interface Width
    32 or 80Memory Bandwidth (GB/sec)
  - GTX 950M Technology Support:
    YesCUDA
    Compute capability: 5.0
    Página web geforce-gtx-950m

▪ Microcontrolador NVIDIA Jetson TX2 (ARM)

- GPU
  256-core NVIDIA Pascal^{TM} GPU architecture with 256 NVIDIA CUDA cores
- CPU
  Dual-Core NVIDIA Denver 2 64-Bit CPU
  Quad-Core ARM® Cortex®-A57 MPCore
  Memory: 8GB 128-bit LPDDR4 Memory 1866 MHx - 59.7 GB/s
  Storage: 32GB eMMC 5.1
  Power: 7.5W / 15W
  Características de los modelos de microcomputadores Nvidia Jetson TX2

	TX2 4GB	TX2	TX2i
AI Performance	1.3 TFLOPs		
GPU	256-core NVIDIA Pascal^{TM} GPU		
CPU	Dual-Core NVIDIA Denver 2 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore		
Memory	4GB 128-bit LPDDR4 Memory 1600 MHz 51.2 GBs	8GB 128-bit LPDDR4 Memory 1866MHz 59.7 GB/s	8GB 128-bit LPDDR4 (ECC support) 1600MHz 51.2 GB/s
Storage	16GB eMMC5.1	32GB eMMC5.1	32GB eMMC5.1
Power	7.5W / 15W		10W / 20W
PCIE	Gen 2 | 1x4 + 1x1 OR 2x1 + 1x2		
CSI	12x CSI-2 D-PHY 1.2 (Up to 30 GB/s)		
Display	Two Multi-Mode DP 1.2 eDP 1.4 HDMI 2.0 Two 1x4 DSI (1.5Gbps/lane)		
DL Accelerator	-		
Vision Accelerator	-		
Wi-Fi	No	Yes	No

Para ampliar diferencias entre las memorias de las GPUs Nvidia:

- Memoria GDDR5 SDRAM
- Memoria LPDDR

Para ampliar diferencias entre las arquitecturas de las GPUs Nvidia:

- Arquitectura Pascal

Apéndice B

SW y enlaces de instalación

- **Visual Studio Code:**

  IDE multilenguaje de Microsoft que cuenta con un complemeto para python que permite utilizar un terminal sobre entornos virtuales para la ejecución del código.

  Para instalar en Windows

- **CUDA:**

  Entorno de Nvidia que ofrece diversas funcionalidades pensando en la computación de propósito general mediante la aceleración con GPUs Nvidia CUDA.

  Para instalar drivers CUDA en Windows

- **Gestores de entornos Python-Tensorflow:**

  Conda: gestor de entornos y paquetes para distintos lenguajes de programación en diversos sistemas operativos. Para Windows requiere de la instalación de Anaconda.

  Para instalar anaconda en hrefhttps://www.anaconda.com/distribution/Windows

  Python virtualenv y pip: gestores de entornos y paquetes genéricos de Python y manejables desde la instalación de Python.

  Para la instalación de Python.

- **R** y **RStudio**:

  Lenguaje e IDE para análisis estadístico y visualización.

  Para instalar RStudio.

- **Linux** client para windows 10 y el protocolo **SSH**:

  Secure shell scrip permite interactual con máquinas remotamente mediante una conexión segura a nivel de red por el puerto 22. Para utilizar este protocolo se ha instalado el cliente Linux para Windows10.

  Para instalar en Windows.

- **Putty scp, Pscp**:

  Para la transferencia de archivos entre máquinas se ha utilizado el comando scp de linux, evitando usar el protocolo ftp. Este comando se apoya sobre una conexión ssh autenticada entre las dos máquinas. Gracias a la instalación de un cliente putty en el ordenador local Windows, se agilizó este proceso de transferencia entre máquinas con

diferentes sistemas operativos. En este caso, ha sido el sistema Windows el actuador desde este cliente putty usando el comando pscp desde su consola.

Para la instalación en Windows.

Apéndice C

Comandos útiles

En este apéndice se incluyen comandos útiles utilizados en el proyecto. Además, todos los códigos de este proyecto se encuentran en el repositorio
https://github.com/lorrenlml/testingtensorflow

- Ejecutar script de python:

  $ python SCRIPT.py

- Ejecutar script de python con fichero de configuración:

  $ python SCRIPT.py –NNfile CONFIG.json

- Instalación de un entorno mediante pip:

  pip install tensorflow==2.0.0-beta1

- Activación de un entorno python con tensorflow localizado en el directorio /optshared/tensorflow_p3_gpu/:

  $ cd /optshared/tensorflow_p3_gpu/ && . bin/activate

- En dispositivos con conda:

  - Instalación de entornos tensorflow para CPU y GPU:
    $conda create -n tensorflow_env tensorflow
    $conda create -n tensorflow_gpuenv tensorflow-gpu
  - Activación de entorno CPU y GPU:
    $conda activate tensorflow_env
    $conda activate tensorflow_gpuenv
  - Información de entornos disponibles:
    $conda info –envs
  - Información de paquetes instalados en un entorno:
    $conda list

- En Jetson TX2 activar entorno: $workon tf1.13

- Ejemplos de carga de librerías mediante variables de entorno:

export LD_LIBRARY_PATH=/optshared/cuda-9.0/lib64/:
/optshared/common/lib/:/optshared/cuda-9.0/extras/CUPTI/lib64/:
$LD_LIBRARY_PATH

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/
usr/local/cuda-10.0/extras/CUPTI/lib64/

- Visualización de Tensorboard en el navegador:
  - En un entorno tensorflow activar el servidor Tensorboard:
    $ tensorboard –logdir=directoriodelogs
  - En un navegador acceder al puerto 6006:
    localhost:6006
- Visualización de trazas de ejecución en un navegador:
  - chrome://tracing
  - LOAD: para cargar el fichero .json generado
- Comando para una conexión SSH mediante un cliente linux.

  $ ssh user@HOST

  Siendo los hosts:
  - lormar@bujaruelo.dacya.ucm.es
  - lormar@sardina.dacya.ucm.es
- Dentro de sardina para acceder a Jetson TX2 se debe ejecutar:

  $ ssh tegra2 -l lormar
- Transferencia de archivos mediante pscp desde el cmd de Windows:
  - De windows a Linux:
    C:\ Users \ Lorenzo \ Documents\ TFM\ CODES>pscp CONVOLUCIONALES.zip lormar@bujaruelo.dacya.ucm.es:
  - De Linux a windows: (al reves)
    C:\ Users \ Lorenzo \ Documents \ TFM \ CODES>pscp lormar
    @bujaruelo.dacya.ucm.es:solarcasting-ml/ICMAT/proposal.md C:\Users\ Lorenzo \Documents
- Comando para comprobar el rendimiento y los procesos de una GPU Nvidia:

  $ nvidia-smi
- Ver capacidades hardware del dispositivo:

  $ lscpu

  $ df

  $ htop

Apéndice D

Códigos y fichero de configuración

- Códigos python tensorflow:
  - Prueba rápida de eficiencia para una misma tarea en tensorflow sobre CPU y GPU.
    0quick_performance_test.py
  - Código original en sklearn de Guillermo Yepes:
    1NN_model_sklearn.py
  - Generador de redes piramidales:
    2nets_generator.py
  - Profiler de redes neuronales fully connected.
    3fullyconnected_nets_profiler.py
  - Extractor de tiempos de red convolucional Keras.
    4keras_CNN_prof.py
  - Extrator de predicciones de un modelo fully connected.
    5checkeador.py
  - Inferencia sobre múltiples modelos fully connected.
    6inferencia_redes.py
  - Carga de un modelo tensorflow y realización de inferencia.
    7inferencia_grafo_importado.py
  - Carga de un modelo keras y realización de inferencia.
    8inferencia_conv.py
  - Entrenamiento distribuido con Keras y TF 2.
    directorio /distrib
- Se ha dejado identificado cada uno de los scripts con los ficheros de configuración mediante el número que comparten en su nombre. Para el script 7 será necesario el conjunto de ficheros que definen el grafo y sus pesos.
- Ficheros de configuración:

  Los ficheros de configuración se pasan para la ejecución del programa python 3fully-connected_nets_profiler.py de esta forma:

  python programa.py –NNfile configuracion.json

Aquí se muestran todos los parámetros modificables que se pueden encontrar en ellos:

orig_folder:	*directorio donde se localizan los datos*
des_folder:	*no aplica*
hor_pred:	*horizonte de predicción en segundos*
days_info:	*información de los días para el preprocesado*
features:	*variables del dataset*
train_size:	*numero de iteraciones sobre el conjunto de entrenamiento para el preprocesado*
hls:	*numero de neuronas por red de partida para el generador de redes*
n_nets :	*número de redes a generar*
batch_size :	*tamaño del lote*
learning_rate :	*tasa de aprendizaje*
epochs :	*iteraciones sobre el conjunto de entrenamiento en el entrenamiento alpha: factor del preprocesado*
time_granularity:	*granularidad temporal de las muestras seleccionadas*
seed:	*aleatorización del preprocesado*
img_rows:	*dimensión horizontal de la imagen*
img_cols:	*dimensión vertical de la imagen*
device:	*dispositivo utilizado*
device_name:	*modelo del dispositivo utilizado*
cut:	*porcentaje del tamaño del conjunto de validación*

- Ejemplos de ficheros de configuración para cada caso:
  - 1NN_models_sklearn.json
  - 3config_nets_training.json
  - 4config_nets_training.json