Análisis de un modelo predictivo basado en Google Cloud y TensorFlow

Alberto Terceño Ortega

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Grado

Curso 2016/2017

Directora: Victoria López López
Codirector: Alberto Font Rytzner

Dedicado a
la memoria de mi padre

Agradecimientos

Aprovecho esta página para agradecer a mi familia el esfuerzo y los ánimos para que yo ahora pueda estar escribiendo estas líneas en este trabajo. Tampoco puedo olvidarme de mi otra familia de Madrid, ni del Chami, que cambió mi vida para siempre y me dio los mejores amigos de mi vida.

Este trabajo va también para mi eterno compañero de prácticas y de estudio. Gracias David por todos esos consejos y ayudas en la sala de la televisión, sin todos ellos creo que jamás se me habría ocurrido hacer un trabajo de este tipo.

Es increíble, pero todavía no he sido capaz de abandonar la nostalgia del Erasmus. No me puedo olvidar de Copenhague y de toda la gente que conocí en aquel año mágico. Este trabajo lleva una parte de todos vosotros. No hay ninguna duda de que también lleva un trozo de Benot: os estaré siempre agradecido por saber estar conmigo en los momentos buenos y en los malos.

Por último, gracias a Victoria y a Alberto por dejarme embarcar en este proyecto. Todavía recuerdo aquella llamada de Alberto y como fue capaz de convencerme mientras yo andaba dando vueltas en círculos en Piazza Navona, dudando hasta el extremo del proyecto que me estaba planteando. No se como pude estar tan equivocado.

Índice general

Índice de figuras

Índice de tablas

Resumen

Con este trabajo se propone un ecosistema de herramientas que permitan analizar y desarrollar modelos predictivos utilizando técnicas de deep learning, en particular redes neuronales. Con la ayuda de TensorFlow, una librería diseñada para resolver problemas de aprendizaje automático, se desarollará en local un benchmark que permita evaluar diferentes topologías de redes neuronales sobre un conjunto de datos.

Además, con el objetivo de poder analizar modelos en la nube utilizando grandes cantidades de datos, completaremos el ecosistema con dos herramientas de Google Cloud Platform: Datalab y ML Engine. La primera de ellas nos permitirá realizar una exploración y análisis inicial de datasets, incluyendo entrenamientos con redes neuronales para muestras de datos. Por otro lado, con ML Engine podremos realizar en la nube entrenamientos con datos a gran escala utilizando modelos *Wide and Deep*, los cuales combinan las ventajas de las regresiones lineales y no lineales.

A lo largo del trabajo se proporcionarán ejemplos y código para saber como trabajar con las distintas herramientas propuestas. En particular, se analizará un modelo predictivo con el objetivo de discernir transferencias fraudulentas en tarjetas de crédito.

Palabras clave: TensorFlow, redes neuronales, Datalab, ML Engine, Google Cloud Platform, deep learning, aprendizaje automático, big data, Keras.

Abstract

This bachelor thesis is intended to introduce a set of tools in order to analyze and develop predictive models using deep learning techniques, in particular neural networks. Thanks to TensorFlow, a framework designed to solve machine learning problems, a local benchmark will be coded, so different neural networks can be tested on a dataset.

What is more, in order to achieve cloud analysis on models using massive datasets, a couple of Google Cloud Platform services will be added up to the tools presented: Datalab and ML Engine. The first one will allow us firstly to explore and analyze datasets, including neural networks trainings for sampled data. On the other hand, ML Engine will let us train large-scale datasets using *Wide and Deep* models, which benefit from both linear and non linear regressions.

Samples and code will be presented throughout the whole thesis, in order to know how to use the proposed tools. Particularly, we will analyse a predictive model that detects fraudulent transactions in credit cards.

Keywords: TensorFlow, neural networks, Datalab, ML Engine, Google Cloud Platform, deep learning, machine learning, big data, Keras.

1 Introducción

Aunque muchas de las técnicas utilizadas por algoritmos de deep learning fueron elaboradas de forma teórica el siglo pasado, en la actualidad están cobrando fuerza, usándose en modelos predictivos de todo tipo [73]. Esto ha sido en buena medida gracias a la cantidad de datos que es posible recolectar actualmente, los cuales pueden ser almacenados y tratados por máquinas con gran capacidad de cómputo.

En particular distintos tipos de redes neuronales se han ajustado con éxito en distintos campos: desde redes neuronales distribuidas que recomiendan películas [4], hasta redes neuronales recurrentes (LSTMs) que corrigen errores gramaticales [11], pasando por redes convolucionales (CNN) capaces de reconocer señales de tráfico [54].

Con el objetivo de resolver problemas de clasificación (binarios y multiclase) nace el benchmark que he construido sobre TensorFlow 1.1.0 y que podemos encontrar en el repositorio Github del trabajo, junto al resto del código desarrollado [55]. De cara a otorgar flexibilidad al usuario, este benchmark ofrece un variado número de parámetros adicionales (también llamados hiperparámetros de la red neuronal) y técnicas de optimización, siendo por lo tanto apto para adaptarse a problemas con diferentes casuísticas.

Además, he querido acompañar este benchmark con otras dos herramientas de Google Cloud (Datalab y ML Engine) que permitirán enfrentarnos a cantidades de datos superiores con las que podría tratar el benchmark local. Con la combinación de estas tres herramientas podremos obtener un modelo final óptimo, así como un conjunto de logs y gráficos sobre las distintas pruebas realizadas sobre el modelo, para poder de esta manera evaluar de forma rápida y sencilla su rendimiento.

1.1. Motivación del problema y plan de trabajo

El origen de este trabajo se encuentra en un proyecto de investigación interno realizado en la empresa The Cocktail Experience, donde empecé unas prácticas extracurriculares el 17 de abril de 2017.

En un primer momento se me pidió probar a utilizar redes neuronales en un proyecto de la compañía que intentaba predecir el comportamiento de usuarios en internet. La razón de ésto fue comparar los resultados obtenidos con las redes neuronales frente a otros algoritmos utilizados en ese proyecto. Aunque se logró un mismo porcentaje de aciertos en las predicciones, las redes neuronales ofrecieron un tiempo de entrenamiento y de

cálculo de predicciones ostensiblemente menor al que se venía haciendo en el proyecto con algoritmos en R.

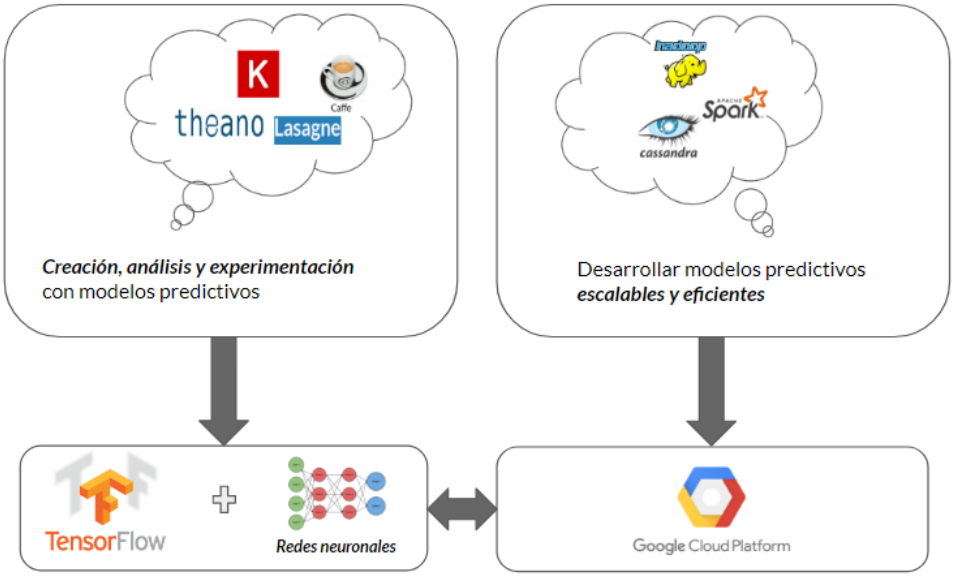


Figura 1.1: Objetivos y herramientas propuestas para el trabajo

Fuente: Elaboración propia

Otro de los problemas que surgió durante el proyecto fue la inestabilidad de RStudio (un IDE para el lenguaje de programación R) al cargar y operar con grandes volúmenes de datos (en torno a 5 millones de observaciones y 50 variables) y la lentitud a la hora de computar algoritmos de selección de variables en R. Debido al hecho de que las redes neuronales que se construyeron utilizaban la API de TensorFlow para Python, se empezó a observar como estos problemas se iban solventando. Esto se debe principalmente a que R utiliza por defecto un solo hilo de ejecución [23], mientras que TensorFlow puede ejecutarse sobre varios hilos [5].

Por último, también existía la dificultad de que la parte de entrenamiento que realizaban los algoritmos del proyecto se hacía exclusivamente en local, usando además en algunos casos pequeñas muestras de datos. Esto hizo que, para finalizar el proyecto de investigación, tuviera que dedicarme a estudiar soluciones y herramientas en la nube que permitieran entrenar en remoto modelos con conjuntos de datos completos, sin muestreo de ningún tipo. Finalmente escogí Datalab y ML Engine, que se usarán en este proyecto para la exploración y el entrenamiento a gran escala respectivamente.

En la figura 1.1 se resumen los objetivos que nos hemos marcado y el stack tecnológico en el que nos basamos. En general, con este trabajo queremos conseguir un análisis

exhaustivo de diferentes modelos que den solución a problemas de tipo predictivo (por ejemplo imaginemos una entidad bancaria intentando predecir el riesgo de impago de sus clientes o una institución educativa que pretende predecir si sus alumnos están teniendo problemas a la hora de cursar una asignatura) mediante redes neuronales, sin olvidarnos de cuestiones como la escalabilidad, en el caso de que la cantidad de datos a procesar sea realmente grande. En los apéndices 3, 4 y 5 podemos encontrar ejemplos de análisis detallados de resultados obtenidos a partir de las distintas herramientas del trabajo.

1.2. Dataset de ejemplo (Credit Card Fraud Detection)

Con el objetivo de experimentar con un modelo predictivo que muestre las funcionalidades de cada una de las tres fases principales del proyecto, se ha escogido un dataset del portal Kaggle [37]. Este dataset consiste en 284.407 transacciones realizadas mediante tarjetas de crédito a lo largo de dos días en septiembre de 2013 [17, 68]. Cada una de estas transferencias está etiquetada si es fraudulenta o no con un 1 o un 0 respectivamente (variable denominada *Class* en el dataset), por lo que el problema que tenemos que resolver es predecir si una transferencia dada es fraudulenta o no. Tal y como se comprueba en el capítulo 3, se trata de un dataset realmente desbalanceado, pues únicamente el 0.17 % de las transferencias totales son fraudulentas. Para tratar esta problemática se ha propuesto, o bien realizar *undersampling* del dataset, o bien evaluar modelos con la métrica AUC.

Además, para entrenar diversos modelos predictivos dispondremos de otras 30 variables para cada transacción realizada: por un lado está la variable de tiempo (denominada *Time*, y que contiene la diferencia en segundos transcurridos desde que ocurrió la primera transacción hasta la transacción analizada) y la cantidad de dinero transferida (variable *Amount*). Por otro lado, tenemos 28 variables adicionales (denominadas $V1, \ldots, V28$), resultado de un PCA previo y anonimizadas por motivos de confidencialidad según la descripción del dataset [17].

Por cuestión de notación, a lo largo del trabajo denominaremos a este dataset como CCF, siglas pertenecientes a “Credit Card Fraud”.

1.3. Estructura de la memoria

La memoria se ha estructurado en tres partes bien diferenciadas y que se corresponden a los capítulos 3, 4 y 5: una primera parte de análisis de datos en la nube con Datalab, una segunda donde se desarrolla con TensorFlow un benchmark local de entrenamiento, y, la última, donde se expone como realizar (a través de ML Engine) entrenamientos en la nube con una gran cantidad de datos usando modelos *Wide and Deep*. Sin embargo, debido al importante número de herramientas y librerías que utilizaremos en el trabajo, dedicaremos en primer lugar un capítulo al estado del arte.

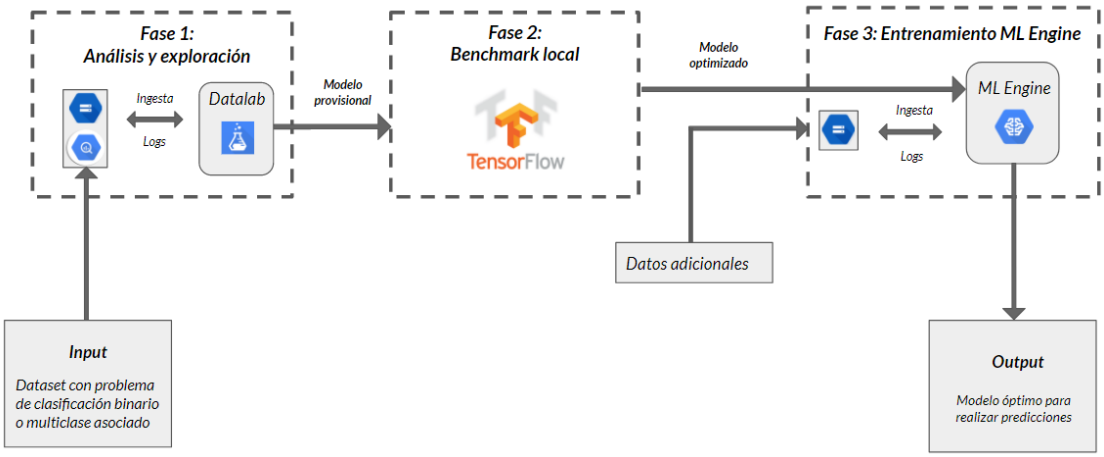


Figura 1.2: Pipeline del ecosistema desarrollado

Fuente: Elaboración propia

El motivo del orden en el que se presentan estos tres capítulos es que se concibe el ecosistema de herramientas propuestas como un marco de trabajo de tres etapas (correspondientes con cada uno de los tres capítulos que mencionábamos anteriormente) con el que se puede trabajar para desarrollar y sobre todo analizar un modelo predictivo cualquiera. El marco diseñado puede verse en la figura 1.2.

Para finalizar, en el último capítulo de la memoria se realiza una valoración del trabajo realizado, así como la presentación de una serie de propuestas para el trabajo que podrían realizarse en un futuro.

2 Estado del arte

2.1. Google Cloud

Google Cloud es una colección de servicios en la nube (podemos ver algunos de ellos en la figura 2.1) que ofrece la multinacional Google. La principal ventaja de esta plataforma frente a otras es la fiabilidad, rapidez y escalabilidad que presenta, dado que la infraestructura y la tecnología utilizada es la misma que la usada por Google para sus propios productos. Esto permite olvidarse de tareas como el mantenimiento y administración de servidores o la configuración de redes [45].

Figura 2.1: Pequeña muestra de distintas herramientas de Google Cloud Platform

Fuente: Google Imágenes

Además, este conjunto de servicios va renovándose con el paso del tiempo aportando APIs novedosas en campos de investigación actuales como Big Data o aprendizaje automático. Sin embargo, esto también es una de las desventajas de estas plataformas en la nube, puesto que ante la abundancia de herramientas ciertas APIs pueden verse descontinuadas.

Otros servicios similares a Cloud Platform son Microsoft Azure, Amazon Web Services o IBM Cloud.

2.1.1. Conceptos básicos

A continuación presentaremos distintos términos relativos a Google Cloud que se utilizan a lo largo del trabajo. Este apartado se ha obtenido a partir de la página de documentación de Google Cloud Platform [8].

En primer lugar tenemos el concepto de *región*. Los distintos recursos que ofrece Google a través de Cloud se encuentran en centros de datos (*data centers*) situados en distintas partes del mundo. Estos centros se sitúan en *regiones globales* como Europa occidental (*europe-west1* y *europe-west2*) o Australia sudoriental (*australia-southeast1*). Además, estas regiones se dividen en *zonas* (por ejemplo: *us-central1-a*, *europe-west1-b*, etc.). Nótese que para distinguir una zona de otra necesitamos el nombre de la región asociada y la zona, ambas separadas por un guión. Esta división por regiones permite tolerancia a fallos en los centros de datos, así como la posibilidad de clasificar recursos según su zona (discos duros de máquinas virtuales) o región (IPs estáticas).

Por otro lado, tenemos *proyectos*, los cuales permiten englobar distintos servicios de Cloud Platform bajo unos permisos y ajustes determinados. Un proyecto consta de un *nombre de proyecto* (proporcionado por el usuario), un *ID* (proporcionado por el usuario o por Cloud Platform) y un *número de proyecto* (proporcionado por Cloud Platform). Estas tres variables son necesarias para realizar llamadas a las diversas APIs de Google Cloud Platform.

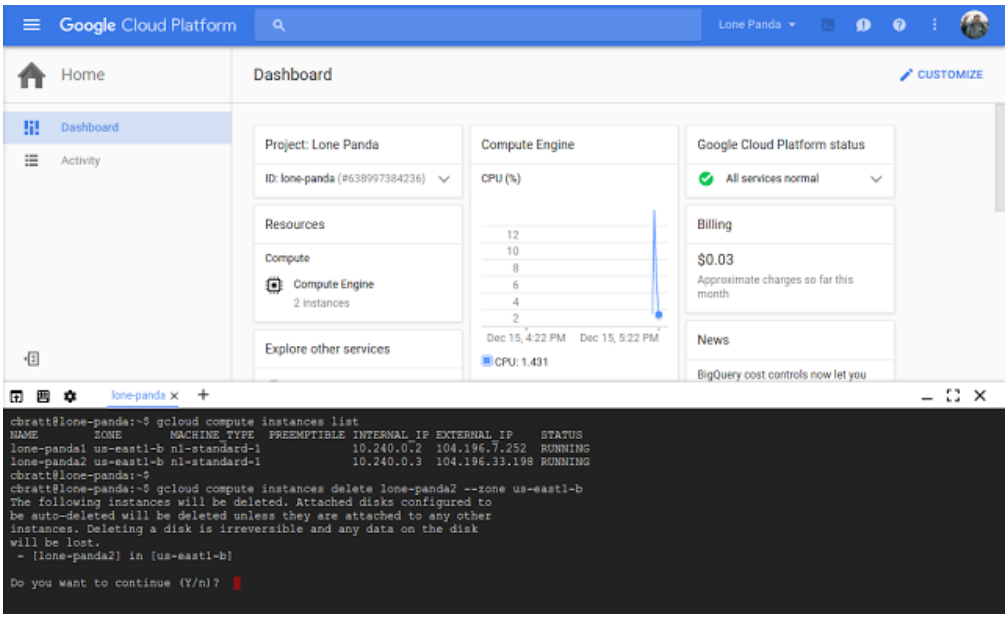


Figura 2.2: Captura de Google Cloud Platform Console y la Cloud Shell en la parte inferior.

Fuente: Google Imágenes

En último lugar, presentaremos dos formas distintas de conectar con los servicios de Cloud Platform. La primera es Google Cloud Platform Console [28], una interfaz gráfica que nos permite acceder a los servicios de Google Cloud a través del navegador. Por medio de esta interfaz se puede acceder también a Google Cloud Shell, una terminal Linux con la que podremos utilizar comandos del SDK de Google. Este software (disponible para Windows, Mac y Linux) es la segunda forma de conectar con las herramientas de Cloud Platform. Gracias al SDK, podremos usar en la terminal local de nuestro ordenador el comando *gcloud*, el cual permite interactuar con distintos servicios de Google Cloud Platform.

2.1.2. Precio

El precio de Google Cloud Platform es calculado por uso (normalmente por hora o incluso por mes) de cada una de las herramientas existentes. En las distintas secciones del trabajo se detallan los precios según los servicios que se hayan utilizado. En el presente trabajo se ha utilizado una cuenta personal de Google Cloud Platform, aunque hay que destacar que existen otras dos opciones gratuitas que pueden usarse para poder reproducir las distintas fases del trabajo. Por un lado existe una prueba gratuita durante 12 meses con posibilidad de gastar hasta 300$ [48], lo cual da margen necesario para hacer pruebas exhaustivas con Cloud Platform (incluyendo servicios algo más costosos, como ML Engine). Finalmente, comentar que existe un programa de Google en el que instituciones educativas pueden apuntarse para conseguir *créditos* para los alumnos [47].

2.1.3. Datalab

Se trata de una herramienta (a fecha de julio de 2017 en fase Beta) desarrollada por Google que permite el análisis, exploración y visualización de datos [21]. El código de Datalab se puede encontrar bajo licencia Apache en Github [50] y además la herramienta está diseñada para que pueda ser integrada con otros servicios de Cloud Platform como BigQuery o Cloud Storage. Para ello se basa en Jupyter Notebook (anteriormente denominado IPython), una aplicación web que sirve como entorno de programación para Python, donde el código se escribe en *notebooks* que se dividen en *celdas* que pueden irse ejecutando de manera interactiva.

Datalab se encuentra alojada en un contenedor de Docker[1] y puede ejecutarse tanto en local como en una instancia de Google Compute Engine, esto es, máquinas virtuales que forman parte de la infraestructura de Google y que éste ofrece bajo demanda como un producto más de Cloud Platform. En este trabajo se utiliza Datalab sobre Google Compute Engine.

Debido al fuerte componente de integración de Datalab con Cloud Platform, es bastante recomendable consultar las distintas formas de conectar Datalab a otros productos

[1] Se trata de una popular aplicación que permite virtualizar entornos (también denominados *contenedores*) dentro de Windows y Linux.

(BigQuery, por ejemplo) en los notebooks proporcionados [52]. Cabe destacar que varios de estos archivos han servido de inspiración para el desarrollo de los notebooks de este trabajo, por lo que he decidido incluirlos en el repositorio Github del proyecto [55].

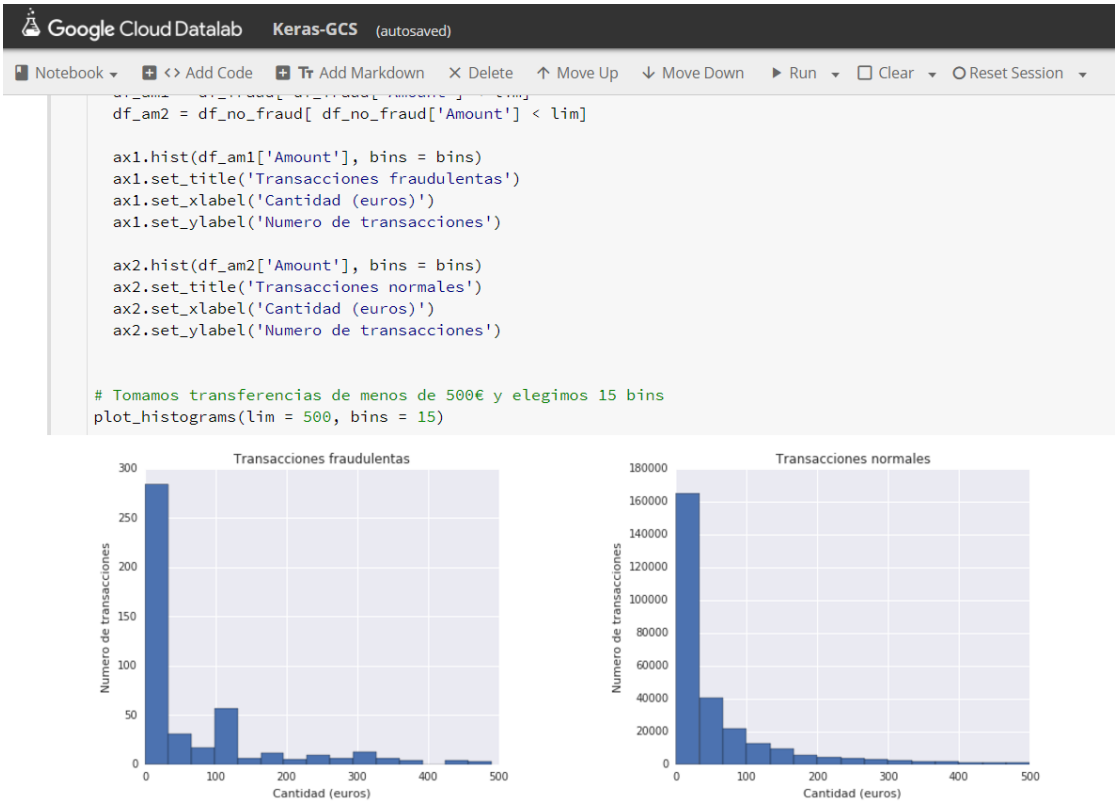


Figura 2.3: Captura de pantalla de Datalab

Fuente: Keras-GCS.ipynb [55]

También es interesante comentar otras alternativas a Datalab que, a pesar de dificultar tareas como la ingesta de datos[2], conservan en general la funcionalidad de esta herramienta. Por ejemplo, es posible instalar Jupyter Notebook en un servidor [33], pudiéndose extender a máquinas individuales de Amazon Web Services [31] o clusters de Google Cloud Platform [32].

2.1.4. Cloud Storage

Google Cloud Storage es el servicio de almacenamiento en la nube de Google Cloud Platform. La ventaja de este servicio es que permite almacenar datos en servidores de

[2]En otros casos tendríamos que utilizar APIs como la de Cloud Storage para Python [38], que es más compleja de instalar y utilizar que la API nativa de Storage para Datalab.

distintas regiones del mundo, garantizando su persistencia y disponibilidad. Además, Cloud Storage es escalable, lo que permite que podamos depositar archivos de hasta varios exabytes de datos [29]. Estas razones, así como la facilidad de integración con Datalab y ML Engine, han sido determinantes para que haya sido elegida junto a BigQuery como una de las herramientas utilizadas para la ingesta de datos en la nube. Por último, mencionar que también existen otras alternativas (bastante similares a Storage) de almacenamiento en la nube, tales como Amazon S3 o Microsoft Azure Storage.

2.1.5. BigQuery

Big Query es un producto de Google Cloud que se utiliza como almacén de datos masivos (*data warehouse*), ofreciendo también soporte para queries interactivas en SQL a través de una interfaz gráfica. Este producto es de hecho una implementación pública de las características fundamentales de la tecnología Dremel, usada internamente en Google para tareas como análisis de spam o generación de informes de fallo de distintos productos de la compañía [74]. Big Query comparte con Dremel el rendimiento, así como la estructura interna, compuesta de datos ordenados por columnas (*column-oriented storage*) y la división de queries en servidores a través de una estructura en forma de árbol.

Figura 2.4: Logo de BigQuery

Fuente: Google Imágenes

El acceso a Big Query se realiza de diversas maneras: a través de un cliente web, por terminal (gracias a la API, disponible para Java, Go o Python entre otros lenguajes) o por software de terceros. En este trabajo se utiliza la API integrada dentro de Datalab, facilitando en gran manera la integración de BigQuery con Datalab.

Por último, destacar que las principales alternativas a Big Query que podemos encontrar en el mercado son Amazon RedShift (perteneciente a Amazon Web Services) y Apache Drill, este último con licencia Apache, a diferencia del resto.

2.1.6. ML Engine

Cloud Machine Learning Engine es una herramienta de Google Cloud Platform que permite realizar entrenamientos y obtener predicciones con modelos de la librería TensorFlow, todo ello de manera remota. En este trabajo nos centraremos exclusivamente en la primera funcionalidad, que ofrece servicios como el ajuste autómatico de hiperpárametros (*hyperparameter tuning*) para redes neuronales o entrenamientos en determinados clusters (siguiendo el modelo de ejecución distribuida de la sección 2.2.2). Gracias a esta última característica y al hecho de que ML Engine posee una fuerte integración con Cloud Storage, podremos realizar sin problemas entrenamientos con datasets de millones de filas.

Otra alternativa a este producto es Amazon Machine Learning, aunque por el momento (julio de 2017) no se pueden usar modelos propios de aprendizaje automático ni exportar fuera de Amazon Web Services los modelos obtenidos al usar la herramienta [27].

2.2. TensorFlow

TensorFlow es una librería orientada a la creación de diversos modelos de aprendizaje automático, permitiendo su ejecución de manera eficiente en equipos con un rango de prestaciones hardware muy variable (por ejemplo, desde dispositivos móviles a grandes centros de procesamiento de datos con numerosas GPUs) [65]. Esta librería fue liberada bajo licencia Apache en 2015 por el equipo de investigación de Google Brain como sucesor de DistBelief [69], desarrollado en 2011 para productos internos de Google tales como Street View [66] o YouTube [72].

Figura 2.5: Logo de TensorFlow

Fuente: Google Imágenes

La API más estable y más desarrollada a fecha de julio de 2017 es la de Python (que es la elegida para realizar este trabajo), aunque existen otras librerías, menos desarrolladas, para C++, Java y Go. Aunque recientemente TensorFlow se ha convertido en una librería muy popular (podemos comprobar esto en estadísticas recientes de Github [56]) también existen otras librerías similares como DeepLearning4J, Caffe o Theano. Las razones que nos han llevado a elegir TensorFlow por encima de otras han sido: la gran

comunidad de usuarios que colabora con TensorFlow, su eficiencia (está optimizado para ejecutarse sobre GPUs), la facilidad de integración con herramientas como ML Engine y la posibilidad de visualizar resultados a través de TensorBoard.

Para finalizar, mencionar que los dos artículos fundamentales para comprender el funcionamiento interno de TensorFlow, los cuales han sido utilizados para desarrollar esta sección, se pueden encontrar en la bibliografía adjunta [64], [65].

2.2.1. Funcionamiento

El modelo de funcionamiento de TensorFlow es el siguiente: los cómputos se realizan gracias a un grafo dirigido, donde los datos van siguiendo un flujo determinado. Podemos ver un ejemplo de estos grafos en la figura 2.6. Cada uno de los nodos que forman este grafo poseen unos determinados inputs y outputs y representan una operación, la cual posee unos *atributos* necesarios para su ejecución, facilitando de esta manera un cierto polimorfismo. Además, estas operaciones pueden tener implementaciones concretas para un determinado hardware, recibiendo el nombre de *kernel*.

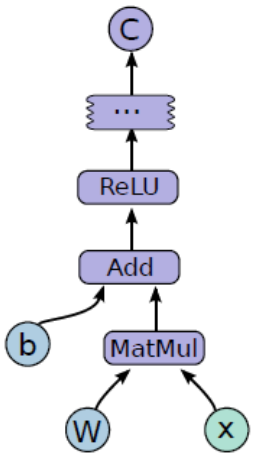


Figura 2.6: Ejemplo de grafo de cómputo para una capa de una red neuronal

Fuente: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems [64]

Por otro lado, los estructuras de datos que van desde outputs a inputs por las aristas del grafo se denominan *tensores* (*tensors*), similares a arrays o listas pero con un determinado tipo y dimensión. Sin embargo, los valores de estos tensores se conciben como valores intermedios del grafo por lo que los parámetros de los modelos se suelen guardar en *variables*, una clase especial de operaciones. También existe otro tipo de vértices llamados *dependencias de control*, que aseguran que el cómputo en el nodo origen termine

antes de que la operación comience en el nodo destino. Finalmente, la manera para que los clientes puedan interaccionar con estos gráficos de cómputo se lleva a cabo por medio de *sesiones*, las cuales poseen un método *run* que devuelve los tensores pedidos a partir de otros que previamente se han calculado o que incluso hayamos podido proporcionar.

2.2.2. Ejecución distribuida

Antes de explicar las posibilidades de paralelizar código con TensorFlow, comentaremos brevemente algunos conceptos relativos a la implementación de TensorFlow, claves para entender como se consigue esta paralelización .

Al lanzar un programa, TensorFlow se ocupa de crear tres procesos: *cliente*, *master* y *trabajador*. El cliente se comunica por medio de *sesiones* con el master, el cual crea uno o varios trabajadores que a su vez acceden a *dispositivos*, a saber, GPUs y CPUs. Como es de suponer, los encargados de realizar los cómputos de los distintos nodos del grafo son los trabajadores. En el caso de que tengamos un único dispositivo, por ejemplo una sola CPU, TensorFlow creará un cliente, un master y un único trabajador que accederá a este recurso. Sin embargo, en caso de que haya varios dispositivos se añade una dificultad al proceso, pues por un lado se debe decidir qué dispositivo se asigna a cada nodo del grafo y por otro se complica la coordinación entre los resultados devueltos por estos dispositivos. Para resolver el primer problema se usa un algoritmo voraz que realiza la asignación en función de datos heurísticos (tipo de la operación del nodo, restricciones de dispositivos a la hora de ejecutar ciertas operaciones, ejecuciones previas del grafo,etc.) [64]. En cuanto al segundo problema, una vez ya asignados los dispositivos, se procede a partir el grafo de cómputo en subgrafos de tal manera que todos los nodos de esos subgrafos pertenezcan al mismo dispositivo. Ahora bien, sustituyendo las aristas originales que van de un subgrafo a otro (es decir, que cruzan dispositivos) por nodos de envío y recepción, la comunicación entre dispositivos y trabajadores se facilitará en gran manera.

En este punto podemos proceder a explicar las posibilidades de ejecución distribuida que ofrece TensorFlow. En primer lugar, tenemos que decidir si queremos realizar la distribución de los cómputos a nivel de datos o a nivel del grafo. En el primer caso, como podemos observar en las imágenes de la figura 2.7, se replica en varias máquinas el grafo que hayamos creado, de manera que los datos de entrenamiento queden divididos entre estas réplicas. Tras ejecutar nuestros modelos obtenemos la actualización de nuestros parámetros (denotada en la figura 2.7 como $\Delta\omega$), pudiendo aplicar estos cambios a los grafos de manera síncrona o asíncrona. Nótese como en estos casos se instancian tantos clientes como réplicas tengamos. Este tipo de distribución se puede realizar con algoritmos de entrenamiento que trabajen por *batches* o lotes, como es el caso del descenso de gradiente estocástico. Este tipo de paralelización es la que realiza ML Engine en el caso de que realicemos entrenamientos con clusters predefinidos (sección 5.1.1). Además, otra ventaja de ML Engine es que toda la parte de coordinación en la actualización de parámetros se realiza de manera transparente al programador, de forma que éste pueda centrar sus esfuerzos en la parte de desarrollo del modelo.

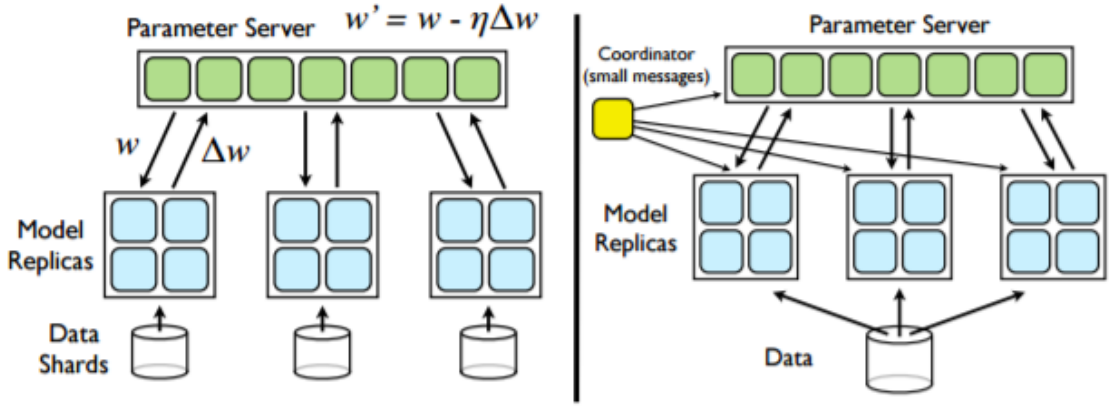


Figura 2.7: Paralelización (a nivel de datos) síncrona y asíncrona para un modelo realizado en TensorFlow

Fuente: Google Imágenes

Por otro lado, tenemos el caso de distribuir las distintas operaciones dentro del propio grafo, es decir, distintas máquinas se ocuparán de realizar distinas operaciones del grafo. A diferencia de la paralelización a nivel de datos, aquí solo tendríamos una instancia del grafo pero a cambio la implementación de este método es mucho más compleja.

2.2.3. TensorBoard

Se trata de un conjunto de herramientas que permiten visualizar grafos de cómputo y resultados de ejecuciones relativos a modelos entrenados en TensorFlow. Estas herramientas se encuentran dentro de la propia librería de TensorFlow y se utilizan por medio de una aplicación web (veáse figura 2.8). Para conocer el funcionamiento interno de TensorBoard me he basado en dos tutoriales de Tensorboard sobre el seguimiento de las métricas a lo largo del entrenamiento [36] y la visualización del grafo interactivo de cómputo [63]. Los datos que obtiene TensorBoard a partir de las ejecuciones de TensorFlow se realizan con *summary ops*, operaciones al igual que las sumas, multiplicaciones o funciones de activación que se aplican sobre tensores. La única diferencia es que estas operaciones, al ser ejecutadas como cualquier nodo del grafo de cómputo, producen información serializada que solo puede leer Tensorboard.

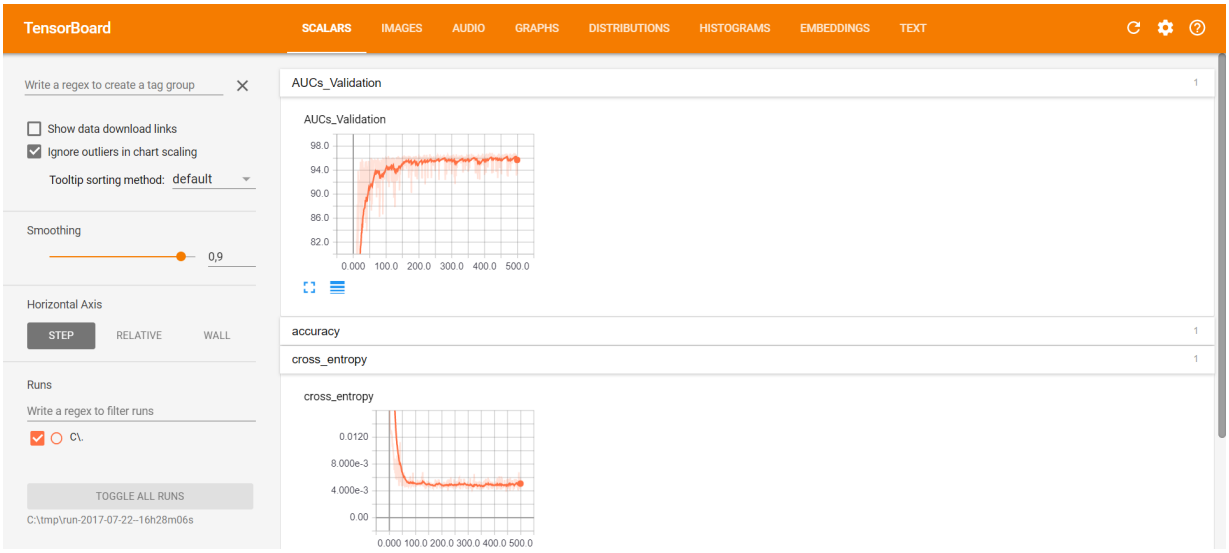


Figura 2.8: Captura de pantalla de TensorBoard

Fuente: Log de una ejecución del benchmark

Tenemos varios tipos de estas operaciones, por ejemplo *tf.summary.scalar*, *tf.summary.image* o *tf.summary.histogram*, aunque en nuestro caso nos interesa la primera de ellas, pues queremos guardar información sobre tensores de tipo numérico. Por ejemplo, supongamos que quisiéramos guardar la información sobre la métrica *accuracy* y la función de coste de nuestro modelo. Para ello escribiríamos las siguientes instrucciones dentro de nuestro código:

```
##### Parte de la definicion del grafo de computo #####

# ....

# accuracy y xentropy son dos tensores que ya hemos declarado
tf.summary.scalar('accuracy', accuracy)
tf.summary.scalar('xentropy', xentropy)


############### Parte de entrenamiento ##################
with tf.Session() as sess:
        # Unimos todas las summary ops y creamos el directorio
        # TENSORBOARD_LOG, donde guardaremos la informacion
        merged = tf.summary.merge_all()
        train_writer = tf.summary.FileWriter(TENSORBOARD_LOG,
        sess.graph)

        # Guardamos informacion cada 10 iteraciones
        for i in range(NUM_EPOCHS):
                if i % 10 == 0:
```

```
# Corremos el conjunto de summary ops
# para obtener la informacion serializada
# y a la vez entrenamos el modelo
summary, acc = sess.run([merged, accuracy],
        feed_dict=feed_dict(False))
# Escribimos en disco la informacion serializada
test_writer.add_summary(summary, i)
```

Vemos como el proceso consiste en definir el nombre y el valor de los tensores que queremos guardar, unirlos en un único nodo y correr este nodo durante el entrenamiento. Finalmente, iremos guardando los resultados en disco usando el objeto *FileWriter* que hemos definido previamente. Nótese como al instanciar este *writer*, pasamos también el grafo de cómputo de la sesión actual, para así poder interactuar posteriormente con él en TensorBoard. Tras ejecutar el código, se generará un archivo cuyo nombre contiene *tfevents*, que almacena todos los logs generados.

Como último paso, sólo nos queda abrir la interfaz de TensorBoard para el archivo que acabamos de guardar en disco. Para ello, ejecutamos el comando:

```
$ tensorboard --logdir = TFEVENTS_PATH
```

TFEVENTS_PATH es la ruta donde almacenamos uno o varios archivos tfevents (pues TensorBoard nos permite comparar logs relativos a varias ejecuciones). Tras ejecutar este comando, TensorBoard indicará que se ha abierto una conexión en http://localhost:6006. Accediendo a esa url, podemos ver la interfaz gráfica de TensorBoard. En la pestaña *scalars* se encuentran las gráficas con los valores de los tensores que hemos guardado, mientras que en la pestaña *graphs* se encuentra el grafo de cómputo interactivo.

2.2.4. Keras

Keras es una librería para Python diseñada para construir redes neuronales de forma rápida y modular en TensorFlow. Fue creada en el año 2015 por el ingeniero de Google François Chollet y a principios del año 2017 se convirtió en parte de la librería de TensorFlow [34]. No obstante, su vocación de ser una interfaz para diversas bibliotecas de deep learning se mantiene y actualmente posee también soporte para Theano o Microsoft CNTK. Para este trabajo se utiliza Keras corriendo sobre TensorFlow.

Destacar además que recientemente han surgido varias alternativas a Keras, aunque todas ellas se ejecutan exclusivamente sobre TensorFlow: Sonnet (creada por Google Deepmind), TF-Slim o TFLearn son algunos ejemplos.

2.3. Pandas

Se trata de una libería para Python con licencia BSD orientada al preprocesamiento y análisis de datos relacionales y series temporales. Ofrece estructuras de datos tales como *dataframes* (prácticamente idéntico al término homónimo en R) así como operaciones eficientes sobre estas estructuras (eliminación de *NaN* y *nulls*, joins, mutaciones de columnas, etc.). Esta librería es la elegida para realizar la parte de preprocesamiento y exploración de datos en Datalab (sección 3.2).

2.4. Scikit-learn

Librería de software libre (licencia BSD) para Python enfocada en el campo del aprendizaje automático. Posee una gran variedad de algoritmos de clasificación, regresión y reducción de dimensionalidad. No obstante, en este trabajo se ha utilizado esta librería en la parte de análisis de datos (sección 3.2), realizando tareas muy variadas: división de datasets en conjuntos de entrenamiento, validación y test, PCA, etc.

Es interesante destacar la excelente documentación que posee esta librería en su web, puesto que incluye ejemplos muy detallados sobre ramas muy diversas del aprendizaje automático [25].

3 Análisis y exploración de datos en la nube

Este capítulo se corresponde con la primera fase del proceso de desarrollo y análisis del modelo predictivo. La totalidad de esta fase se realizará en remoto, utilizando diversos servicios de Google Cloud Platform por medio del lenguaje Python. Esto nos permitirá evitar instalar Python en local, así como un gran número de librerías para este lenguaje, las cuales son necesarias para realizar primero la exploración y posteriormente un pequeño entrenamiento con los datos. Aunque los servicios que utilizaremos para almacenar los datos con los que alimentaremos el modelo (BigQuery y Storage) están preparados para cantidades de varios terabytes, tenemos que remarcar que en esta fase se debe trabajar con pequeñas muestras de datos. Se han realizado varias pruebas y se ha comprobado que si trabajamos con muestras superiores a 300.000 filas, Datalab se comporta de forma inestable, por lo que el análisis es bastante insatisfactorio. No obstante, esta limitación no aplica al dataset CCF que estamos utilizando de ejemplo, el cual posee aproximadamente 285.000 filas, evitando en este caso trabajar con muestras.

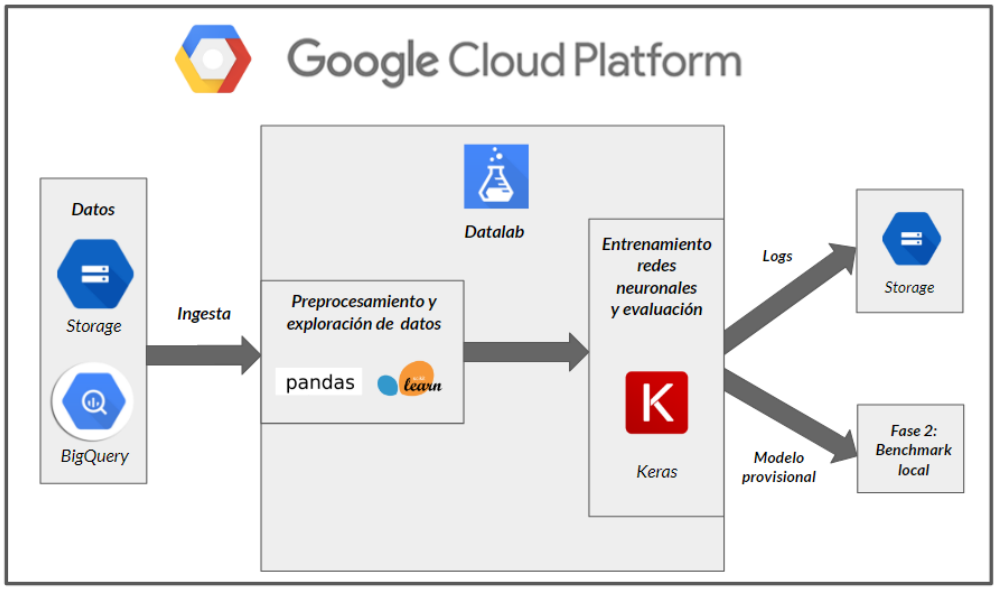


Figura 3.1: Etapas en las que se divide la fase de análisis y exploración

Fuente: Elaboración propia

Esta fase se divide en otras cuatro (veáse figura 3.1): en primer lugar se procede a realizar la importación de los datos necesarios para crear nuestro modelo predictivo. Este

proceso se denomina *ingesta de datos* (*data ingestion*) y en nuestro caso mostraremos dos maneras distintas de realizarlo, utilizando dos servicios de Google Cloud: BigQuery y Storage. Como hemos mencionado previamente, es necesario trabajar con muestras de los datos alojados en estos dos servicios. Para ello, en el caso de BigQuery simplemente añadiremos la claúsula *LIMIT* en las queries a realizar, mientras que para Storage nos cercioraremos previamente de que los archivos csv que depositemos en este servicio no superen las 300.000 filas.

Tras completar la ingesta y aprovechando las librerías presentes en Datalab, procederemos a la parte de preprocesamiento y exploración de los datos importados, comenzando de esta manera el análisis de los mismos. Tras este paso, utilizaremos los datos ya preprocesados y explorados con la librería Keras para realizar pequeños entrenamientos utilizando redes neuronales.

Por último, cuando hayamos finalizado con los entrenamientos y nuestro modelo sea capaz de realizar predicciones (por ejemplo, distinguiendo transferencias fraudulentas y normales en el caso del dataset CCF), pasaremos a almacenar logs en Storage para los resultados obtenidos, al igual que información adicional relativa al modelo utilizado, como por ejemplo los hiperparámetros de una red neuronal (número de capas ocultas, número de neuronas en las capas, funciones de activación, etc.). Estos valores nos serán de ayuda en la fase posterior con el benchmark local, donde trataremos de mejorar los resultados obtenidos en esta primera fase.

En este capítulo se han desarrollado dos notebooks de Datalab (*Keras-GCS.ipynb* y *Keras-BQ.ipynb*) que se encuentran en el repositorio Github del proyecto [55] y que servirán de guía para explicar en detalle las distintas subfases que conforman esta primera parte del proyecto. Además, en el caso de que se quiera replicar esta fase en otras cuentas de Google Cloud, se ha incluido en el apéndice 2 una serie de instrucciones para instalar y configurar en la nube una instancia de Datalab.

Para finalizar este capítulo se explicarán los costes derivados de utilizar Datalab, aportando además una estimación total de los mismos en el caso de realizar esta fase con otros datasets.

3.1. Ingesta de datos

La fuerte integración de Datalab con otras herramientas de Google Cloud permitirá que este primer paso sea bastante sencillo, evitando tener que usar servicios de autenticación de Google Cloud como *OAuth 2.0* o las *cuentas de servicio* [3]. Aunque en el resto de etapas de esta fase de exploración los dos notebooks que he preparado muestren funcionalidades bastante similares, en *Keras-GCS.ipynb* se realiza la ingesta de datos por medio de Cloud Storage mientras que en *Keras-BQ.ipynb* se hace con BigQuery.

3.1.1. Google Cloud Storage

El notebook *Keras-GCS.ipynb* realiza una ingesta del dataset CCF desde Storage. Para poder realizar esta operación, necesitamos haber depositado previamente el archivo csv del dataset en una carpeta o *bucket* de esa herramienta. Esto se puede realizar a través de Google Cloud Console o usando el comando *gsutil cp* del SDK de Google Cloud. En este caso se ha usado la primera opción dado que es más comoda, aunque la segunda opción es preferible si estamos trabajando con scripts Bash, tal y como ocurre en el capítulo 5. Usando la sintaxis del notebook de ejemplo *Importing+and+Exporting+Data.ipynb* [52], importaremos nuestros datos desde Storage y los transformaremos en un dataframe de Pandas (podemos ver un ejemplo de esto en la figura 3.2).

```
# Leemos el csv desde Cloud Storage
%storage read --object gs://analiticauniversal/DatasetsTF/creditcard.csv --variable creditcards

# Guardamos el csv en un dataframe de Pandas
df = pd.read_csv(StringIO(creditcards))
```

Figura 3.2: Ingesta de datos desde Cloud Storage

Fuente: Keras-GCS.ipynb [55]

Nótese como *gs://analiticauniversal/DatasetsTF/creditcard.csv* es la ruta de Storage donde se encuentra el archivo csv de nuestro dataset, mientras que *pd* es el alias que hemos asignado al paquete Pandas en Python tras haber ejecutado previamente la instrucción *import pandas as pd*.

3.1.2. BigQuery

En este caso usaremos uno de los datasets públicos ofrecidos por BigQuery [20] y que consiste en una tabla que recoge todos los viajes en taxi realizados en la ciudad de Chicago desde el año 2013 hasta la actualidad (agosto de 2017), siendo ésta actualizada de forma mensual [19]. A fecha de 19 de julio de 2017, la tabla ocupa 34 GB y se compone de 99.761.096 filas (es decir, el número total de viajes en taxi registrados desde 2013) y 23 columnas. Para realizar la consulta a esta tabla y trabajar con los datos en un notebook de Datalab, utilizaremos el paquete *google.datalab.bigquery*, el cual se encuentra ya instalado por defecto. Tal y como podemos ver en los notebooks de ejemplo *BigQuery+Commands.ipynb* y *BigQuery+Magic+Commands+and+DML.ipynb* [52], existen dos maneras de usar el paquete de BigQuery para realizar la ingesta. La primera manera es escribiendo la query como un string, instanciar un objeto para guardar este string, ejecutar la query y volcarla en un *dataframe* de Pandas (figura 3.3).

En nuestro notebook he probado esta manera con una query que devolvía viajes realizados cada 15 minutos, para así luego preprocesar esa información con Pandas y mostrar en un gráfico los viajes realizados cada hora en octubre de 2016.

```
import google.datalab.bigquery as bq

# Aprovechando que los tiempos de los viajes se guardan en intervalos de 15 minutos,
# realizamos una query con el número de viajes en cada uno de esos intervalos

# La query se pasa como un string
query_string = 'SELECT trip_start_timestamp AS timestamp, COUNT(trip_start_timestamp) AS num_trips FROM `b
igquery-public-data.chicago_taxi_trips.taxi_trips` GROUP BY timestamp ORDER BY timestamp ASC'
query = bq.Query(query_string)

# Pasamos a un dataframe de Pandas (la librería se importa automáticamente) el resultado de la query
df = query.execute(output_options=bq.QueryOutput.dataframe()).result()
```

Figura 3.3: Ingesta de datos desde BigQuery pasando la query como string

Fuente: Keras-BQ.ipynb [55]

La segunda manera de realizar la ingesta en BigQuery es aún más sencilla, y se realiza por medio del operador %% (veáse figura 3.4).

```
%%bq query -n taxi_queries
SELECT trip_seconds, trip_miles, pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude, f
are
FROM `bigquery-public-data.chicago_taxi_trips.taxi_trips`
WHERE trip_miles IS NOT NULL AND
      trip_seconds IS NOT NULL AND
      pickup_latitude IS NOT NULL AND
      pickup_longitude IS NOT NULL AND
      dropoff_latitude IS NOT NULL AND
      dropoff_longitude IS NOT NULL AND
      fare IS NOT NULL
LIMIT 300000
```

```
# Ejecutamos la query y la volcamos en un Pandas dataframe
# Gracias a BigQuery, el tiempo de ejecución de la query es menor al minuto a pesar de ser una tabla de 10
0 millones de filas.
df = taxi_queries.execute(output_options=bq.QueryOutput.dataframe()).result()
```

Figura 3.4: Ingesta de datos desde BigQuery usando el operador %%

Fuente: Keras-BQ.ipynb [55]

En este caso realizaremos una consulta limitada a 300.000 filas y 7 columnas. En la tabla 3.1 se encuentra una descripción más detallada de las columnas que hemos escogido. Es necesario destacar que las columnas relativas a latitudes y longitudes se refieren a centros geográficos de los barrios donde se encuentran los taxis en el momento de recoger o dejar a los clientes.

También queremos remarcar en este punto que la velocidad de BigQuery para obtener queries sobre el dataset de los viajes en taxi es realmente alta, pudiendo ejecutar queries de cierta complejidad en menos de un minuto. Este hecho es bastante positivo, pues como hemos comentado previamente, este dataset posee un tamaño realmente grande.

3.2. Preprocesamiento y exploración de datos con Pandas y Scikit-Learn

En esta parte tenemos bastante libertad para realizar la exploración de datos, aunque nos centraremos en tres herramientas únicamente: Pandas, Scikit-Learn y Matplotlib.

Aprovecharemos por un lado la potente funcionalidad de Pandas con los dataframes obtenidos en la etapa anterior para realizar una serie de preprocesamientos (por ejemplo normalizando datos o separando datos en dos clases disjuntas) que nos permitan conocer la distribución de los datos (histogramas, diagramas de dispersión, etc...), así como algunos estadísticos básicos. Con Scikit-learn aplicaremos algoritmos de reducción de dimensionalidad (t-SNE y PCA) y dividiremos los datos en tres conjuntos disjuntos de cara a la etapa posterior de entrenamientos con Keras. Por último, Matplotlib, librería de Python destinada a generar gráficos, nos permitirá pintar por pantalla los resultados que hemos obtenido al usar las dos librerías anteriores.

Para tener algo de contexto antes de exponer la exploración de datos que he preparado en los dos notebooks de ejemplo, explicaremos brevemente los problemas asociados a los datasets que manejamos en esos notebooks. En *Keras-GCS.ipynb* se trabaja con el dataset CCF, por lo que abarcaremos el problema de clasificación binario de discernir entre transferencias fraudulentas y normales. Por otro lado, en *Keras-BQ.ipynb* intentaremos predecir la tarifa de un viaje en taxi en la ciudad de Chicago a partir de las 6 primeras columnas indicadas en la tabla 3.1. Nótese como la tarifa de un viaje es un valor real positivo y continuo, por lo que este problema es de regresión.

Tabla 3.1: Columnas utilizadas en el dataset de viajes en taxi en Chicago

Fuente: BigQuery

Nombre columna	**Tipo**	**Descripción**
trip_seconds	Integer	Segundos que dura el viaje
trip_miles	Float	Distancia del viaje en millas
pickup_latitude	Float	Latitud del punto de recogida
pickup_longitude	Float	Longitud del punto de recogida
dropoff_latitude	Float	Latitud del punto de destino
dropoff_longitude	Float	Longitud del punto de destino
fare	Float	Tarifa del viaje (en dólares)

3.2.1. CCF dataset

Debido a que 28 de las 30 variables de este dataset están anonimizadas y no sabemos nada sobre ellas [17], he pintado histogramas con las otras dos variables que conocemos: el tiempo y las cantidades transferidas para cada uno de los dos tipos de transferencias del dataset. En primer lugar obtenemos que solo el 0.17 % de las transferencias son de tipo fraudulento, pero el dato más relevante obtenido en este análisis es que aunque las transferencias fraudulentas no se distribuyen de forma homogénea en el tiempo, en torno a 200 de las 492 transferencias de este tipo son menores a 3 euros. Este trabajo se ha realizado principalmente con Pandas, ayudándonos de Matplotlib para generar varios gráficos en una misma ventana, lo cual ayuda bastante a analizar datos y extraer conclusiones.

También he aplicado en este notebook (Keras-GCS.ipynb) un algoritmo de reducción de dimensionalidad: Principal Component Analysis (PCA). Podemos ver en la figura 3.5 el resultado de aplicar este algoritmo a las 30 variables del dataset para quedarnos con solo 2. Las conclusiones que obtenemos es que no podemos distinguir de forma clara las transferencias fraudulentas del resto. Esto puede ser debido a que necesitamos pedir más de 2 o 3 variables resultantes al PCA, aunque esto impediría poder visualizar el resultado en el plano.

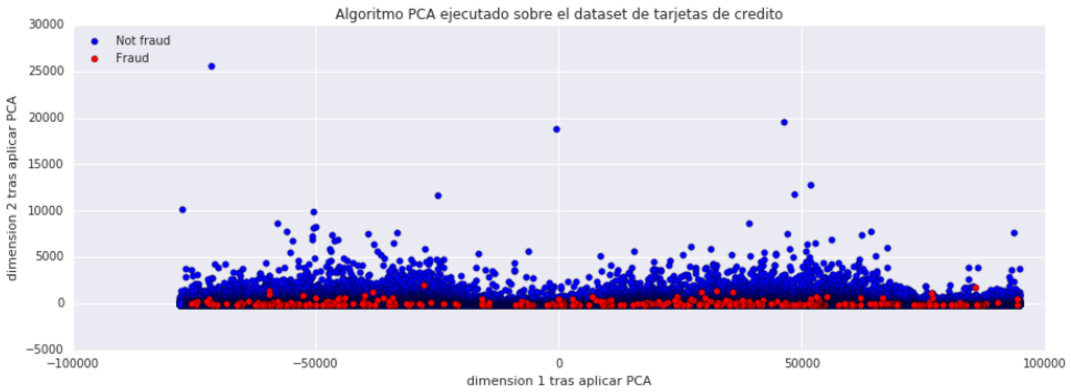


Figura 3.5: Resultado de un PCA aplicado al dataset CCF

Fuente: Keras-GCS.ipynb [55]

3.2.2. Chicago taxi trips dataset

Para este dataset se ha hecho hincapié en mostrar distintos gráficos (de dispersión, de barras, histogramas, etc..) usando la función *plot* de Pandas, que a su vez llama a la función homónima de Matplotlib. Los resultados obtenidos nos han enseñado como existe una cierta correlación entre la variable de millas recorridas en el viaje y la tarifa

del mismo (coincidiendo con la lógica de los taxímetros), o que la gran mayoría de viajes poseen una tarifa entre 4 y 8 dólares (veáse figura 3.6). Esta última observación ha dado pie a que apliquemos (gracias a Scikit-Learn) el algoritmo t-SNE sobre este tipo de viajes y el resto. Este algoritmo reduce las 6 variables utilizadas en el dataset a únicamente 2, por lo que podemos pintar puntos en el plano para cada uno de los viajes en taxi realizados. En *Keras-BQ.ipynb* podemos comprobar como existe un cierto patrón entre los viajes que han costado entre 4 y 8 dólares y el resto.

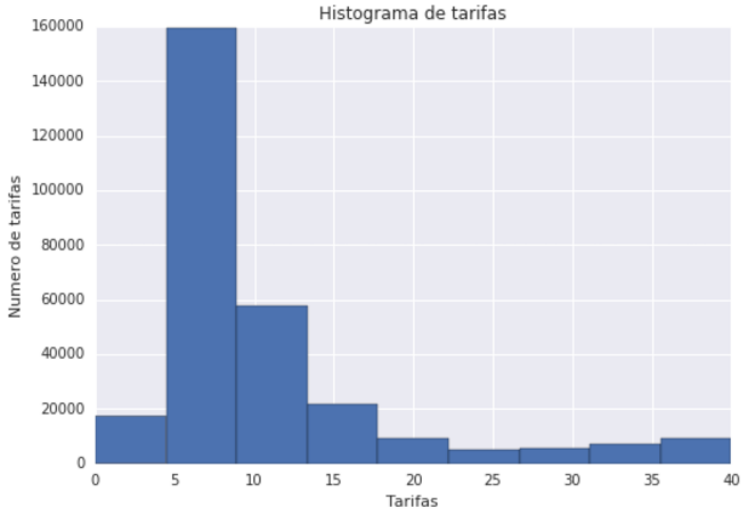


Figura 3.6: Histograma de tarifas de viajes en taxi en la ciudad de Chicago

Fuente: Keras-BQ.ipynb [55]

3.3. Entrenamiento de redes neuronales con Keras

La penúltima fase del proceso de análisis de datos consiste en entrenar redes neuronales utilizando la librería de Keras, la cual aporta un nivel de abstracción bastante alto respecto a TensorFlow. Para poder realizar esta fase, se ha trabajado principalmente con la documentación de Keras [24].

Si queremos entrenar redes neuronales en Keras necesitamos que los datos de cada uno de los dos notebooks desarrollados estén representados en arrays de Numpy, una estructura de datos para representar vectores y matrices de forma eficiente. Esta transformación se realiza por medio de la función *as_matrix()* de Pandas.

El siguiente paso es dividir (usando la función *train_test_split()* de Scikit-Learn) los arrays de Numpy obtenidos en un conjunto de entrenamiento, uno de validación y otro de test[1]. El primero de ellos será el utilizado para entrenar la red neuronal, mientras que el segundo será aquel que utilizaremos para evaluar determinadas métricas al finalizar cada *epoch* o iteración de la red neuronal. El conjunto de test se utiliza únicamente al final del entrenamiento, ya sea tras acabar después de realizar las iteraciones indicadas o debido a una finalización prematura causado por el mecanismo de pronta parada de Keras (el cual es un *callback*, como los que se ven en 3.4), capaz de detener el entrenamiento si la función de coste no disminuye (al evaluarse sobre los datos de validación) un determinado valor a lo largo de de un número determinado de iteraciones. Tanto ese valor como el número de iteraciones se pueden pasar como parámetros a Keras.

Debemos destacar que el conjunto de test nos permite tener una evaluación precisa de las predicciones del modelo desarrollado, pues ese conjunto no interviene en ningún momento de la fase de entrenamiento y por tanto no la puede sesgar de ninguna manera.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20)                620
_________________________________________________________________
batch_normalization_1 (Batch (None, 20)                80
_________________________________________________________________
activation_1 (Activation)    (None, 20)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 20)                0
_________________________________________________________________
dense_2 (Dense)              (None, 15)                315
_________________________________________________________________
batch_normalization_2 (Batch (None, 15)                60
_________________________________________________________________
activation_2 (Activation)    (None, 15)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 15)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 32
=================================================================
Total params: 1,107.0
Trainable params: 1,037.0
Non-trainable params: 70.0
_________________________________________________________________
```

Figura 3.7: Ejemplo de red neuronal entrenada en Keras para el dataset CCF

Fuente: Archivo results.txt en datalab/resultados/log2.zip [55]

Ahora bien, para realizar el entrenamiento también necesitamos comunicar a Keras los hiperparámetros y la estructura de la red neuronal que vamos a construir. Esto se realiza

[1]En el caso del dataset CCF, dado que la clase de transferencias fraudulentas está muy desbalanceada respecto a la otra, he realizado previamente a este paso un proceso de undersampling para así obtener un dataset balanceado y evitar encontrarnos con resultados sesgados tras el entrenamiento.

en primer lugar llamando a la función *Sequential()*, añadiendo las instrucciones necesarias en el mismo orden que las operaciones que queramos aplicar. Supongamos a partir de ahora que *model* es una instancia de *Sequential()*. Usando *model.add(Dense(N))* añadiremos al modelo una capa oculta de N neuronas, mientras que *model.add(Activation('elu'))* aplicará la función de activación ELU sobre la capa oculta que se añadió previamente. Con una sintaxis similar se pueden añadir otras técnicas de optimización tales como *dropout* o *batch normalization*.

Es necesario recordar que la última capa tendrá tantas neuronas como clases tengamos que predecir (2 en el caso del dataset CCF), aplicando sobre ellas una función *softmax*, de manera que las neuronas de salida tomen valores entre 0 y 1. No obstante, en el caso de que tengamos un problema de regresión como el de los viajes en taxi (*Keras-BQ.ipynb*) simplemente tendremos una sola neurona en la capa de salida, sin función de activación aplicada sobre ella, dado que el resultado puede ser cualquier valor real.

El siguiente paso es indicar en *model.compile()* la función de coste que queremos minimizar, el optimizador a usar en el algoritmo de descenso de gradiente y las métricas con las que vamos a evaluar los tres conjuntos de datos que tenemos. Nótese como para el problema de las transferencias utilizamos *cross entropy* (veáse ecuación 4.8) como función de coste, mientras que en el problema de predicción de tarifas de los viajes en taxi se ha escogido la función de error cuadrático medio:

$$L(X, W, Y) = \frac{1}{N} \sum_{i=1}^{N} \|f(\vec{x_i}, W) - y_i\|_2^2 \tag{3.1}$$

$$X = \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_N} \end{pmatrix} \tag{3.2}$$

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \tag{3.3}$$

Donde X es el vector con cada una de las observaciones (cada una de ellas es un vector de dimensión 6, que es el número de variables que consideramos en el problema), W los pesos de la red neuronal y $f(\vec{x_i}, W)$ la predicción de nuestra red neuronal para la tarifa del viaje i-ésimo en taxi, cuyo valor real es y_i. Para este problema en concreto el número de observaciones o filas del dataset, N, es 300.000.

Respecto a las métricas de evaluación en las que nos apoyamos, en el dataset CCF se utiliza *accuracy* y AUC. Nótese que no podemos calcular el AUC directamente con Keras, por lo que usamos una función auxiliar de Scikit-Learn. En cambio, para el otro dataset se utiliza el error medio absoluto (MAE) y el error porcentual medio absoluto (MAPE):

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|f(\vec{x_i}, W) - y_i| \tag{3.4}$$

$$MAPE = \frac{100}{N}\sum_{i=1}^{N}\frac{|f(\vec{x_i}, W) - y_i|}{y_i} \tag{3.5}$$

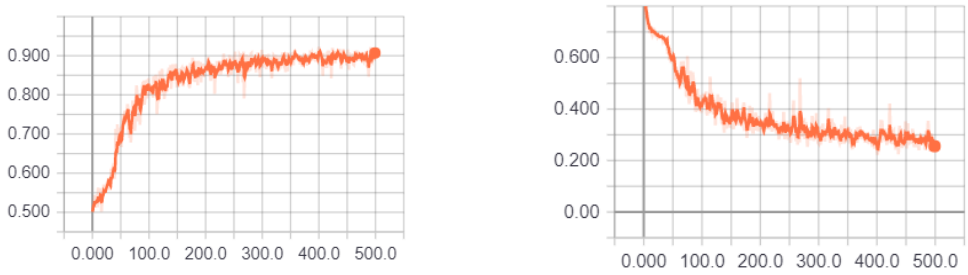
Ya solo queda ejecutar el entrenamiento en Keras, que se realiza llamando a la función *fit()*, a la cual tenemos que pasar los datos de entrenamiento y validación, el número de iteraciones a realizar y el tamaño del batch (*batch size*). En la figura 3.7 mostramos un ejemplo de red neuronal entrenada por Keras. Esta imagen se puede visualizar usando el comando *model.summary()*, justo después de haber añadido las capas a nuestro modelo. En el ejemplo de la figura la red se compone de dos capas ocultas con 20 y 15 neuronas respectivamente, así como 1.037 pesos a ajustar durante el entrenamiento.

3.4. Almacenamiento de modelos y logs de ejecución

Esta fase solo se ha desarrollado para el dataset CCF en *Keras-GCS.ipynb* y consiste en guardar en Cloud Storage información relacionada con el modelo entrenado en Keras en la etapa anterior, así como resultados del propio entrenamiento. Con toda esta información guardada en la nube se intenta conseguir que una o varias personas puedan extraer conclusiones a partir de una serie de entrenamientos realizados con distintas topologías de redes neuronales.

En primer lugar, justo antes de realizar el entrenamiento con Keras, se crea la jerarquía de directorios para los logs tanto en la propia máquina que está ejecutando Datalab, como en Cloud Storage. Para realizar todas estas tareas de manejo de ficheros entre estos directorios, se ha usado por comodidad el módulo *gfile* de TensorFlow, que permite interactuar con sistemas de ficheros como HDFS o Cloud Storage [62].

El siguiente paso es proporcionar a Keras (cuando se llama a *fit()*) una serie de funciones a ejecutar en cada iteración del entrenamiento. Estas funciones se denominan *callbacks* y van a proporcionarnos por un lado información en un archivo *training.log* sobre las métricas y la función de coste a lo largo de los *epochs* para los datos de entrenamiento y validación. También se generará un archivo que podremos abrir con Tensorboard para ver el grafo interactivo de cómputo generado en TensorFlow, al igual



(a) Predicciones correctas (%)

(b) Función de coste

Figura 3.8: Logs de Tensorboard para un entrenamiento de Keras

Fuente: Tensorboard (log2.zip) [55]

que gráficas con los valores de la función de coste y las métricas a lo largo de sucesivas iteraciones (veáse figura 3.8). Por último, se genera un archivo *results.txt* que contiene la siguiente información:

- Un JSON generado por Keras para identificar el modelo usado para el entrenamiento. En caso de querer volver reusar un modelo, podemos hacerlo a partir del JSON guardado y la función *model_from_json()*. Un ejemplo de uno de estos JSON (ya formateado) se puede encontrar en el apéndice 6.
- Hiperparámetros utilizados para el entrenamiento (*batch size*, *epochs* y *tasa de dropout*).
- Un resumen del modelo utilizado, resultado de la llamada a *model.summary()*.
- La función de coste y el porcentaje de instancias correctamente clasificadas (*accuracy*) sobre el conjunto de test.
- El valor de la métrica *AUC*, bastante útil cuando entrenamos datasets desbalanceados.

Para encontrar información más detallada sobre los logs generados con Keras, incluyendo un análisis de varios entrenamientos realizados sobre el dataset CCF, es conveniente consultar el apéndice 3.

3.5. Precio

Aunque no existe un precio por utilizar Datalab como herramienta, existen costes derivados de ella [12]. En este trabajo son los siguientes: costes de máquinas virtuales, de almacenamiento y de BigQuery.

El primero de ellos se produce al ejecutar por horas o meses un determinado tipo de máquina virtual. Para este trabajo se ha utilizado la máquina por defecto si no se pasa el argumento correspondiente al comando *datalab create*. Esta máquina se denomina *n1-standard-1* y posee una CPU con 3.75 GB de memoria RAM. Cabe destacar que para este trabajo ha sido más que suficiente esta máquina en concreto, aunque existen otras tantas que pueden consultarse en la tabla 3.2 junto a sus precios por hora (a fecha de julio de 2017). Podríamos considerar usar una máquina virtual con GPU para reducir el tiempo de entrenamiento (sobre todo cuando utilicemos redes neuronales bastante profundas), pues comparando con los precios de 5.1.3, el coste es superior si utilizamos GPUs con una de las máquinas de ML Engine. Sin embargo, la principal diferencia es que ML Engine utiliza Tensorflow optimizado para GPUs, a diferencia de Datalab, lo cual evidencia que no hay grandes ventajas en utilizar Datalab con máquinas que utilizan GPUs.

Tabla 3.2: Tabla de precios para máquinas de Compute Engine localizadas en Bélgica (julio de 2017)

Fuente: Web de costes de Google Compute Engine [15]

Nombre máquina	**Número de CPUs**	**Memoria**	**Precio por hora**
n1-standard-1	1	3.75 GB	$0.0110
n1-standard-2	2	7.5 GB	$0.0220
n1-highmem-4	4	26 GB	$0.0550
n1-standard-1 + GPU Tesla K80	1	1 GB + 12 GB GPU	$0.781

Además, aumentar el número de cores tampoco ayuda a reducir el tiempo de ejecución, pues cada uno de los notebooks utiliza un solo hilo de ejecución [10]. No obstante, las máquinas con alta capacidad de memoria podrían ser interesantes para el caso de que queramos cargar en memoria datasets algo más grandes a los que hemos utilizado.

Por otro lado está el coste de almacenamiento, que se desglosa a su vez en el almacenamiento de la propia máquina virtual y en Cloud Storage. En el primer caso los discos duros tradicionales tienen un coste de $0.040 GB/mes, mientras que el precio para SSD es de $0.170 cada GB al mes [15]. Recordemos que en las máquinas sobre las que se ejecuta Datalab únicamente precisamos guardar los notebooks para realizar la fase de análisis, por lo que aún contando con un almacenamiento de 10 GB, el precio mensual sería inferior a cincuenta centavos. Para el caso de Cloud Storage, tenemos 5 GB de

almacenamiento y 15.000 operaciones de modificación, borrado, etc. [14] algo más que suficiente para almacenar y tratar archivos csv de tamaño incluso hasta mucho mayor que el dataset CCF.

Otro de los costes que se nos aplicaría en este trabajo sería el del uso de BigQuery. En este caso es gratis tanto la carga de datos en tablas como 10 GB de almacenamiento mensuales[2]. También se dispone de forma gratuita lanzamientos de queries hasta llegar a un 1 TB de datos procesados en total [13]. Para poder ilustrar esto, comentar que a través de la aplicación web de BigQuery se puede comprobar que la segunda query realizada en el notebook *Keras-BQ.ipynb* procesa 4.71 GB, lo cual nos da margen para trabajar con tablas de gran tamaño.

Es necesario remarcar que todo el proceso que hemos realizado de análisis y exploración de datos no es algo que se dilate demasiado en el tiempo y que precise de la colaboración de muchas personas (provocando por lo tanto que no haga falta crear un gran número de instancias de Datalab), por lo que podemos pensar que el precio resultante de esta fase es bastante razonable.

Para dar un ejemplo, suponiendo que un empleado utiliza la máquina virtual por defecto utilizada en este capítulo (*n1-standard-1*) durante una jornada laboral (8 horas) a lo largo de un mes (160 horas laborables) y que el coste conjunto de almacenamiento y BigQuery está entre los 3 o 4 dólares mensuales (utilizando por ejemplo 10 GB de almacenamiento para la máquina virtual y 25 GB en Storage, más 1,5 TB de procesamiento de queries en BigQuery), el coste mensual ascendería a unos 10 dólares, una cantidad verdaderamente asequible.

[2]Para el caso de datasets públicos de BigQuery, como el de los viajes en taxi en Chicago, esta norma no aplica.

4 Benchmark local de entrenamiento para redes neuronales

El capítulo actual trata sobre la fase central del trabajo, donde he desarrollado un benchmark local que permite experimentar y entrenar con distintas topologías de redes neuronales. Este benchmark ha sido desarrollado en TensorFlow (usando la versión 1.1.0 de la API para Python) y está compuesto de una serie de utilidades que nos permiten validar y analizar en detalle diversos modelos predictivos (veáse figura 4.1). Todo el código del benchmark, así como algunos resultados de ejecución del mismo se puedes encontrar en el repositorio Github del proyeto [55].

El objetivo de esta fase es obtener un modelo óptimo a partir de los modelos provisionales obtenidos con Keras en la sección 3.3. Para ello se va probando con distintos hiperparámetros de redes neuronales: número de capas ocultas y neuronas en cada capa, número de iteraciones a realizar en el entrenamiento, tipo de optimizadores a aplicar, etc.

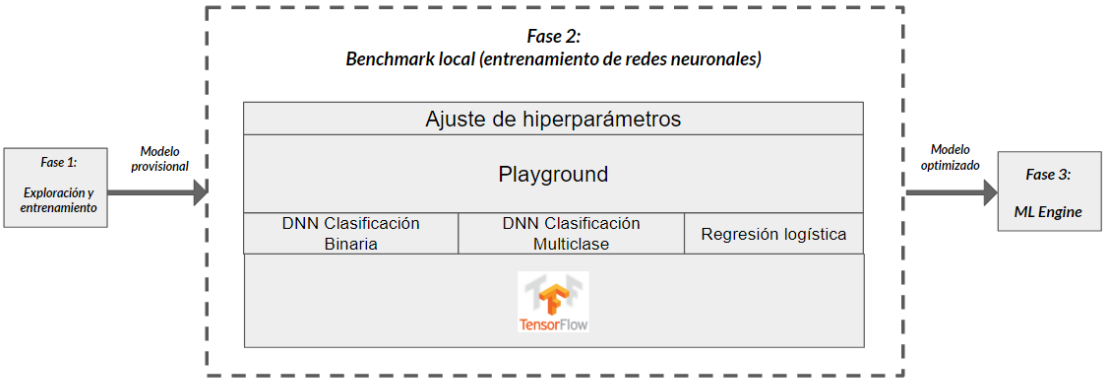


Figura 4.1: Conjunto de utilidades desarrolladas para el benchmark

Fuente: Elaboración propia

Esta búsqueda no será en absoluto desinformada, pues recordemos que ya hemos realizado un análisis exploratorio del dataset y conocemos como están distribuidos los datos y si existe algún tipo de particularidad en ellos.

El output de está fase corresponde con una serie de logs detallados que nos permiten evaluar el rendimiento de distintas redes neuronales sobre el dataset que estamos considerando. Además, estos logs contienen el conjunto de hiperparámetros necesarios para replicar el modelo entrenado y utilizarlo en la siguiente fase con ML Engine, donde podremos realizar entrenamientos con datasets de tamaño aún mayor a los utilizados en esta fase. En el caso de que se desee ver un detallado análisis de los resultados obtenidos a partir de las herramientas disponibles en el benchmark, se puede consultar el apéndice 4.

Para poder conocer en detalle los conceptos e ideas plasmados en el benchmark, realizaremos en primer lugar una introducción a las redes neuronales, haciendo hincapié en el conjunto de hiperparámetros que podemos utilizar en las distintas herramientas del benchmark.

4.1. Introducción a las redes neuronales

Para realizar esta sección sobre redes neuronales, nos hemos basado en el capítulo 10 del libro de Aurelien Geron [70], el curso CS231n de Stanford [18] (en particular el módulo 1) y apuntes de la asignatura de Geometría Computacional.

El primer modelo de redes neuronales se remonta a 1943, cuando el neurofisiólogo Warren McCulloch y el matemático Walter Pitts mostraron como neuronas artificiales podían construir cualquier tipo de proposición lógica. El siguiente modelo fue el perceptrón, creado por Frank Rosenblatt en 1957 y que podemos ver en la figura 4.2.

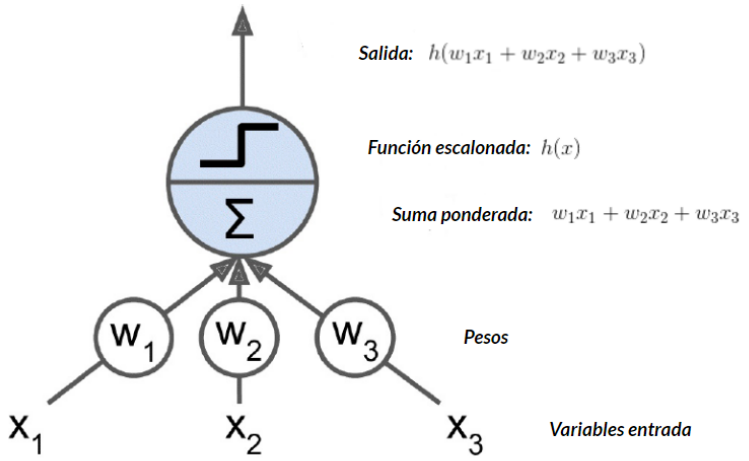


Figura 4.2: Perceptrón de Rosenblatt

Fuente: Hands-On Machine Learning with Scikit-Learn and TensorFlow [70]

Dada una observación $\vec{x} = (x_1, \ldots, x_k)$ con k variables de tipo numérico, primero se computa una suma ponderada de todas ellas: $\sum_{j=1}^{k} w_j x_j$ para posteriormente aplicar una función escalonada[1]: $h(\sum_{j=1}^{k} w_j x_j)$, normalmente una función de heaviside o una función signo:

$$heaviside(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

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

La salida del Perceptrón nos permite realizar predicciones binarias: aquellas observaciones cuya predicción es 1 y 0 se denominan *ejemplos positivos* y *negativos* respectivamente. Dado que el tipo de aprendizaje que se realiza en este trabajo es de tipo supervisado, dispondremos de un dataset con N observaciones (apiladas en una matriz X) y un vector de etiquetas denotado como T, que indican a que tipo de clase pertenecen las distintas observaciones:

$$X = \begin{pmatrix} \overrightarrow{x_1} \\ \overrightarrow{x_2} \\ \vdots \\ \overrightarrow{x_N} \end{pmatrix} \tag{4.1}$$

$$T = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{pmatrix} \tag{4.2}$$

Donde $t_i \in \{0, 1\}$ es decir, cada una de las dos clases que estamos considerando. Por ejemplo, en el dataset CCF, tenemos un problema de clasificación binario con dos clases: las transferencias de tipo fraudulento ($t_i = 1$) y las normales ($t_i = 0$). Además, debemos recordar que todas y cada una de las observaciones tienen k variables:
$\vec{x_i} = (x_{i1}, \ldots, x_{ik})$.

El mecanismo de aprendizaje propuesto por Rosenblatt para ajustar los pesos de un Perceptrón consiste en aplicar la siguiente fórmula de manera iterativa sobre cada una

[1] Es interesante apuntar que si en este punto aplicamos una función sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$, en realidad estamos realizando una regresión logística.

de las observaciones:

$$w_j^{(t+1)} = w_j^{(t)} - \eta\big[t_i - h(\sum_{l=1}^{k} w_l x_{il})\big]x_{ij} \qquad \forall j \in \{1, 2, \ldots, k\} \tag{4.3}$$

Donde η es la tasa de aprendizaje (con un valor normalmente entre 0 y 1) y $h(\sum_{l=1}^{k} w_l x_{il}) \in \{0, 1\}$ la predicción del Perceptrón para $\vec{x_i}$. Esta regla se va aplicando sobre todas las observaciones $\vec{x_i}$ hasta lograr la convergencia de los pesos, esto es, hasta que se alcance una iteración $t+1$ tal que: $\forall j, w_j^{(t+1)} = w_j^{(t)}$.

4.1.1. Perceptrón Multicapa

El principal problema del Perceptrón radica en la dificultad para resolver problemas de clasificación más complejos, por lo que el concepto evolucionó en el llamado Perceptrón Multicapa, que consiste en concatenar una serie de Perceptrones a lo largo de una serie de capas (veáse figura 4.3).

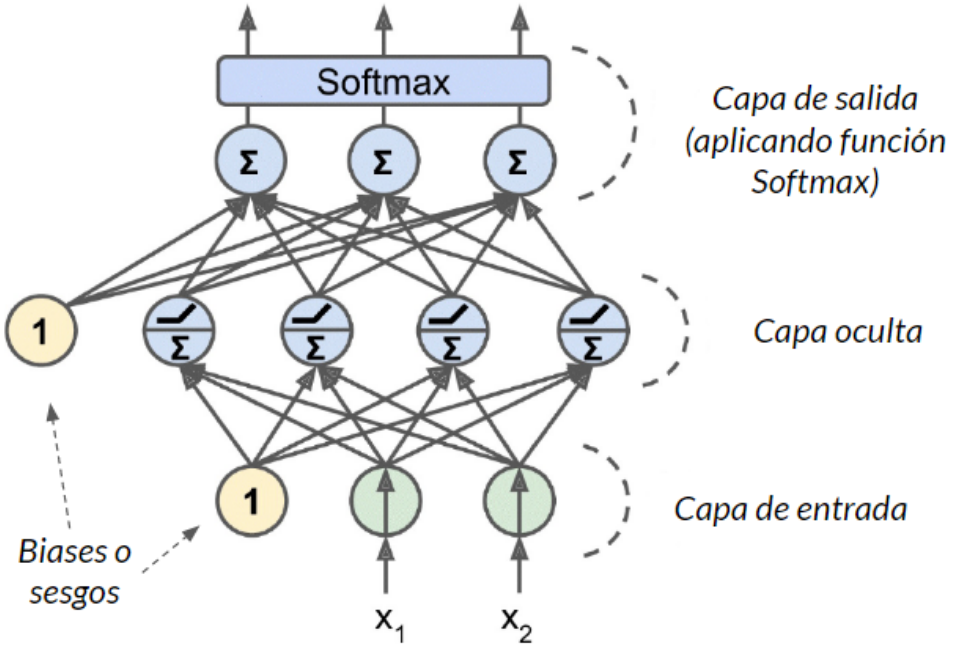


Figura 4.3: Perceptrón Multicapa

Fuente: Hands-On Machine Learning with Scikit-Learn and TensorFlow [70]

Cuando tenemos dos o más capas ocultas, diremos que se trata de una *red neuronal profunda* (*deep neuronal network*). Por otro lado, el hecho de que las neuronas de una capa estén conectadas con todas las de la siguiente lleva a llamar a estas capas como

totalmente conectadas (*fully-connected layers*). Finalmente, si la red neuronal no tiene ciclos, como en el caso de la figura, diremos que se trata de una red *feed-forward*. A continuación, presentaremos una serie de conceptos que introducen este tipo de redes.

En primer lugar vemos como en el Perceptrón Multicapa no existe una única salida, si no que ahora existen varias, lo que nos permitirá realizar predicciones para problemas de clasificación multiclase. Este tipo de problemas pueden verse como una generalización de los problemas binarios. A partir de ahora, supondremos que estamos tratando el caso de que las observaciones pertenezcan a m clases disjuntas, por lo que la matriz de etiquetas será:

$$T = \begin{pmatrix} \vec{t_1} \\ \vec{t_2} \\ \vdots \\ \vec{t_N} \end{pmatrix} \tag{4.4}$$

Donde si $\vec{x_i}$ pertenece a la clase j-ésima, entonces $\vec{t_i} = (t_{i1}, \ldots, t_{im})$ posee 0 en todas las componentes, a excepción de t_{ij}, que tendrá valor 1. Esta codificación se denomina *one-hot* y es interesante ver que para problemas de clasificación binarios podríamos tomar $m = 2$, con etiquetas de dos componentes, pero esto es equivalente a que las etiquetas tomen valores binarios o booleanos.

Otro de los cambios producidos es que las funciones escalón de los Perceptrones se sustituyen por las denominadas *funciones de activación*, las cuales son de tipo no lineal (algunos ejemplos son las funciones ReLU, elu, sigmoide o tangente hiperbólica). Nótese como en la capa de salida no hay función de activación, puesto que se aplica en este caso la función *softmax*[2]:

$$softmax(\vec{z}) = \left(\frac{e^{z_1}}{\sum_{k=1}^{m} e^{z_k}}, \ldots, \frac{e^{z_m}}{\sum_{k=1}^{m} e^{z_k}}\right) \tag{4.5}$$

Con $\vec{z} = (z_1, \ldots, z_m)$ los valores de las neuronas en la capa de salida. Nótese que tenemos tantas neuronas en la capa de salida, m, como clases distintas pueden ser predichas. Es fácil comprobar que las componentes del vector de la ecuación 4.5 toman valores entre 0 y 1, por lo que la componente j-ésima indicará la probabilidad otorgada por la red neuronal a una observación de pertenecer a la clase j-ésima. Por lo tanto, el índice de la componente del vector con mayor valor será la clase predicha por la red neuronal. Matemáticamente, esto se expresa con la siguiente ecuación:

$$P(t_{ij} = 1|\vec{x_i}) = softmax_j(\vec{z}) \tag{4.6}$$

[2]En 4.2.4 se ve como también se puede aplicar una función sigmoide a la única neurona de la capa de salida para resolver problemas de clasificación binarios, o incluso no aplicar ninguna función a la neurona de salida, en el caso de querer resolver problemas de regresión como el descrito en 3.3.

En la ecuación hemos denotado a $softmax_j(\vec{z})$ como la componente j-ésima del vector $softmax(\vec{z})$. También debemos señalar que, debido a como está definida la función *softmax*, se verifica que $\sum_{j=1}^{m} P(t_{ij} = 1|\vec{x_i}) = 1$ para cualquier observación $\vec{x_i}$.

Por otra parte (aunque esto no es algo exclusivo del Perceptrón Multicapa), en la figura 4.3 se muestra como también se puede añadir en cada capa una neurona extra, denominada sesgo o bias, cuyo valor (previamente a ser multiplicado por el peso correspondiente) siempre es 1.

Tras presentar toda esta serie de conceptos, dejaremos indicada la formulación matemática de un Perceptrón Multicapa para problemas de clasificación multiclase (m clases disjuntas) con N capas (incluyendo la capa de entrada y de salida) y d_l neuronas en la capa l-ésima:

$$a^{(l+1)} = h^{(l+1)}(a^{(l)}W^{(l)} + b^{(l)}) \qquad l \in \{1, 2, \ldots, L-1\} \tag{4.7}$$

siendo $a^{(l+1)}$ las *activaciones* de la red (con $a^{(1)} = \vec{x_i}$ para una observación $\vec{x_i}$ cualquiera) , $W^{(l)}$ y $b^{(l)}$ los *pesos* y *sesgos* de la capa l-ésima (matrices con dimensiones $d_l \times d_{l+1}$ y d_{l+1} respectivamente), $h^{(l+1)} : \mathbb{R}^{d_{l+1}} \to \mathbb{R}^{d_{l+1}}$ la *función de activación* para esa misma capa y L el número de capas totales (contando la de entrada y la de salida). Por lo tanto, la clase predicha por un Perceptrón Multicapa para una observación $\vec{x_i}$ es el índice de la componente con mayor valor en el vector de salida, $softmax(a^{(L)})$.

En este punto necesitamos definir una función de coste que permita conocer, con un valor real, como de precisas son las predicciones de una red neuronal concreta para un conjunto de observaciones dadas. Dada una matriz X con N observaciones, T la matriz de etiquetas de X y W un conjunto con los pesos $W^{(l)}$ y sesgos $b^{(l)}$, la función de coste de un Perceptrón Multicapa se denomina *cross-entropy* y es la siguiente:

$$L(X, T, W) = -\sum_{i=1}^{N}\sum_{j=1}^{m} t_{ij} log(y_j(\vec{x_i}, W)) \tag{4.8}$$

$$W = \{W^{(1)}, \ldots, W^{(L-1)}, b^{(1)}, \ldots, b^{(L-1)}\} \tag{4.9}$$

Donde t_{ij} es la etiqueta de la observación $\vec{x_i}$, valiendo 1 si ésta pertenece a la clase j-ésima y 0 en caso contrario. De la misma manera, $y_j(\vec{x_i}, W)$ es 1 si la predicción de la red neuronal (computada utilizando los pesos y biases pertenecientes a W) para la observación $\vec{x_i}$ es que ésta pertenece a la clase j-ésima y 0 en caso contrario. Esta función se minimiza por medio de un descenso de gradiente por lotes, utilizando el algoritmo

de *backpropagation*[3] para computar las derivadas parciales de las siguientes expresiones, que realizan las actualizaciones de pesos y sesgos:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \eta \frac{\partial L(X_{lote}, T_{lote}W)}{\partial W_{ij}^{(l)}} \tag{4.10}$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \eta \frac{\partial L(X_{lote}, T_{lote}W)}{\partial b_{i}^{(l)}} \tag{4.11}$$

η es la tasa de aprendizaje (normalmente con valor mayor que 0 y menor 1), $W_{ij}^{(l)}$ son los elementos de la matriz de pesos y $b_i^{(l)}$ las componentes de los biases en la capa l-ésima respectivamente. Nótese que hemos dividido las observaciones y sus correspondientes etiquetas (matrices X y T respectivamente) en *lotes* de un mismo tamaño prefijado, algo que se conoce como *batch size*. Hemos representado un lote cualquiera como X_{lote} y T_{lote} para denotar que se realizan tantas actualizaciones de pesos y biases como número de lotes tengamos (esto es, el resultado de la división entera entre N y el *batch size*). Cuando se hayan realizado las actualizaciones de parámetros tomando cada uno de los lotes, diremos que se habrá completado un *epoch* o iteración del algoritmo.

4.1.2. Hiperparámetros

En la sección anterior hemos visto como las redes neuronales, y en particular los Perceptrones Multicapa, poseen un gran número de parámetros que debemos escoger (previamente al entrenamiento) para definir una topología concreta de red neuronal. Este tipo de parámetros se denominan hiperparámetros (para así distinguirlos de los parámetros o pesos de la red neuronal) y, aprovechando la notación usada para el Perceptrón Multicapa de la sección anterior, entre ellos encontramos:

- Tasa de aprendizaje (*learning rate*), con valores entre 0 y 1. Aquellos valores cercanos a 0 impiden la actualización de parámetros mientras que valores mayores a 1 suelen implicar que los parámetros vayan tendiendo a infinito.
- Número de capas ocultas,esto es, $L-2$ según nuestra notación, y número de neuronas en cada capa oculta, denotado como d_l para la capa l-ésima.
- Funciones de activación a aplicar en cada una de las capas ocultas. Por ejemplo, $h^{(l+1)}$, que se aplica en la capa l. A modo de referencia, podemos encontrar un ejemplo con las funciones de activación más comunes en la figura 4.4.
- El número de iteraciones a realizar durante el entrenamiento (denominado también como *epochs*) y el tamaño del lote (*batch size*) a la hora de realizar las actualizaciones de parámetros en las ecuaciones 4.10 y 4.11.

[3] Este algoritmo escapa al alcance de este trabajo, además de que ya se encuentra implementado de manera eficiente por TensorFlow.

- El tipo de optimizador a usar en el descenso de gradiente por lotes. Para más información sobre este amplio campo es interesante consultar un artículo sobre optimizadores en la página web de Sebastian Ruder [44].

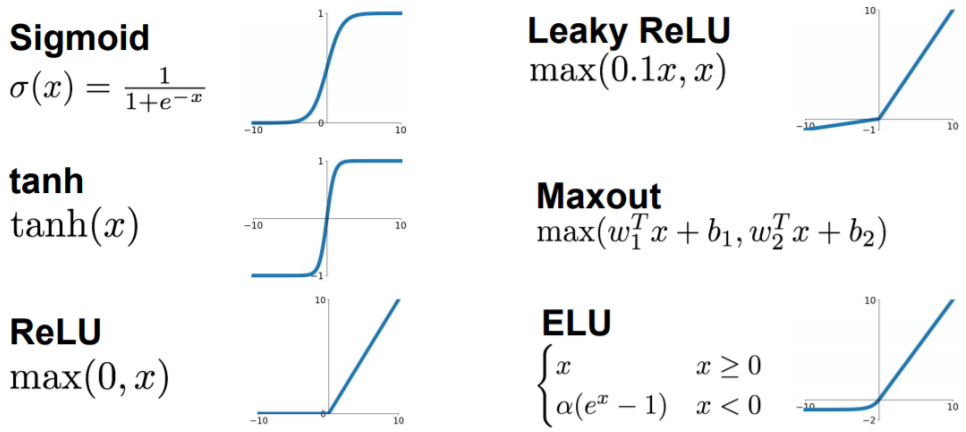


Figura 4.4: Ejemplos de funciones de activación

Fuente: CS231n Stanford. Lecture 6 - Diapositiva 15

Todos estos hiperparámetros se pueden escoger de forma flexible en el benchmark que se ha desarrollado, permitiendo por lo tanto experimentar y analizar distintos modelos de redes neuronales.

4.1.3. Optimizaciones

Para terminar esta sección hablaremos de tres técnicas que se pueden usar en redes neuronales profundas y que han sido implementadas en el benchmark. Estas optimizaciones permiten atacar problemas como el *sobreajuste* durante el entrenamiento, esto es, redes neuronales que no permiten realizar predicciones precisas para datos que no sean los del conjunto de entrenamiento.

Dropout

Esta técnica fue detallada por primera vez en un artículo de 2014 [75] y consiste en ignorar o *apagar* neuronas de la red neuronal, a lo largo del proceso de entrenamiento, con una cierta probabilidad denominada *tasa de dropout* (*dropout rate*). Al ir eliminando neuronas de forma aleatoria conseguimos ir probando nuevas topologías de redes neuronales en el entrenamiento, haciendo que nuestra red sea más robusta frente a pequeñas variaciones en los valores de la capa de entrada. En la figura 4.5 se muestra el resultado de aplicar dropout a un Perceptrón Multicapa con una sola neurona de salida.

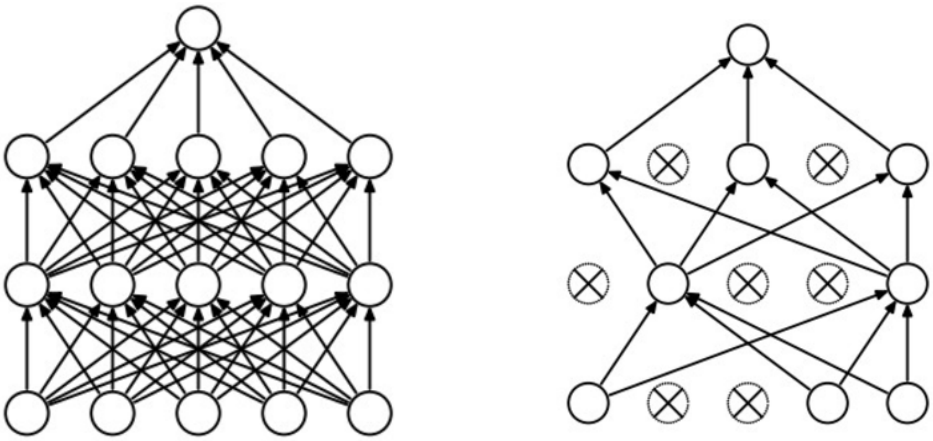


Figura 4.5: Redes neuronales antes y después de aplicar *dropout* respectivamente

Fuente: Dropout: A Simple Way to Prevent Neural Networks from Overfitting [75]

El benchmark soporta la posibilidad de realizar dropout durante el entrenamiento de una red neuronal, otorgando al usuario la posibilidad de fijar una tasa de dropout determinada.

Batch Normalization

Otra técnica surgida recientemente, concretamente en un artículo de 2015 [71], es *batch normalization*, que permite corregir durante el entrenamiento el problema de la alta variabilidad en la distribución de los inputs para cada neurona de la red. Para ello, se realiza el siguiente proceso en cada capa, justo antes de aplicar la función de activación: en primer lugar, para cada *batch* o lote de datos (con tamaño *batch_size*) de entrenamiento se normalizan los valores de entrada de las neuronas (restando la media y diviendo por la varianza) para posteriormente multiplicarlos por un parámetro (γ o *scaling parameter*) y sumarles otro más (β o *shifting parameter*):

$$\mu_{batch} = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} x_i \tag{4.12}$$

$$\sigma^2_{batch} = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} (x_i - \mu_{batch})^2 \tag{4.13}$$

$$x_i^{(norm)} = \frac{x_i - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \epsilon}} \qquad i \in \{1, 2, \ldots, batch\_size\} \tag{4.14}$$

$$z_i = \gamma x_i^{(norm)} + \beta \qquad i \in \{1, 2, \ldots, batch\_size\} \tag{4.15}$$

En las ecuaciones 4.12, 4.13 y 4.14 se realiza la normalización de los datos de entrada (es decir, los distintos x_i del batch, ya sean pertenecientes a la capa de entrada o a una neurona oculta cualquiera), donde μ_{batch} y σ^2_{batch} son la media y la varianza de un batch determinado y ϵ es un parámetro usado para evitar la división entre cero.

Por último, en la ecuación 4.15 *escalamos* y *desplazamos* los datos normalizados, para obtener así la salida deseada, z_i.

En resumen, para cada capa se ha añadido cierta complejidad al proceso de entrenamiento, puesto que se tiene que realizar el *aprendizaje* (utilizando también descenso de gradiente y backpropagation) de 4 parámetros: μ y σ^2, esto es, la media y la varianza del conjunto de entrenamiento (necesarias para realizar predicciones cuando no estemos entrenando), así como los parámetros γ y σ.

Las ventajas de esta técnica son mejores resultados en menos *epochs* de entrenamiento (debido a que los creadores de esta técnica probaron que se podían usar tasas de aprendizaje muy altas), mejoras en las predicciones realizadas por las redes y eliminación del problema de *sobreajuste* (es decir, batch normalization actúa también como *regularizador*) [71]. Además, si en la capa de entrada aplicamos batch normalization, estaremos aprovechando para realizar una normalización de los datos de entrada, en el caso de que no lo estuvieran previamente.

Regularización L1 y L2

Las regularizaciones L1 y L2 permiten reducir el sobreajuste producido en el entrenamiento (*overfitting*) de redes neuronales, añadiendo términos extra a la función de coste (véase ecuación 4.8) y provocando así una penalización en el caso de que haya pesos en la red neuronal cuyo valor absoluto sea muy grande.
En el caso de la regularización L1, la función de coste es la siguiente:

$$L_{L1}(X,T,W) = L(X,T,W) + \sum_{i,j,l} |W_{ij}^{(l)}| \tag{4.16}$$

Para la regularización L2, la función de coste se convierte en:

$$L_{L2}(X,T,W) = L(X,T,W) + \| \sum_{i,j,l} W_{ij}^{(l)} \|_2^2 \tag{4.17}$$

Destacar que en ambas ecuaciones seguimos denotando como $W^{(l)}$ a las matrices de pesos de la capa l-ésima.

4.2. Desarrollo del benchmark

El objetivo de está sección es dar una visión general del benchmark local que he diseñado para este trabajo. Antes de presentar el conjunto de herramientas creado, es preciso detallar como funciona la ingesta de datos en el benchmark y como éste realiza una división en tres conjuntos de datos. Además, se comentarán las distintas formas que tiene el benchmark de evaluar los entrenamientos realizados, algo crucial para un posterior análisis de resultados.

Por último, hablaremos de las herramientas desarrolladas para el benchmark (veáse figura 4.6), realizando una descripción de sus funcionalidades y ejemplos de uso de las mismas. Los códigos desarrollados para estas herramientas se pueden encontrar en la carpeta *benchmark* del repositorio Github del trabajo [55].

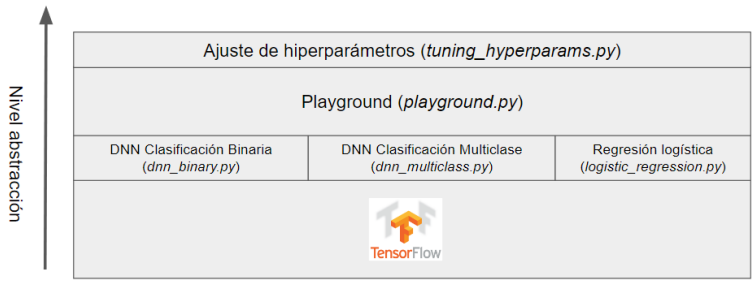


Figura 4.6: Herramientas y códigos desarrollados para el benchmark

Fuente: Elaboración propia

4.2.1. Ingesta y particionado para entrenamiento, validación y test

El proceso de ingesta de datos para un dataset cualquiera y su posterior de división en tres conjuntos disjuntos se realiza con la clase *Dataset* de Python (*dataset.py*). Con el objetivo de desarrollar un módulo ligero y eficiente, esta clase admite datasets en forma de csv o archivos de Numpy (.npy). En ambos casos, las distintas observaciones del dataset se encuentran apiladas como vectores fila, donde la última variable o componente de cada observación es la clase o etiqueta a la que pertenece. Todas y cada una de las componentes deben tener tipo númerico, incluyendo las etiquetas, que toman valores enteros comenzando desde el 0. Es decir, si tenemos 2 clases las indicaremos con los valores 0 y 1, para 3 clases tendremos los valores 0, 1 y 2, y así sucesivamente.

El módulo se ha diseñado de tal manera que si le pasamos por primera vez un archivo *csv*, lo parseará, lo convertirá en un array de Numpy y finalmente lo guardará en un archivo *npy*. De esta manera logramos por un lado poder evitar usar el archivo csv, el cual

suele ser más grande que el archivo .npy[4], y así utilizar en varias ejecuciones del benchmark este último tipo de archivos, que permite cargar directamente en memoria *arrays de Numpy*, una estructura de datos utilizada para entrenar los modelos de TensorFlow.

Otra de las funciones de la clase Dataset es la de dividir los datos recibidos en conjuntos (disjuntos) de entrenamiento, validación y test. Al instanciar un objeto de esta clase, además de indicar el nombre del dataset sobre el que trabajaremos (sin extensión de archivo csv o npy), también podemos fijar el porcentaje de observaciones del dataset que poseerán cada uno de estos conjuntos. Esta división se realiza tras barajar en primer lugar todas las observaciones del dataset y permite que cada una de estas tres particiones del dataset tengan una función específica durante la ejecución del benchmark, tal y como vemos en la figura 4.7.

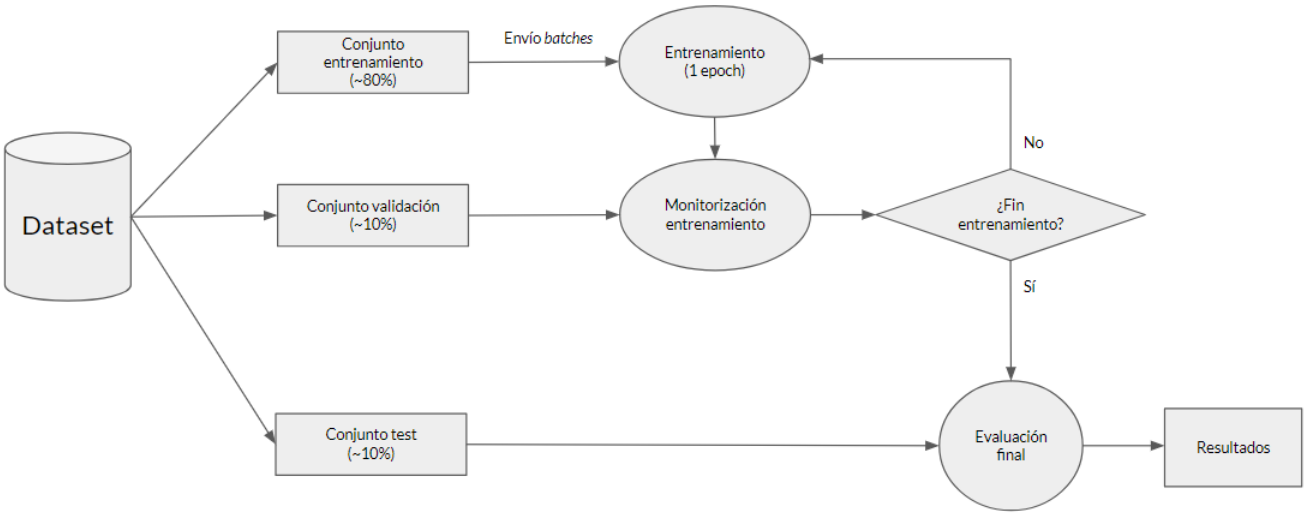


Figura 4.7: Funcionalidad de los conjuntos de entrenamiento, validación y test

Fuente: Elaboración propia

Por un lado, el conjunto de entrenamiento (normalmente el 80 % de las observaciones) se destinará al proceso de aprendizaje de la red neuronal, que se realizará utilizando un descenso de gradiente por medio de lotes o *batches* con un tamaño prefijado (*batch size*). Estos lotes se obtienen gracias a la función *next_batch()* de la clase *Dataset*.

En cambio, los datos de validación (en torno al 10 % de observaciones del dataset normalmente) servirán para verificar si al acabar una iteración (*epoch*) del entrenamiento el modelo se está entrenando de forma correcta. En caso contrario, se pueden detectar a

[4]Por ejemplo, el dataset CCF en formato csv ocupa unos 150 MB, mientras que en formato npy ocupa 70 MB, algo menos de la mitad.

tiempo sobreajustes del modelo (*overfitting*) al conjunto de entrenamiento, evitando de esta manera numerosos fallos de predicción para el resto de observaciones del dataset.

Finalmente, el conjunto de test (comúnmente el 10 % de filas del dataset) se reserva para cuando el modelo predictivo haya sido entrenado por completo. Como los datos de test no han influido directamente sobre el proceso de entrenamiento, las predicciones realizadas al aplicar estos datos sobre el modelo entrenado nos dan una estimación real de la calidad del entrenamiento.

4.2.2. Evaluación de modelos

En este apartado hablaremos de los distintos métodos de evaluar los modelos de redes neuronales entrenados por el benchmark, usando principalmente la información contenida en el libro de Geron [70]. Todos estos métodos de evaluación se han aplicado en los distintos códigos Python del benchmark.

Accuracy

Se denomina *accuracy* al porcentaje de observaciones correctamente clasificadas. Nótese como esta métrica es válida tanto para problemas de clasificación binarios como multiclase:

$$accuracy(\,\%) = \frac{inst\_correctas}{N\_eval} * 100 \tag{4.18}$$

Con N_eval el número de observaciones sobre el que estamos evaluando esta métrica e $inst\_correctas$ el número de observaciones para las que la clase predicha y la clase con la que se etiqueta la observación son iguales.

Hay que advertir que el valor que aporta esta métrica es realmente pequeño para el caso de problemas de clasificación donde el número de observaciones pertenecientes a una clase es bastante superior al resto de clases (por ejemplo, en el dataset CCF).

AUC

Antes de definir está métrica debemos suponer que estamos ante un problema de clasificación binario (denoteramos las dos clases objetivo como las clases 1 y 0) y presentar una serie de conceptos:

TP (*true positives*) y TN (*true negatives*) son el número de observaciones que hemos predicho de forma correcta que pertenecen a las clases 1 y 0 respectivamente. Por otro lado, FP (*false positives*) y FN (*false negatives*) son el número de observaciones que hemos predicho de manera incorrecta que pertenecían a las clases 1 y 0 respectivamente.

Con ello, podemos definir los ratios TPR y FPR (*true positive rate* y *false positive rate*):

$$TPR = \frac{TP}{TP + FN} \tag{4.19}$$

$$FPR = \frac{FP}{TN + FP} \tag{4.20}$$

Una curva ROC se puede pintar en una gráfica enfrentando valores TPR en el eje de abcisas y valores FPR en el de ordenadas. Para obtener distintos valores de estos ratios tomaremos distintos *thresholds* entre 0 y 1, los cuales imponen un límite tal que, dada una observación, si el valor resultante de realizar una predicción es superior a ese límite, será transformado en 1 y por lo tanto la observación será catalogada como un ejemplo positivo (esto es, que pertenece a la clase 1) . Normalmente, cuando se realizan evaluaciones sobre el conjunto de test se impone un *threshold* de 0.5, aunque esto puede variar en el caso de que prefiramos que FP sea mayor a FN[5].

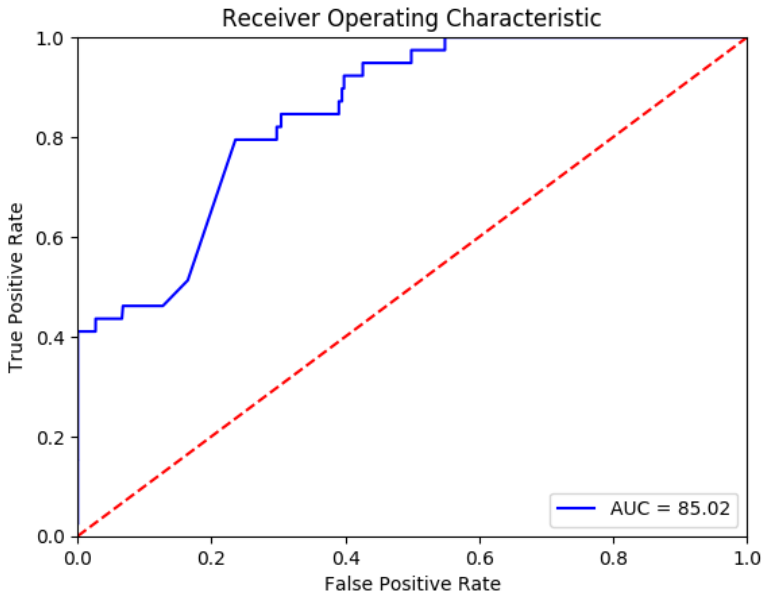


Figura 4.8: Ejemplo de curva ROC con su AUC correspondiente

Fuente: Imagen producida por una ejecución del benchmark (*playground.py*) [55]

[5]En el caso del dataset CCF, suponiendo que la clase 1 es la de transferencias fraudulentas, es preferible tener un bajo número de FN, esto es, transferencias que no hemos podido detectar como fraudulentas, aunque eso conlleve aumentar los falsos avisos de fraude (FP).

Ahora bien, definimos el AUC (*Area Under the Curve*) de una curva ROC como el valor del área que encierra ésta. Un predictor binario que devuelve valores aleatorios posee un AUC de 0.5, mientras que valores en torno a 0.9 son considerados bastante buenos [41]. Esta métrica está normalmente destinada a evaluar problemas donde una de las dos clases está desproporcionada respecto a la otra. En la figura 4.8 podemos ver un ejemplo de curva ROC y su valor AUC. Aunque el valor AUC está siempre comprendido entre 0 y 1, en nuestro benchmark se ha decidido devolverlo en forma de porcentaje.

Matriz de confusión

El objetivo de esta métrica es ver en detalle cuantas predicciones hemos realizado para cada una de las clases objetivo, cuales han sido correctas y cuales no.

Como podemos ver en la figura 4.9, en una matriz de confusión $CF = (cf_{ij})_{i,j}$ se etiquetan las filas y las columnas con los nombres de las clases del problema de clasificación a resolver (en el caso del dataset CCF son las transferencias fraudulentas y las normales),

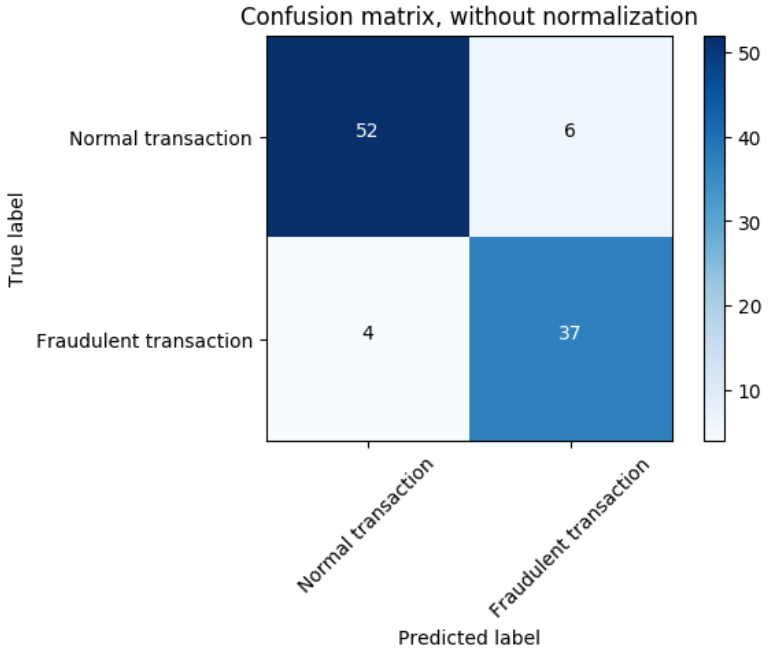


Figura 4.9: Matriz de confusión para un conjunto de datos de test del dataset CCF

Fuente: Imagen producida por una ejecución del benchmark (*playground.py*) [55]

definiendo sus elementos, cf_{ij}, como el número de observaciones que han sido clasificadas en la clase de la columna j-ésima y que pertenecen realmente a la clase de la fila

i-ésima. Nótese como las observaciones clasificadas correctamente son aquellas situadas en la diagonal de la matriz de confusión. Con esta información podemos averiguar si se están haciendo un gran número de predicciones correctas para una determinada clase o si por el contrario, existe algún tipo de confusión entre ciertas clases a la hora de realizar predicciones. Además el benchmark también ofrece matrices de confusión normalizadas, donde los elementos tienen la forma:

$$\tilde{cf}_{ij} = \frac{cf_{ij}}{\sum_j cf_{ij}} \tag{4.21}$$

Esta métrica de evaluación es válida tanto para problemas de clasificación binarios como multiclase, obteniendo en general una matriz de confusión de dimensión $m \times m$, con m el número de clases definidas en el problema de clasificación correspondiente.

4.2.3. Ajuste de hiperparámetros

Basándonos en las posibles soluciones al problema de búsqueda y ajuste de hiperparámetros de una red neuronal (*hyperparameter tuning*) [1], he desarrollado un código (*tuning_hyperparams.py*) que realiza una búsqueda exhaustiva en un conjunto de hiperparámetros dados (*grid search*), generando sucesivos modelos que se entrenan y evalúan independientemente. Editando el script Python mencionado, simplemente tenemos que indicar la ruta del dataset (este último sin extensión, para que sea pueda ser utilizado por la clase Dataset) y las listas con los distintos valores de hiperparámetros a probar. Se debe prestar atención a este último punto y no poner demasiados valores distintos, pues al realizarse *grid search* se probará con todas las combinaciones de hiperparámetros indicadas, pudiendo resultar en un problema con complejidad exponencial.

Los resultados de este ajuste de hiperparámetros se irán guardando por defecto dentro de una carpeta *tmp* del directorio raíz[6], generando ahí un archivo de texto (*tuning_results.txt*), así como los distintos modelos entrenados (archivos *ckpt*), de cara a poder ser reusados por otros códigos de TensorFlow en un futuro [57]. El archivo de texto contiene los tiempos de entrenamiento y los hiperparámetros utilizados para cada uno de los modelos entrenados. En la parte final del txt se indica el modelo óptimo, elegido por tener un AUC superior al resto. No obstante, podría ser interesante cambiar este criterio por otro que ponderara AUC con el tiempo de entrenamiento, aunque esto depende de las prioridades que nos hayamos marcado para resolver nuestro problema.

Por otro lado, también se guardan archivos de TensorBoard para cada modelo, indispensables para comparar de manera gráfica métricas como AUC o *accuracy*. Por ejemplo, en la figura 4.10 vemos como evoluciona el AUC a lo largo del entrenamiento

[6] Dentro de *tmp* se guardan todos estos resultados en una carpeta con el esquema *hyptuning-run-fecha–hora*.

para 4 modelos distintos. Nótese como se ha aplicado una media exponencial (*smoothing*) en TensorBoard para corregir los picos en los resultados de los modelos y analizar las tendencias de estas métricas a medida que avanza el entrenamiento.

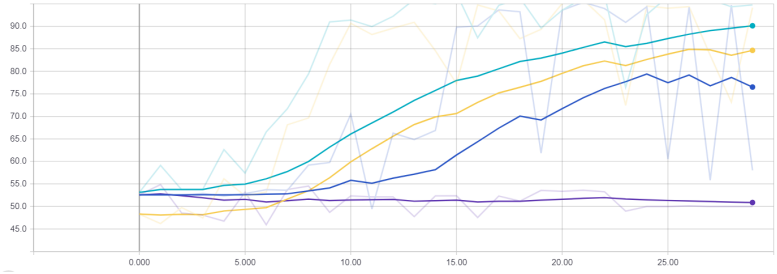


Figura 4.10: Resultados del ajuste de hiperparámetros en Tensorboard

Fuente: Resultado de una ejecución de *tuning_hyperparams.py* [55]

Debido al alto nivel de abstracción de este componente del benchmark, pues únicamente es necesario introducir el nombre del dataset y los distintos hiperparámetros a probar, sería una buena idea empezar con él las pruebas de entrenamiento en esta segunda fase del marco de trabajo (véase figura 4.1). Además, es interesante utilizar como punto de partida los hiperparámetros utilizados en la sección 3.3, dado que pueden servirnos como guía para realizar un ajuste óptimo.

Finalmente, indicar que en el presente trabajo también se explica como realizar un ajuste automático de hiperparámetros en la nube, utilizando en este caso ML Engine (sección 5.1.2).

4.2.4. Playground

El archivo *playground.py* es la parte central del benchmark desarrollado y consiste en un script que realiza un parseo de hiperparámetros por línea de comandos, instanciando una determinada red neuronal para resolver problemas de clasificación binarios o multiclase (clases *DNN* de los archivos *dnn_binary.py* y *dnn_multiclass.py* respectivamente). También se debe indicar la ruta de un archivo csv o npy (sin extensión) para que la ingesta y la partición del dataset pueda ser realizada por la clase *Dataset* (script *dataset.py*).

A continuación mostramos un ejemplo de llamada a *playground.py* a través de la línea de comandos, suponiendo que el archivo *creditcard* (ya sea en formato npy o csv), esté en la misma ruta que el script.

```
$ python .\playground.py \
--dataset_file creditcard \
--hidden_layers 10 5 \
--epochs 100 \
--batch_size 50 \
--learning_rate 0.1
```

En la llamada hemos utilizado el dataset CCF, creando una red neuronal con dos capas ocultas de 10 y 5 neuronas respectivamente. Hemos realizado un entrenamiento de 100 iteraciones o *epochs*, utilizando para el mismo un *batch size* de 50 observaciones. Por último, la tasa de aprendizaje que toma el optimizador por defecto (*Adam*) es de 0.1. Dado que no hemos indicado algunos hiperparámetros que también acepta el script, se han aplicado aquellos por defecto: se ha usado la función de activación *elu* y *batch normalization*, no se aplica regularización *L1* ni *L2* y no se ha utilizado *dropout*. Además, como no hemos indicado un directorio donde guardar los resultados (argumento *log_dir*), se han creado un par de carpetas (*tmp* y *playground-run-fecha–hora* dentro de la primera) por defecto en el directorio raíz.

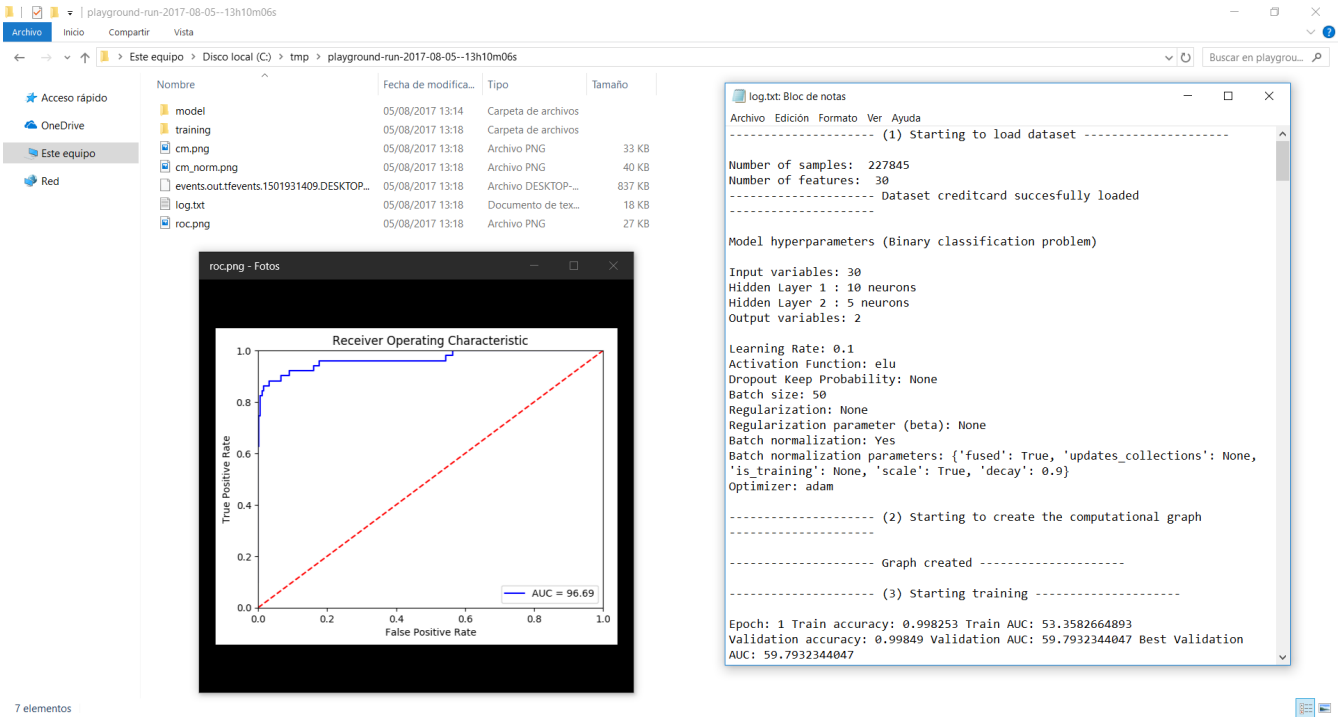


Figura 4.11: Archivos generados por *playground.py*, con detalle de log y curva ROC

Fuente: Resultado de una ejecución de *playground.py* [55]

El resultado del comando anterior se muestra en la figura 4.11. Se puede ver como se han generado varios archivos en una carpeta del directorio */tmp*, mostrando un detalle del log de resultados (*log.txt*) y la curva ROC (*roc.png*, con el AUC correspondiente). Los archivos *cm.png* y *cm_norm.png* contienen las matrices de confusión (sin normalizar y normalizadas respectivamente) para los datos de test. Por otro lado, el archivo cuyo nombre contiene la cadena de caracteres *tfevents* es el encargado de almacenar la información necesaria para mostrar los resultados del entrenamiento en TensorBoard. Las carpetas *model* y *training* contienen el modelo con los parámetros óptimos (mejor resultado AUC sobre los datos de validación) y el último modelo entrenado respectivamente, para que así puedan ser usados de nuevo por TensorFlow.

Este playground es recomendable aplicarlo tras realizar un ajuste de hiperparámetros primero (*tuning_hyperparams.py*) y así utilizar los hiperparámetros del modelo óptimo para obtener resultados más exhaustivos. Finalmente, podemos volver a reutilizar estos hiperparámetros para entrenar un modelo en Google Cloud con ML Engine (sección 5.1.1), en el caso de que contemos con datasets adicionales de varios millones de filas y/o las redes neuronales a entrenar sean bastante *profundas* y por lo tanto tengan un gran número de parámetros a entrenar.

A continuación describiremos los dos modelos de redes neuronales desarrollados en TensorFlow y que puede utilizar nuestro playground (por defecto utiliza el de redes neuronales para problemas de clasificación multiclase). Estos dos códigos se han desarrollado utilizando partes de un notebook del repositorio del libro de Geron en Github [42], ejemplos de TensorFlow de otro repositorio [51], tutoriales de TensorFlow.org [39] y código desarrollado para la asignatura de Geometría Computacional [53].

Redes neuronales de clasificación multiclase

Apoyándonos en la teoría expuesta en la sección 4.1.1 he creado un Perceptrón Multicapa en TensorFlow (*dnn_multiclass.py*) para resolver problemas de clasificación multiclase.

La clase *DNN* creada en el script instancia grafos de cómputo en TensorFlow que modelan un Perceptrón Multicapa a partir de una serie de hiperparámetros que fijamos. Cuando hayamos creado un objeto de la clase DNN (siglas provenientes de Deep Neural Networks), podremos realizar un entrenamiento de la red neuronal por medio del método *train*, al que pasaremos entre otro argumentos, el número de *epochs* a realizar y el *batch_size*.

Existe también un método *test* que permite obtener el valor AUC y el *accuracy* sobre un conjunto de datos de test determinados. Además, si queremos conocer en detalle las predicciones realizadas por nuestra red, tenemos el método *predict*, el cual devuelve un vector de probabilidades para cada clase y cada observación, y *predict_class*, que nos

devuelve la clase predicha para cada observación (tomando la probabilidad mayor entre todas las clases).

Otros métodos auxiliares de esta clase son *save_roc*, *save_cm*, que guardan en formato png una curva ROC y una matriz de confusión respectivamente. Estos dos métodos son usados por *playground.py* para generar archivos como los de la figura 4.11

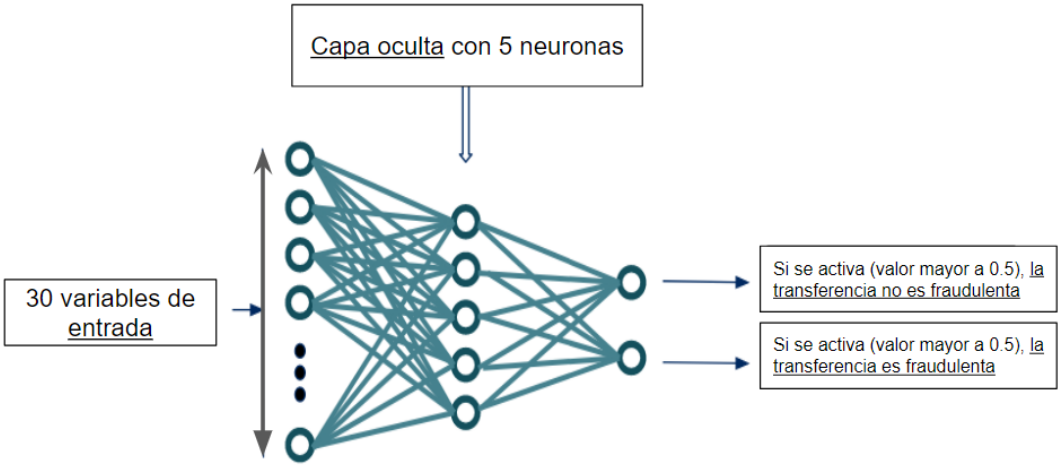


Figura 4.12: Ejemplo de red neuronal aplicado al dataset CCF

Fuente: Elaboración propia

Nótese como podemos resolver problemas de clasificación binarios si escogemos $m = 2$, pues tendremos 2 neuronas en la capa de salida. Por ejemplo, esto se puede usar en el dataset CCF, utilizando estructuras de redes neuronales como la propuesta en la figura 4.12.

Redes neuronales de clasificación binaria

Este modelo es prácticamente idéntico al anterior, con la salvedad de que el código (*dnn_binary.py*) está preparado para resolver problemas de clasificación binarios, utilizando una única neurona en la capa de salida. Utilizando la notación de la sección 4.1, en este caso hemos adaptado por un lado la función de coste:

$$L(X, T, W) = -\sum_{i=1}^{N} t_i log(y(\vec{x_i}, W)) + (1 - t_i) log(1 - y(\vec{x_i}, W)) \tag{4.22}$$

En este caso $y(\vec{x_i}, W) \in \{0, 1\}$ es un valor binario y no un vector debido a que solo podemos realizar dos posibles predicciones para cada observación. Nótese como esta función de coste es equivalente a la descrita en la ecuación 4.8 con $m = 2$. Esto se debe

al hecho de que tras aplicar la función softmax a la dos neuronas de la capa de salida, la suma de sus valores es igual a 1.

Por último, en este caso, como solo tenemos una neurona de salida no podemos aplicar sobre ella la función softmax, por lo que se utiliza en este caso una función sigmoide. Es interesante ver como la segunda componente del vector softmax (esto es, la probabilidad de que una observación pertenezca a la clase 1), en el caso de que estuviéramos utilizando una red neuronal con dos neuronas de salida ($m = 2$), se trata de una función sigmoide aplicada a la diferencia de los dos valores de las de las neuronas de salida:

$$\begin{aligned} P(t_{i2} = 1|\vec{x_i}) = 1 - P(t_{i1} = 1|\vec{x_i}) = 1 - softmax((z_1, z_2)) = 1 - \frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \\ = \frac{e^{z_2}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{-(z_2 - z_1)}} = \sigma(z_2 - z_1) \end{aligned} \tag{4.23}$$

z_1 y z_2 son los valores de las dos neuronas de salida de la red neuronal (antes de aplicar la función softmax) y t_{i1} y t_{i2} las componentes que indican si la clase de la observación $\vec{x_i}$ es la 0 ($t_{i1} = 1$) o la 1 ($t_{i2} = 1$). Por tanto, como en un problema de clasificación binario solamente estamos interesados en conocer $P(t_{i2} = 1|\vec{x_i})$, es claro que, reajustando el valor de los pesos, podemos sustituir las dos neuronas de la capa de salida por una sola y aplicar sobre ella una función sigmoide. Es interesante destacar que número de parámetros de la red neuronal disminuye si usamos este último enfoque en vez del otro.

5 Entrenamiento a gran escala de redes neuronales en la nube

En este capítulo explicaremos como se pueden realizar entrenamientos de redes neuronales con datasets de gran tamaño. Este proceso se corresponde con la tercera y última fase del marco propuesto en la figura 1.2. La idea de esta fase es que una vez que se hayan probado en el benchmark local distintos hiperparámetros de redes neuronales, se pueda escalar este entrenamiento (usando los hiperparámetros del modelo óptimo del benchmark) a datasets de tamaño bastante superior al utilizado en un principio. Esto es interesante en el caso de que se hayan recogido observaciones adicionales a las que teníamos en un principio o si desde un primer momento hemos estado trabajando con un dataset muestreado (con observaciones suficientemente representativas) y queramos extender el análisis y los entrenamientos al dataset completo.

Aunque no sea un dataset de gran tamaño, utilizaremos el dataset CCF para probar los distintos servicios de entrenamiento que nos ofrece ML Engine. Esta vez utilizaremos TensorFlow 1.2 en vez de 1.1, como hicimos en los capítulos 3 y 4, dado que los ejemplos de ML Engine adaptados en este capítulo estaban realizados para esa versión de la librería. Además, para cerrar este capítulo hablaremos de modelos Wide and Deep, utilizados en los entrenamientos de ML Engine y que combinan regresiones de tipo lineal y redes neuronales para ofrecer modelos más robustos y flexibles.

5.1. ML Engine

Debido a la complejidad de la herramienta y la abundante documentación que posee [61], todo ello unido al hecho de que es un producto bastante reciente y cambiante (de hecho salió de la fase beta en marzo de 2017 [40]), he decidido adaptar uno de los ejemplos ofrecidos por ML Engine para realizar nuestro entrenamiento con el dataset CCF [26]. Este ejemplo usa librerías de alto nivel de TensorFlow (*tf.contrib.learn* y *tf.contrib.layers*) que he adaptado fácilmente a nuestro dataset.

De forma similar a como hicimos en el capítulo 3, ML Engine realizará la ingesta desde Cloud Storage y depositará ahí también los modelos entrenados (en formato .ckpt para que puedan ser usados por otros programas de TensorFlow), así como archivos de TensorBoard con los que visualizar los resultados obtenidos. Todo esto se realizará a través del script Bash *scriptml.sh* que podemos encontrar en el repositorio Github del proyecto [55]. El único requisito para poder ejecutarlo es que tenemos que haber

instalado y configurado previamente el SDK de Google Cloud en nuestro ordenador. Hay que destacar que en el apéndice 5 podemos encontrar un análisis de los resultados de ML Engine para un ajuste de hiperparámetros y un par de entrenamientos, usando en todos ellos modelos Wide and Deep y el dataset CCF.

Para cerrar esta sección se expondrán los precios por el uso de esta herramienta y se ofrecerán un par de estimaciones de coste.

5.1.1. Entrenamiento

El objetivo de este apartado es comentar de forma breve los pasos para realizar un entrenamiento en ML Engine [46]: en primer lugar depositaremos un par de archivos csv en Cloud Storage que servirán como datos de entrenamiento y datos de validación. En nuestro caso, he dividido el dataset CCF con un script en R (*split.R*, que además permite decidir si realizar o no undersampling), para después subir manualmente los archivos csv resultantes a Cloud Storage. Este último paso también se podría realizar por medio del comando *gsutil cp*, perteneciente al SDK de Google Cloud.

El siguiente paso es desarrollar el código para entrenar los modelos en la nube. Dado que era bastante díficil adaptar el código del benchmark local desarrollado en la fase previa, se ha optado por adaptar un código de ejemplo proporcionado por ML Engine [26]. Esta adaptación ha consistido en definir las columnas de nuestro dataset (y declararlas de tipo númerico) en el archivo *model.py*, el cual se encuentra dentro de la carpeta *trainer*, presente también en el repositorio del trabajo [55].

En este script Python también se proponen dos maneras distintas para resolver el problema de clasificación asociado al dataset que estemos usando: mediante redes neuronales profundas (función *DNNClassifier*) o con modelos Wide and Deep (*DNNLinearCombinedClassifier*). Es interesante destacar que, dado que son funciones de muy alto de nivel del paquete *tf.contrib.learn* y están bastante orientadas para realizar diversas pruebas sobre ellas, aceptan también variables categóricas o discretas. Este tipo de variables pueden ser de tipo string (cadena de caracteres), a diferencia del tipo de variables que hemos estado tratando en este trabajo, todas ellas númericas.

Por tanto, solo necesitamos ejecutar el siguiente comando para entrenar el modelo:

```
$ gcloud ml-engine jobs submit training JOB_ID  \
--stream-logs \
--runtime-version 1.2 \
--job-dir $GCS_JOB_DIR \
--module-name trainer.task \
--package-path trainer/ \
--scale-tier $SCALE_TIER \
--region us-central1 \
-- \
--train-files $TRAIN_FILE  \
--train-steps $TRAIN_STEPS \
--eval-files $EVAL_FILE \
--first-layer-size 25 \
--num-layers 2 \
--verbosity DEBUG \
--eval-steps $EV_STEPS
```

ML Engine se encargará de arrancar un cluster de máquinas determinado con SCALE_TIER, ejecutando en cada una de ellas el código que he alojado en la carpeta *trainer*, el cual se ha convertido a un paquete de Python de forma automática. Comentar como curiosidad, que si elegimos *STANDARD_1* como cluster, ML Engine se encargará automáticamente de realizar entrenamientos usando TensorFlow distribuido (sección 2.2.2).

Aunque existen numerosos parámetros para este comando [6], sintetizaremos diciendo que *JOB_ID* es un identificador alfanumérico único que debemos dar al trabajo de entrenamiento que estamos realizando, mientras que *GCS_JOB_DIR* almacena una ruta en Storage donde se depositarán los resultados. Nótese como hemos estamos ejecutando ML Engine con la versión 1.2 de TensorFlow y como el argumento " − −" separa los primeros argumentos, que utiliza ML Engine, del resto, los cuales son parseados por el archivo *task.py*, situado en la carpeta *trainer*.

Por último, el script que he desarrollado descargará los resultados desde Storage, los comprimirá en un zip y lo subirá de nuevo a Storage, borrando la carpeta original de resultados. En el caso de que queramos seguir en tiempo real el log de ejecución de los trabajos de entrenamiento, podemos utilizar la herramienta Stackdriver de Google Cloud Platform (podemos acceder buscando Logging en la *Cloud Console*) y ver los logs de un trabajo determinado, como podemos apreciar en la figura 5.1.

5.1.2. Ajuste automático de hiperparámetros

Otro de los interesantes servicios que proporciona ML Engine es el de realizar un ajuste automático de hiperparámetros (*hyperparameter tuning*) en nuestros modelos [22]. Con ello podremos realizar varias pruebas (incluso algunas de ellas de manera simultánea) con distintos hiperparámetros de nuestra red neuronal (número de capas ocultas, neuronas

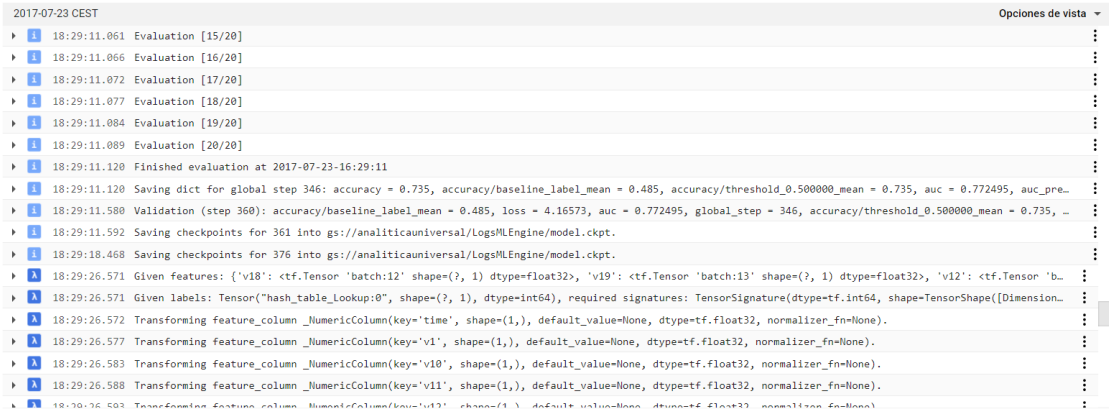


Figura 5.1: Log de un trabajo realizado por ML Engine

Fuente: Stackdriver Logging

en cada capa oculta, tamaño del batch, etc.). Esta serie de pruebas es similar a la que se realiza en el benchmark que he desarrollado (sección 4.2.3).

Todo esto se realiza de manera muy sencilla, creando un archivo *hptuning_config.yaml*, el cual podemos encontrar de ejemplo tanto en el repositorio de ejemplo de ML Engine [26] como en el del presente trabajo [55]. En el archivo YAML necesitaremos indicar en el campo *goal* la métrica que maximizaremos o minimizaremos. El nombre de esta métrica viene dado en *hyperparameterMetricTag* y para el ejemplo que tenemos, he seleccionado *auc*, aunque cualquiera de las métricas que evaluamos y guardamos en TensorBoard puede ser escogida (por ejemplo, *accuracy*). Ahora bien, también necesitaremos indicar el número de pruebas para cada hiperparámetro y el número de pruebas a realizar en paralelo (campos *maxTrials* y *maxParallelTrials* respectivamente). Por último, escribiremos los hiperparámetros a ajustar por medio de su nombre (tal y como aparece en los argumentos aceptados por *task.py*) en el campo *parameterName*, su tipo, los valores mínimos y máximos a tomar durante la prueba y el tipo de escala (*scaleType*), que indica si los valores de los hiperparámetros estarán equiespaciados (*UNIT_LINEAR_SCALE*), o si lo estarán dentro una escala logarítmica (*UNIT_LOG_SCALE*, donde habrá más valores situados cerca del valor máximo definido). Para más información sobre las distintas opciones que ofrece el archivo YAML de configuración de hiperparámetros, se puede consultar la documentación de ML Engine en la bibliografía [9].

Los entrenamientos con ajuste de hiperparámetros se realizan con el mismo comando que el utilizado en la sección anterior, solo que ahora añadiremos el siguiente argumento (antes del argumento ” − −”, pues es información que debe ser pasada a ML Engine):

```
$ --config $HPTUNING_CONFIG
```

HPTUNING_CONFIG indica en este caso la ruta del archivo YAML con la configuración de hiperparámetros deseada. Destacar que tras el realizar el trabajo de ajuste, se generan tantos modelos de TensorFlow y archivos de TensorBoard como pruebas distintas hayamos indicado. Estos últimos nos serán de gran utilidad en el caso de que queramos realizar un análisis de manera visual del ajuste de hiperparámetros.

5.1.3. Precio

Los costes de ML Engine se dividen por una parte en aquellos derivados de Cloud Storage (Veáse [14] y sección 3.5), pues en esa herramienta se almacenan los datos de entrenamiento y validación que utilizará ML Engine.

Por otra parte, ML Engine computa costes por *unidades de entrenamiento* (*ML Engine training units*), un concepto difuso que se corresponde a la potencia de una máquina estándar que realiza un trabajo de entrenamiento [16]. El precio (a fecha de julio de 2017) por cada unidad de entrenamiento es de $0.49 cada hora. El número de unidades que se utilizarán durante el entrenamiento se establece por medio del argumento *SCALE_TIER*, perteneciente al comando *gcloud* y descrito en 5.1.1.

Para tener un resúmen de los precios y los clusters ofrecidos en ML Engine, conviene consultar la tabla 5.1.

Tabla 5.1: Tabla de precios para distintos clusters de ML Engine en EEUU (julio 2017)

Fuente: Precios de ML Engine

Tipo cluster	**Descripción**	**Coste por hora ($)**
BASIC	Una sola máquina sin GPU	0.49
BASIC_GPU	Una sola máquina con GPU	1.47
STANDARD_1	Gran número de máquinas y unos pocos servidores para almacenar parámetros	4.9
PREMIUM_1	Gran número de máquinas y servidores para almacenar parámetros	36.75

Con el objetivo de ofrecer una estimación de los costes de esta herramienta, tomaremos como ejemplo todas las pruebas realizadas para este trabajo. En total se han realizado unas 10 horas de pruebas, utilizando los dos clusters más básicos (una máquina sin y con GPU respectivamente), costando en total unos $11, una cifra bastante razonable. Debemos remarcar que la mayor parte del tiempo se ha dedicado a experimentar con

los modelos Wide and Deep que hemos utilizado con ML Engine, por lo que una vez adquirida experiencia sobre el funcionamiento de estos modelos, es razonable pensar que pagaremos únicamente por realizar un entrenamiento para ajustar hiperparámetros y otro entrenamiento más (con más iteraciones que en el ajuste) utilizando los hiperparámetros óptimos. Además, también hay que apuntar que antes de lanzarnos a realizar pruebas en la nube con ML Engine, es conveniente realizar entrenamientos en local por medio del comando *gcloud ml-engine local train*, evitando de esta manera posibles costes adicionales no deseados.

A partir de los resultados del entrenamiento en ML Engine realizado en el apéndice 5 para el dataset CCF, realizaremos otra estimación para un dataset con el mismo número de variables pero con un número mayor de observaciones. El entrenamiento mencionado se realizó sobre el dataset CCF utilizando una máquina con GPU (tipo de cluster *BASIC_GPU*) en 5 minutos (unos 35 minutos menos que el mismo entrenamiento realizado en el benchmark local). Suponiendo que el tiempo aumenta de manera proporcional respecto al número de filas del dataset[1], podríamos entrenar unas 10 millones de observaciones en 2 horas y media, lo cual equivale a $3.7. Esta cantidad es algo bastante razonable, teniendo en cuenta que hemos entrenado nuestro modelo en remoto con un gran número de observaciones.

5.2. Modelos Wide and Deep

En esta sección realizaremos una introducción teórica a los modelos *Wide and Deep*, los cuales combinan ventajas de modelos lineales (como las regresiones lineales o logísticas) con modelos de aprendizaje automático profundo (deep learning). Existe una API de alto nivel en TensorFlow que permite utilizar estos modelos [2] y que he probado para realizar entrenamientos en ML Engine con el dataset CCF (apéndice 5). El desarrollo de esta sección se ha realizado tras estudiar el artículo de investigación que presentó este tipo de modelos y que podemos encontrar en la bibliografía [67], así como los tutoriales de TensorFlow para aplicar modelos de regresión lineales [35, 59] y Wide and Deep [60].

En primer lugar, al igual que se hizo en la exposición de la sección 4.1, hay que insistir en que las variables que consideraremos en el desarrollo teórico de esta parte serán todas de tipo numérico. Dado que es algo común encontrarse en datasets con variables categóricas con valores de tipo cadena de caracteres, un tipo de transformación a realizar es usando vectores dispersos, es decir, aquellos cuya mayoría de componentes toman valor 0. Por ejemplo, supongamos que tenemos una variable *altura* que toma como valores: *alto*, *medio* y *bajo*. Para que esta variable sea de tipo númerico, crearemos tres variables binarias *es_alto*, *es_bajo* y *es_medio*, indicando el valor original de *altura*. Es decir, si una observación tiene valor *alto*, entonces *es_alto* tomará valor 1, mientras que los otras dos variables que hemos creado valdrán 0. Esto creará vectores dispersos, que

[1] Existen evidencias que demuestran que si utilizamos GPUs en vez de CPUs el tiempo aumenta muy poco con cantidades cada vez mayores de datos [7].

pueden modelizarse fácil y eficientemente en TensorFlow por medio de las funciones *sparse_column_with_keys* o *sparse_column_with_hash_buckets*. No obstante, una única variable categórica con un elevado número de valores posibles puede generar un número altísimo de variables extra en nuestro modelo, por lo que este podría tardar mucho más tiempo en entrenarse. Para poder paliar este problema, los autores de [67] utilizaron en la parte *Deep* (redes neuronales profundas) de estos modelos lo que se conoce como *word embeddings*, una técnica que permite pasar palabras a vectores con unas pocas componentes de tipo numérico [49]. Debido a que es una técnica muy relacionada con el campo del procesamiento natural del lenguaje y que por tanto escapa a los objetivos del trabajo, he creído conveniente no realizar una explicación más detallada de la misma.

Para poder formular matemáticamente un modelo Wide and Deep, necesitamos previamente introducir modelos de tipo lineal y redes neuronales. Empecemos con los primeros: sea $\vec{x} = (x_1, \ldots, x_k)$ una observación compuesta de k variables distintas. Una regresión lineal realiza una predicción

$$y = W \cdot \vec{x} + b = \sum_{j=1}^{k} w_j x_j + b \tag{5.1}$$

donde $W = (w_1, \ldots, w_k)$ es el vector fila de pesos y b es el bias o sesgo del modelo.

En el caso de que estemos ante un problema de clasificación binaria (las observaciones pertenecen a la clase 1 o 0), la regresión lineal se denomina logística y la predicción realizada es:

$$P(t = 1|\vec{x}) = \sigma(W \cdot \vec{x} + b) \tag{5.2}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.3}$$

siendo $P(t = 1|\vec{x})$ la probabilidad de que la observación $\vec{x}$ pertenezca a la clase 1 y $\sigma(x)$ la función sigmoide. Nótese como esta función únicamente toma valores entre 0 y 1.

Estos dos últimos tipos de regresiones realizan predicciones a partir de relaciones lineales entre variables, lo que puede derivar en que no estemos teniendo en cuenta relaciones de tipo no lineal y que pueden ser clave para realizar una predicción precisa. Para corregir esto podemos introducir (siempre de forma manual, usando por ejemplo la función *crossed_column* de TensorFlow) J variables cruzadas a las k anteriores:

$$VC(\vec{x}) = \begin{pmatrix} VC_1(\vec{x}) \\ VC_2(\vec{x}) \\ \vdots \\ VC_J(\vec{x}) \end{pmatrix} \tag{5.4}$$

$$VC_j(\vec{x}) = \prod_{i=1}^{k} x_i^{c_{ij}} \tag{5.5}$$

donde c_{ij} es un valor booleano que indica si la variable i-ésima esta presente en el cruce j-ésimo.

No obstante, modelos aún más complejos como los de deep learning (por ejemplo las redes neuronales con múltiples capas ocultas) permiten realizar predicciones aún más generales, obteniendo combinaciones de variables no descubiertas con otros modelos.

La fórmula para obtener la predicción de una red neuronal (Perceptrón Multicapa) con $L-2$ capas ocultas, d_l neuronas en la capa l-ésima y propagación hacia delante (*feed-forward*) es la siguiente:

$$a^{(l+1)} = h^{(l+1)}(a^{(l)}W^{(l)} + b^{(l)}) \qquad l \in \{1, 2, \ldots, L-1\} \tag{5.6}$$

siendo $a^{(l+1)}$ las activaciones (con $a^{(1)} = \vec{x}$) , $W^{(l)}$ y $b^{(l)}$ los pesos y bias de la capa l-ésima (matrices con dimensiones $d_l \times d_{l+1}$ y d_{l+1} respectivamente), y $h^{(l+1)} : \mathbb{R}^{d_{l+1}} \to \mathbb{R}^{d_{l+1}}$ la función de activación para esa capa (normalmente ReLU o sigmoide) y L el número de capas totales (contando la capa de entrada y la de salida). Suponiendo que estamos resolviendo un problema de clasificación binaria con una sola neurona en la capa de salida, tendríamos que $P(t = 1|\vec{x}) = \sigma(a^{(L)})$. Nótese como una red neuronal con solo una capa de entrada y otra de salida y que aplica una función sigmoide a esta última es equivalente a una regresión logística.

De manera intuitiva, podemos ver como por un lado, en las regresiones lineales cada uno de los pesos indica la importancia de la variable a la que multiplica, donde pesos con valor cercano a 0 indican la irrelevancia de sus variable correspondientes de cara a formular una predicción.

Sin embargo, las redes neuronales multicapa, en vez de dar información específica sobre la importancia de cada variable, abstrae relaciones (de tipo no lineal) no previstas por otros modelos. Este tipo de relaciones se van "almacenando" en cada una de las neuronas de las distintas capas ocultas, permitiendo obtener niveles de abstracción muy altos, aunque díficiles de interpretar desde el punto de vista analítico. Sin embargo, estas redes neuronales a veces pueden realizar sobregeneralizaciones que resulten en predicciones érroneas.

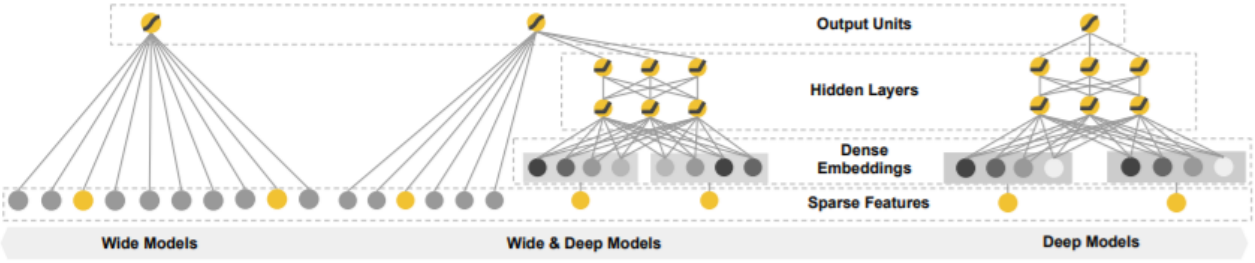


Figura 5.2: Esquema de modelos lineales, "Wide and Deep" y redes neuronales respectivamente

Fuente: Wide and Deep models for Recommender Systems [67]

Por lo tanto, de cara a obtener los beneficios de cada uno de estos dos tipos de modelos, surgen los modelos *Wide and Deep* para problemas de clasificación binarios (veáse figura 5.2), que podemos representar matemáticamente de la siguiente manera:

$$P(t = 1|\vec{x}) = \sigma(W_{lin} \cdot \vec{x}_c + W_{dnn} \cdot a^{(L)} + b) \tag{5.7}$$

$$\vec{x}_c = [\vec{x}, VC(\vec{x})^T] \tag{5.8}$$

W_{lin} es el vector fila con los pesos correspondientes a la parte lineal del modelo, W_{dnn} otro vector fila con los pesos para los valores de la capa de salida de una red neuronal profunda (*deep neural network*) y $\vec{x}_c$ un vector con la observación $\vec{x}$ ampliada con una serie de variables cruzadas.

Otro de los puntos claves de estos modelos es que el entrenamiento de la parte lineal (*Wide*) y la parte de redes neuronales (*Deep*) se realiza de manera conjunta, reajustando al mismo tiempo (por medio del algoritmo de *backpropagation*) W_{lin}, W_{deep} y los pesos y sesgos pertenecientes a la red neuronal. Este entrenamiento conjunto permitirá que a lo largo de las iteraciones se vayan corrigiendo los defectos de la parte lineal (imposibilidad de obtener relaciones no lineales entre variables) y de la parte de red neuronal (posibles sobregeneralizaciones), obteniendo así un modelo más robusto que las regresiones lineales o las redes neuronales por separado.

Si queremos aplicar en TensorFlow (versión 1.2 para Python) los modelos que hemos descrito en esta sección, podemos usar *LinearClassifier* (problemas de clasificación) o *LinearRegressor* (problemas de regresión) para modelos de regresión lineales. De manera análoga, las funciones DNNClassifier y DNNRegressor nos permiten modelar redes neuronales profundas. Además, en el caso de los modelos *Wide and Deep* tenemos otro

par de funciones, que al igual que las anteriormente mencionadas, se pueden aplicar a problemas de clasificación y de regresión respectivamente: DNNLinearCombinedClassifier y DNNRegressor. En el caso de los problemas de regresión, la fórmula que se aplica es idéntica a la escrita en 5.7, solo que en este caso prescindimos de la función sigmoide.

Por último, hay que destacar que estos modelos se han aplicado con éxito al sistema de recomendación de aplicaciones de la Google Play Store, que posee más de 1.000 millones de usuarios activos y 1 millón de aplicaciones, lo cual da idea del nivel de magnitud del conjunto de datos de entrenamiento y la cantidad de variables que componen el modelo.

6 Conclusión y trabajo futuro

6.1. Conclusión

Una vez expuesto el ecosistema de herramientas utilizadas a lo largo de las tres etapas en las que se divide el trabajo, podemos destacar la variedad de campos con los que he trabajado: aprendizaje automático, estadística, computación en la nube y análisis de datos a gran escala son algunos ejemplos. Además, he realizado un trabajo de desarrollo de códigos desde un nivel de abstracción bastante alto, por ejemplo usando la librería Keras, a otro de más bajo nivel, como el del benchmark local, programado mayoritariamente utilizando funciones nativas de TensorFlow.

Resumiendo los resultados obtenidos al utilizar como ejemplo el dataset CCF[1] podemos destacar la facilidad de la ingesta de datos con Google Cloud Storage y BigQuery, posibilitando por lo tanto poder experimentar con casi cualquier dataset que deseemos. Además, con Datalab y una serie de librerías de Python he podido realizar una completa exploración del dataset, donde se ha comprobado que las transferencias fraudulentas son un porcentaje casi residual respecto al total, o que a pesar de que este tipo de transferencias no se produjeron a lo largo del tiempo bajo un patrón determinado, muchas de ellas eran de una cantidad inferior a 3 euros.

Tras realizar un entrenamiento en Keras con una red neuronal de dos capas ocultas, seguimos probando con modelos similares en el benchmark y llegamos a la conclusión de que una red neuronal con una simple capa oculta devolvía resultados realmente buenos. Además, a partir de los resultados devueltos por el benchmark, descubrimos que técnicas como *batch normalization* permitían incluso evitar el sobreentrenamiento sobre el dataset CCF, por lo que obtuvimos resultados realmente buenos incluso sin realizar undersampling. No obstante, cuando no aplicábamos este undersampling ni técnicas que evitaran el sobreentrenamiento, se comprobó que principalmente existía un problema en la detección de transferencias fraudulentas, debido en gran medida a la desproporción de las dos clases objetivo.

En cuanto a ML Engine, los resultados no han sido muy satisfactorios, dado que han sido peores que los obtenidos en el benchmark, aún habiendo utilizado modelos *Wide and Deep* con hiperparámetros similares. Además, tampoco he podido comprobar el verdadero rendimiento de ML Engine entrenando con una gran cantidad de datos, debido a que

[1] Estos resultados, así como un análisis detallado de los mismos se puede encontrar en los capítulos 3, 4 y 5 del apéndice.

el dataset CCF solamente posee unas 285.000 observaciones. No obstante, he comprobado que si utilizamos máquinas con GPU en ML Engine los tiempos de entrenamiento se reducen drásticamente respecto a entrenamientos similares en el benchmark.

Por último, destacar que en general he desarrollado herramientas que permiten que nos olvidemos de los problemas de inestabilidad y los elevados tiempos de ejecución en R, evitado además la necesidad de aplicar algoritmos de reducción de dimensionalidad o de selección de variables (*feature engineering*) para entrenar los modelos. Los tiempos de entrenamiento y los resultados obtenidos para el dataset CCF han sido bastante satisfactorios, más aún si consideramos que hemos estado entrenado modelos con 285.000 observaciones y 30 variables, unas cifras algo elevadas para un dataset de ejemplo.

6.2. Trabajo futuro

Debido a la moderada extensión trabajo, se han ido recogiendo un buen número de propuestas para una futura expansión de la funcionalidad de las herramientas presentadas.

En primer lugar, sería interesante convertir el código desarrollado en el benchmark en un paquete de Python, para posteriormente publicarlo en *PyPI*, el repositorio oficial de paquetes de Python. De esta manera la comunidad de usuarios podría instalar y utilizar de manera más sencilla las funcionalidades del benchmark. Para ello, sería necesario definir de manera clara el conjunto de funciones que va a ofrecer nuestro benchmark. Otra de las ventajas de esta posible mejora sería la posibilidad de utilizar el benchmark en la parte de exploración de datos con Datalab (recordemos que podemos utilizar el comando *pip install* en las celdas de los notebooks para instalar paquetes Python) o incluso en la parte de entrenamiento a gran escala con ML Engine, utilizando por medio de un script Python las distintas funcionalidades que ofrecería el paquete del benchmark. Esto permitiría poder trabajar de manera remota con todas las herramientas presentadas en el trabajo, lo cual es una gran ventaja en el caso de que lo utilizaran de manera cooperativa varios miembros de un equipo de desarrollo o investigadores pertenecientes a distintas instituciones educativas.

Por otra parte, en la primera fase del proyecto, se podría intentar reemplazar los gráficos de la librería *Matplotlib* por otros que ofrezcan un mayor grado de interacción, como por ejemplo los que ofrece la librería Plotly, disponible para Python. Además, otra interesante propuesta sería la de realizar predicciones en Datalab a partir de modelos entrenados con ML Engine, apoyándonos en este caso en la facilidad de integración entre herramientas de Google Cloud. Esto nos permitiría, en el caso de que tengamos un modelo predictivo óptimo y entrenado con ML Engine, que podamos comprobar de manera eficaz si el modelo sigue funcionando para muestras de nuevos datos que hayamos recogido para el mismo problema de clasificación.

Una idea que se propuso a lo largo del desarrollo del trabajo fue la de extender la funcionalidad del benchmark para resolver otros tipos de problemas. Por ejemplo, se podría definir un benchmark para poder probar distintas topologías de redes convolucionales con el objetivo de resolver problemas de reconocimiento de imágenes, o incluso redes recurrentes (LSTMs), las cuales pueden ayudar a solucionar problemas de procesamiento natural del lenguaje. Además también sería interesante que nuestro benchmark pudiera resolver problemas de regresión (como el que se trató en Datalab para el dataset de viajes en taxi en Chicago) dado que actualmente solo está destinado a resolver problemas de clasificación binarios y multiclase.

Es interesante notar que, a diferencia de la API para los modelos *Wide and Deep*, el código desarrollado en Keras y en el benchmark únicamente admite datasets con variables de tipo númerico. Esto puede ser un problema a la hora de utilizar datasets con variables de tipo cualitativo (por ejemplo: nombres de ciudades, estaciones de metro, etc.), por lo que sería recomendable aplicar en estos códigos técnicas como *word2vec*, que permiten asignar palabras a vectores con componentes numéricas.

Por último, sería deseable seguir investigando en como integrar la API de los modelos *Wide and Deep* con otras herramientas de aprendizaje automáticos como Scikit-learn o Pandas, de manera que obtengamos tras el entrenamiento resultados más completos (como las matrices de confusión o curvas ROC que he construido para el benchmark) y satisfactorios a los obtenidos.

Glosario

accuracy Métrica que determina el porcentaje de observaciones correctamente clasificadas en un determinado problema de clasificación binario o multiclase.

AUC Métrica utilizada en problemas de clasificación binarios y que se define como el área que encierra la curva ROC. Se recomienda usar este tipo de métrica en datasets que presenten una considerable desproporción en alguna de sus clases.

benchmark Dentro del ámbito de este trabajo se denomina así al software destinado a medir la calidad de una red neuronal entrenada para resolver un problema determinado.

CCF Siglas de Credit Card Fraud, correspondientes al dataset de ejemplo utilizado en el trabajo y cuyo problema asociado consiste en poder distinguir transferencias normales de aquellas de tipo fraudulento.

CSV Siglas de *comma-separated values*, tipo de archivo que permite almacenar datasets de tipo relacional y donde cada observación se corresponde a una línea del fichero. Tal y como indican las siglas de este tipo de ficheros, los valores de cada una de estas observaciones están separados por comas.

dataset Conjunto de datos, normalmente en formato csv, dispuestos en forma de matriz. Cada una de las filas se denomina observación, mientras que las columnas se denominan variables.

deep learning Campo dentro del aprendizaje automático basado en algoritmos compuestos por una serie de capas que aplican transformaciones no lineales. El objetivo de estos algoritmos es obtener relaciones complejas entre los datos y la extracción de características con distintos niveles de abstracción. (fuente).

entrenamiento Se denomina así al proceso de ajuste de parámetros de una red neuronal utilizando un conjunto de observaciones determinadas (*training data*). El algoritmo que se suele aplicar durante el entrenamiento es un descenso de gradiente por lotes y tiene como objetivo poder generalizar y realizar predicciones precisas en otros conjuntos de datos (*test set*).

etiqueta Variable que indica la clase a la que pertenece una observación. Suele tomar valor 0 ó 1 para problemas de clasficación binarios y un número entero para los multiclase, aunque en este último caso se suele recurrir a una codificación one-hot.

Github Plataforma donde se alojan repositorios de código mediante un sistema Git de control de versiones.

hiperparámetros Conjunto de parámetros adicionales necesarios para el correcto funcionamiento de un algoritmo de aprendizaje automático (en el caso de las redes neuronales: número de capas ocultas, tasa de aprendizaje, etc.).

JSON Siglas de JavaScript Object Notation, un formato de texto con el objetivo de transmitir información independientemente de la plataforma o lenguaje de programación que lo utilice.

Kaggle Comunidad de usuarios dedicada al análisis de datos y al desarrollo de modelos predictivos, donde se organizan competiciones y se alojan datasets de todo tipo.

modelo predictivo Algoritmo de aprendizaje automático (en este trabajo, normalmente redes neuronales profundas) que junto a una serie de hiperparámetros y pesos sirve para resolver un problema de clasificación binario o multiclase.

Numpy Librería de Python que permite operar de forma eficiente sobre vectores y matrices (Numpy arrays).

observación Cada una de las filas presentes en un dataset.

one-hot Tipo de codificación mediante vectores dispersos para problemas de clasificación multiclase.

PCA Siglas de *Principal Component Analysis*, una técnica de reducción de dimensionalidad. Esto se traduce en una transformación de las observaciones de nuestro dataset en otras con un número menor de variables..

problema de clasificación binario Problema que consiste en predecir la clase correcta ($t_i \in \{0, 1\}$) para un conjunto determinado de observaciones con k variables: $x_i = (x_{i1}, x_{i2}, \ldots, x_{ik})$. Un ejemplo de este tipo de problemas puede ser la predicción de si una persona es alérgica o no a un determinado producto alimenticio.

problema de clasificación multiclase Extensión del problema de clasificación binario con el objetivo de poder realizar predicciones para m clases disjuntas. En este caso las etiquetas de la clase objetivo suelen poseer codificación one-hot: $t_i = (t_{i1}, t_{i2}, \ldots, t_{im})$.

Python Lenguaje de programación de propósito general, creado en 1991 y caracterizado por ser de bastante alto nivel.

R Lenguaje de programación surgido en 1993 y orientado al campo de la estadística.

red neuronal Normalmente el término se refiere a un Perceptrón Multicapa con propagación hacia delante (*feed-forward*) y con capas completamente conectadas (*fully-connected layers*). En el caso de que este tipo de Perceptrón tenga dos o más capas ocultas, diremos que se trata de una red neuronal profunda.

ROC La curva ROC (Receiver Characteristic Curve) representa de forma gráfica tasas de falsos positivos (FPR o *false positive rate*) frente a tasas de verdaderos positivos (TPR o *true positive rate*). Estas tasas se obtienen variando el *threshold* que regula las predicciones finales en un modelo dado.

SDK Google Cloud Siglas pertenecientes a *Software Development Kit*. Este SDK se compone de un conjunto de utilidades que permiten comunicarse con herramientas de Google Cloud a través de la línea de comandos. El SDK de Google Cloud está disponible para sistemas operativos Windows, Mac y Linux.

softmax Función de $\mathbb{R}^m$ en $\mathbb{R}^m$ que transforma valores de las neuronas de una capa de salida en probabilidades con valores entre 0 y 1.

TensorFlow Librería destinada a la resolución de problemas con técnicas de aprendizaje automático por medio de grafos de cómputo.

undersampling Técnica consistente en tomar un subconjunto de datos pertenecientes a un dataset con el objetivo de equilibrar la distribución de las clases frente a posibles desbalanceos.

Wide and Deep Tipo de modelos que combinan, entrenando de forma conjunta, regresiones lineales y redes neuronales profundas, obteniendo de esta manera las ventajas de cada uno de estos.

Bibliografía

[1] Algoritmos de búsqueda en el problema de búsqueda de hiperparámetros. https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)#Optimization_algorithms. Consultado: 2017-08-20.

[2] API de modelos Wide and Deep en TensorFlow. https://www.tensorflow.org/api_docs/python/tf/contrib/learn/DNNLinearCombinedClassifier. Consultado: 2017-08-20.

[3] Autenticación en Google Cloud Storage. https://cloud.google.com/storage/docs/authentication. Consultado: 2017-08-19.

[4] Caso de uso con redes neuronales en Netflix. https://goo.gl/TQ8VCa. Consultado: 2017-08-19.

[5] Colas multithreading en TensorFlow. https://www.tensorflow.org/programmers_guide/threading_and_queues. Consultado: 2017-08-19.

[6] Comando de entrenamiento para ML Engine en el SDK de Google Cloud. https://cloud.google.com/sdk/gcloud/reference/ml-engine/jobs/submit/training. Consultado: 2017-09-04.

[7] Comparativa de rendimiento de CPU y GPU en TensorFlow. https://medium.com/@erikhallstrm/hello-world-tensorflow-649b15aed18c. Consultado: 2017-08-20.

[8] Conceptos básicos de Google Cloud Platform. https://cloud.google.com/docs/overview/. Consultado: 2017-07-09.

[9] Configuración de hiperparámetros en ML Engine. https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs#hyperparameterspec. Consultado: 2017-08-20.

[10] Consideraciones a la hora de elegir una máquina virtual para Datalab. https://cloud.google.com/datalab/docs/how-to/machine-type. Consultado: 2017-07-10.

[11] Corrector de errores gramaticales con redes neuronales recurrentes. http://atpaino.com/2017/01/03/deep-text-correcter.html. Consultado: 2017-08-19.

[12] Costes asociados a Datalab. https://cloud.google.com/datalab/docs/resources/pricing. Consultado: 2017-07-10.

[13] Costes de Big Query. https://cloud.google.com/bigquery/pricing. Consultado: 2017-07-11.

[14] Costes de Cloud Storage. https://cloud.google.com/storage/pricing. Consultado: 2017-07-11.

[15] Costes de Google Compute Engine. https://cloud.google.com/compute/pricing. Consultado: 2017-07-10.

[16] Costes de ML Engine. https://cloud.google.com/ml-engine/pricing. Consultado: 2017-07-24.

[17] Credit Card Fraud Detection Dataset. https://www.kaggle.com/dalpozz/creditcardfraud. Consultado: 2017-09-12.

[18] Curso cs231n Stanford - módulo 1. http://cs231n.github.io/. Consultado: 2017-07-28.

[19] Dataset de viajes en taxi en Chicago - BigQuery. https://cloud.google.com/bigquery/public-data/chicago-taxi. Consultado: 2017-07-19.

[20] Datasets públicos en BigQuery. https://cloud.google.com/bigquery/public-data/. Consultado: 2017-08-19.

[21] Descripción general de Datalab. https://cloud.google.com/datalab/. Consultado: 2017-07-09.

[22] Descripción general del ajuste de hiperparámetros en ML Engine. https://cloud.google.com/ml-engine/docs/concepts/hyperparameter-tuning-overview. Consultado: 2017-07-23.

[23] Dificultades de R con el multithreading. https://stackoverflow.com/questions/10835122/multithreading-with-r. Consultado: 2017-08-19.

[24] Documentación de Keras. https://keras.io/. Consultado: 2017-07-22.

[25] Documentación de Scikit-learn. http://scikit-learn.org/stable/documentation.html. Consultado: 2017-08-19.

[26] Ejemplo con el dataset Census Income para ML Engine. https://github.com/GoogleCloudPlatform/cloudml-samples/tree/master/census. Consultado: 2017-07-23.

[27] FAQ de Amazon Machine Learning. https://aws.amazon.com/es/machine-learning/faqs/. Consultado: 2017-08-19.

[28] Google Cloud Platform Console. https://console.cloud.google.com. Consultado: 2017-08-19.

[29] Google Cloud Storage. https://cloud.google.com/storage/?hl=es. Consultado: 2017-07-19.

[30] Guía rápida de Datalab. https://cloud.google.com/datalab/docs/quickstarts. Consultado: 2017-07-09.

[31] Instalación de Jupyter Notebook en Amazon Web Services. https://aws.amazon.com/es/blogs/big-data/running-jupyter-notebook-and-jupyterhub-on-amazon-emr/. Consultado: 2017-09-04.

[32] Instalación de Jupyter Notebook en Google Cloud Platform. https://cloud.google.com/dataproc/docs/tutorials/jupyter-notebookclusters de Google Cloud Platform. Consultado: 2017-08-19.

[33] Instalación de Jupyter Notebook en un servidor. http://jupyter-notebook.readthedocs.io/en/latest/public_server.html. Consultado: 2017-08-19.

[34] Integración de Keras en TensorFlow. https://github.com/fchollet/keras/issues/5050. Consultado: 2017-08-19.

[35] Introducción a los modelos lineales en TensorFlow. https://www.tensorflow.org/tutorials/linear. Consultado: 2017-08-19.

[36] Introducción a TensorBoard. https://www.tensorflow.org/get_started/graph_viz. Consultado: 2017-08-19.

[37] Kaggle. https://www.kaggle.com. Consultado: 2017-08-19.

[38] Librerías para la api de Google Cloud Storage. https://cloud.google.com/storage/docs/reference/libraries. Consultado: 2017-08-19.

[39] Lista de tutoriales oficiales de TensorFlow. https://www.tensorflow.org/get_started/. Consultado: 2017-08-20.

[40] Lista de versiones de ML Engine. https://cloud.google.com/ml-engine/docs/resources/release-notes. Consultado: 2017-08-20.

[41] Medición del rendimiento en aprendizaje automático. https://www.cs.cornell.edu/courses/cs578/2003fa/performance_measures.pdf. Consultado: 2017-08-20.

[42] Notebook deep learning - Aurélien Géron. https://github.com/ageron/handson-ml/blob/master/11_deep_learning.ipynb. Consultado: 2017-08-20.

[43] Opciones de configuración de instancias de Cloud Datalab. https://cloud.google.com/datalab/docs/how-to/lifecycle. Consultado: 2017-07-10.

[44] Optimizaciones en el algoritmo de descenso de gradiente. http://ruder.io/optimizing-gradient-descent/index.html. Consultado: 2017-08-19.

[45] ¿Por qué elegir Google Cloud Platform? https://cloud.google.com/why-google/. Consultado: 2017-07-09.

[46] Proceso de entrenamiento en ML Engine. https://cloud.google.com/ml-engine/docs/concepts/training-overview. Consultado: 2017-08-20.

[47] Programa de ayudas en Google Cloud Platform para instituciones educativas. https://cloud.google.com/edu/. Consultado: 2017-07-09.

[48] Prueba gratuita de Google Cloud Platform. https://cloud.google.com/free/. Consultado: 2017-07-09.

[49] ¿Qué es word embedding en deep learning? https://www.quora.com/What-is-word-embedding-in-deep-learning. Consultado: 2017-08-20.

[50] Repositorio de Datalab en Github. https://github.com/googledatalab/datalab. Consultado: 2017-07-09.

[51] Repositorio de github con ejemplos de TensorFlow. https://github.com/aymericdamien/TensorFlow-Examples. Consultado: 2017-08-20.

[52] Repositorio de notebooks de ejemplo de Datalab en Github. https://github.com/googledatalab/notebooks. Consultado: 2017-07-09.

[53] Repositorio github con código realizado para la asignatura de geometría computacional. https://github.com/zentonllo/gcom. Consultado: 2017-08-20.

[54] Repositorio Github de un proyecto para reconocimiento de señales de tráfico. https://github.com/JamesLuoau/Traffic-Sign-Recognition-with-Deep-Learning-CNN. Consultado: 2017-08-19.

[55] Repositorio Github del trabajo. https://github.com/zentonllo/tfg-tensorflow. Consultado: 2017-09-12.

[56] Repositorios populares de aprendizaje automático en Github. https://github.com/showcases/machine-learning. Consultado: 2017-04-16.

[57] Restauración de modelos de TensorFlow. https://www.tensorflow.org/programmers_guide/saved_model#restoring_variables. Consultado: 2017-09-04.

[58] Tipos de máquinas virtuales para ejecutar Datalab. https://cloud.google.com/compute/docs/machine-types. Consultado: 2017-07-10.

[59] Tutorial de modelos lineales en TensorFlow. https://www.tensorflow.org/tutorials/wide. Consultado: 2017-08-19.

[60] Tutorial de modelos Wide and Deep en TensorFlow. https://www.tensorflow.org/tutorials/wide_and_deep. Consultado: 2017-08-19.

[61] Tutoriales y documentación de ML Engine. https://cloud.google.com/ml-engine/docs/how-tos/. Consultado: 2017-08-20.

[62] Ventajas del módulo gfile de TensorFlow. https://stackoverflow.com/questions/42922948/why-use-tensorflow-gfile-for-file-i-o/. Consultado: 2017-08-19.

[63] Visualización del grafo de cómputo en TensorBoard. https://www.tensorflow.org/get_started/summaries_and_tensorboard. Consultado: 2017-08-19.

[64] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR abs/1603.04467* (2016).

[65] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[66] Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. Multi-digit number recognition from street view imagery using deep convolutional neural networks. In *ICLR2014* (2014).

[67] Cheng, H., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. Wide & deep learning for recommender systems. *CoRR abs/1606.07792* (2016).

[68] Dal Pozzolo, A., Caelen, O., Johnson, R. A., and Bontempi, G. Calibrating Probability with Undersampling for Unbalanced Classification.

[69] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[70] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 1 ed. O'Reilly Media, 3 2017.

[71] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR abs/1502.03167* (2015).

[72] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. Large-scale video classification with convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition* (Washington, DC, USA, 2014), CVPR '14, IEEE Computer Society, pp. 1725–1732.

[73] Lecun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521*, 7553 (5 2015), 436–444.

[74] Sato, K. An inside look at Google BigQuery. Tech. rep., Google Cloud Solutions team, 2012.

[75] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15* (2014), 1929–1958.

Apéndice

1. Estructura del repositorio Github

El repositorio Github del proyecto, el cual posee una licencia MIT, se puede encontrar en la bibliografía adjunta [55]. El repositorio contiene código y resultados de cada una de las tres partes del trabajo.

En la carpeta *cloud* encontraremos otras dos carpetas: *datalab* y *mlengine*, que contienen el código y los resultados de los capítulos 3 y 5 respectivamente.

La carpeta *datalab* contiene los dos notebooks (*Keras-GCS.ipynb* y *Keras-BQ.ipynb*) utilizados para mostrar distintas funcionalidades de Datalab, tal y como se detalla en el capítulo 3. También podremos encontrar *notebooks_ejemplo*, donde se guardan los archivos de ejemplo proporcionados por Datalab que han servido de guía para desarrollar los dos notebooks que mencionábamos. Estos notebooks de muestra se pueden encontrar en un repositorio de Github de Datalab [52], aunque se han incluido aquí (pues poseen licencia Apache) por comodidad. Por último, en *resultados* se encuentran comprimidas tres carpetas con logs de modelos entrenados por Keras, cuyo análisis puede encontrarse en el apéndice 3.

En *mlengine* encontraremos una carpeta *trainer* con los códigos utilizados por ML Engine para realizar el entrenamiento del dataset CCF, así como *scriptml.sh*, el script Bash que ejecuta todo el proceso de entrenamiento. También se encuentra *split.R*, un script desarrollado en el lenguaje de programación R que realiza el proceso de undersampling y permite partir el dataset CCF en un conjunto de entrenamiento y otro de validación, ambos necesarios para el proceso de ingesta en ML Engine. El archivo *hptuning_config.yaml* (obtenido de [26] y ligeramente retocado) es el archivo que utilizamos para el ajuste de hiperparámetros en ML Engine. Dentro de *resultados* se encuentran los archivos generados por ML Engine tras el entrenamiento y el ajuste de hiperparámetros realizado sobre el dataset CCF. El análisis de estos archivos se ve de manera más detallada en el apéndice 5.

Dentro de *benchmark* se ha guardado el código desarrollado para el capítulo 4, así como diversos scripts Python que prueban determinados modelos de redes neuronales usando Keras (*keras_tests.py*) o regresiones logísticas por medio de TensorFlow (*logistic_regression.py*). También existe una carpeta denominada *resultados* y que contiene los entrenamientos que se analizan en el apéndice 4.

2. Configuración de Datalab

En esta sección se procederá a explicar como crear una máquina de Google Compute Engine e instalar Datalab sobre ella. Para ello utilizaremos la Cloud Shell, pues ya trae instalado el SDK de Google Cloud con el comando *datalab*, necesario para crear y eliminar instancias de Datalab. Las fuentes consultadas para esta sección han sido extraídas de las páginas de documentación de Datalab [30, 43].

Tras entrar en la Cloud Platform Console, abrimos la shell (botón más a la izquierda de la esquina superior derecha) y escribimos el comando:

```
$ gcloud projects list
```

Se muestran los proyectos vinculados a nuestra cuenta personal, detallando el nombre y el número del proyecto, así como el ID del mismo. Este último lo necesitamos para indicar a la consola que estamos trabajando sobre el proyecto indicado:

```
$ gcloud config set core/project <project-id>
```

Donde el *project-id* lo hemos obtenido a partir del comando previo. Ahora estamos en condiciones de crear la instancia de Datalab sobre una maquina virtual de Google Cloud (denominada Compute Engine). Es importante destacar que el nombre de la instancia debe comenzar con una letra minúscula:

```
$ datalab create <nombre-instancia> \
--no-connect \
--no-backups \
--no-create-repository \
--disk-size-gb TAM_DISCO \
--machine-type TIPO_MAQUINA
```

Cuando ejecutemos el comando, se nos pedirá elegir una zona (podemos elegir cualquiera sin ninguna restricción) para alojar la máquina virtual que estamos creando. Nótese que el primer flag se ha utilizado para evitar que tras la creación se realice la conexión con la máquina virtual sobre la que se monta Datalab. Esto se ha hecho con el propósito de presentar más adelante los comandos de conexión, desconexión y eliminación de la máquina virtual. El segundo flag evitará que se creen backups periódicos de los noteboooks en Storage, dado que no es crítico para este trabajo. El tercer flag evita que se cree un repositorio remoto, el cual no vamos a utilizar, en otro servicio de Google Cloud denominado Cloud Source Repositories. Por último, con *TAM_DISCO* indicaremos en Gigabytes el tamaño del disco duro que queremos para la máquina virtual, mientras que *TIPO_MAQUINA* indica que tipo de máquina virtual que queremos

utilizar sobre Datalab. Los distintos tipos de máquinas virtuales se pueden encontrar en la bibliografía [58]. Estos dos argumentos repercuten directamente en el precio, tal y como se muestra en el apartado 3.5.

Además, se generará de manera automática una red interna en el proyecto de Cloud (denominada *datalab-network*) así como una regla en el cortafuegos de esa red de manera que se permitan conexiones ssh a la máquina virtual, la cual esta conectada a la red interna recién creada. Esta regla del cortafuegos se puede consultar a través del buscador de la Cloud Console en la sección *Reglas de cortafuegos*. Podemos ver que la regla creada permite conexiones entrantes por el puerto tcp número 22, el cual se utiliza para conexiones *ssh*.

A continuación nos conectaremos a la instancia creada por medio del comando:

```
$ datalab connect <nombre-instancia>
```

En este punto se nos pedirá introducir una clave ssh que usaremos siempre que queramos conectarnos a la máquina virtual. La conexión que acabamos de realizar permanecerá activa siempre y cuando el comando siga ejecutándose en la consola. Para acceder a la interfaz gráfica de Datalab dentro de la máquina virtual, clicamos en el botón del extremo superior izquierdo de la consola y elegimos *Cambiar Puerto* > *Puerto 8081*. Se abrirá una pestaña en nuestro navegador y podremos empezar a desarrollar código en los *notebooks*.

Por último, cuando queramos interrumpir el comando *datalab connect* escribiremos:

```
$ datalab stop <nombre-instancia>
```

Esto permitirá detener la máquina virtual y así poder evitar costes derivados de su ejecución. Por otro lado, para eliminar de forma definitiva la instancia de Datalab junto a su disco de almacenamiento (si no, el disco por defecto no se elimina y se seguiría incurriendo en costes adicionales):

```
$ datalab delete --delete-disk <nombre-instancia>
```

Para acabar esta sección tenemos que destacar una limitación que presentan las instancias de Datalab: cada una de ellas solo permite una conexión al mismo tiempo. Es decir, varios usuarios asociados a un proyecto de Google Cloud no pueden conectarse al mismo tiempo a una instancia de Datalab.

3. Análisis de logs de Keras

En esta parte realizaremos un análisis de tres logs obtenidos al realizar entrenamientos de redes neuronales por medio de la librería Keras en la primera fase de nuestro trabajo (correspondiente al capítulo 3). Estos logs pueden encontrarse en el repositorio Github del proyecto [55] y han sido generados a partir del notebook *Keras-GCS.ipynb*.

En los tres logs se ha utilizado la misma estructura de red neuronal, con dos capas ocultas de 20 y 15 neuronas respectivamente, aplicando batch normalization y una tasa de dropout de 0.5. El dataset utilizado ha sido CCF y se ha realizado un undersampling previo para balancear las dos clases: transferencias fraudulentas y normales. Cada uno de los tres logs son el resultado de realizar un entrenamiento de 500 iteraciones (epochs) con un batch size diferente. El resumen de estos entrenamientos puede verse en la tabla 1.

Tabla 1: Resumen de entrenamientos con redes neuronales en Keras

Fuente: Archivos results.txt en cloud/datalab/resultados [55]

Archivo	**Iteraciones**	**Tasa de dropout**	**Batch size**	**Función coste**	**Accuracy**	**AUC**
log1.zip	500	0.5	500	0.327	89.90	96.95
log2.zip	500	0.5	50	0.198	93.94	97.37
log3.zip	500	0.5	20	0.180	92.93	98.49

Por un lado tenemos los archivos *training.log*, que poseen estructura de archivo csv con las siguientes columnas: número de iteración (*epoch*), función de coste y *accuracy* para los datos de entrenamiento y de validación respectivamente (*acc*, *loss*, *acc_val*, *loss_val*). Estos archivos pueden sernos útiles para pintar gráficas con las librerías adecuadas (por ejemplo, Matplotlib).

Además también tenemos archivos de TensorBoard, que nos permiten comparar de forma interactiva (y simultánea) como evolucionan en el entrenamiento las métricas que comentábamos previamente en el archivo csv. Es necesario recordar que también es posible visualizar los gráfos de cómputo en la pestaña *graphs*, pero debido a la capa de complejidad que presenta Keras (que es quien se encarga de generarlos), no son muy intuitivos de explicar, por lo que nos centraremos en el análisis de las métricas durante el entrenamiento. Por ejemplo, en la figura 1 se puede comparar la evolución del porcentaje de observaciones correctamente clasificadas (accuracy) para el conjunto de entrenamiento y de validación respectivamente. Aunque la leyenda se puede consultar a través de la aplicación web de TensorBoard, por comodidad señalamos que el primer entrenamiento

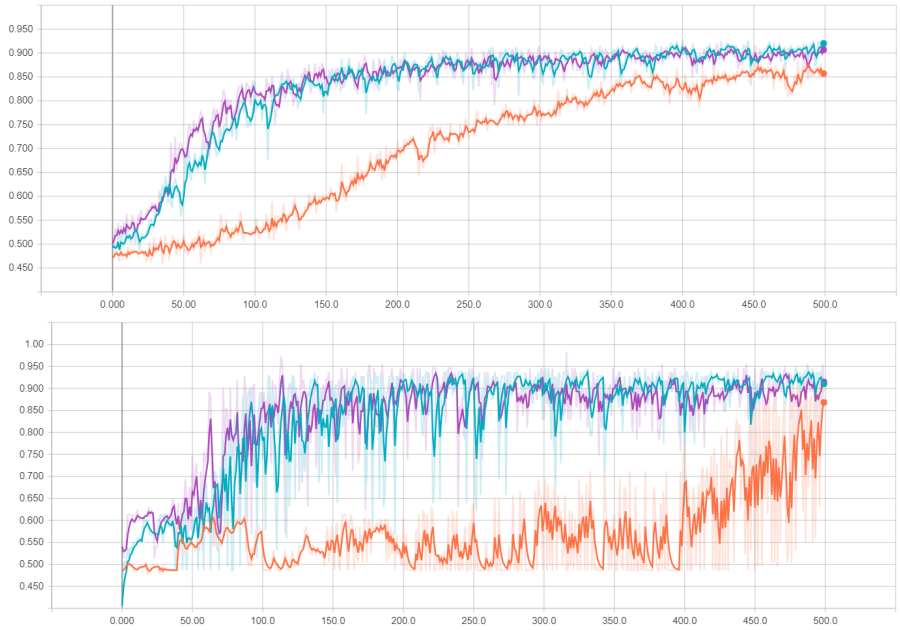


Figura 1: Predicciones correctas sobre el conjunto de entrenamiento y de validación respectivamente

Fuente: Tensorboard (logs de Keras) [55]

es el de color naranja, mientras que el segundo posee un color verde claro, y el tercero es granate.

De esta figura podemos extraer que el primer entrenamiento tarda en torno a 325 iteraciones en alcanzar el 80 % de instancias correctamente clasificadas, mientras que los otros dos entrenamientos alcanzaron esta cifra en tan solo 100 iteraciones o epochs. No obstante en el gráfico para los datos de validación existen unas series de oscilaciones a lo largo del gráfico. Este hecho se debe a que el entrenamiento se realiza sobre el conjunto de datos de entrenamiento, por lo que al actualizar los pesos de la red neuronal tras un epoch, no es seguro que esto lleve a una mejora en el porcentaje de instancias clasificadas correctamente en el conjunto de validación. Debemos destacar que la misión de este conjunto es la de supervisar el entrenamiento a lo largo de un número significativo de epochs. Si miramos la gráfica para los datos de validación vemos como el entrenamiento se realiza de forma correcta y no existe sobreentrenamiento, pues a partir de las 400 iteraciones los resultados parecen estabilizarse.

En TensorBoard también podemos visualizar la evolución de la función de coste para estos dos conjuntos de datos. La función de coste utilizada aquí es *cross-entropy* (veáse ecuación 4.8). Como vemos en la figura 2, las conclusiones extraídas para la gráfica anterior son coherentes con esta otra: el primer entrenamiento va minimizando la función

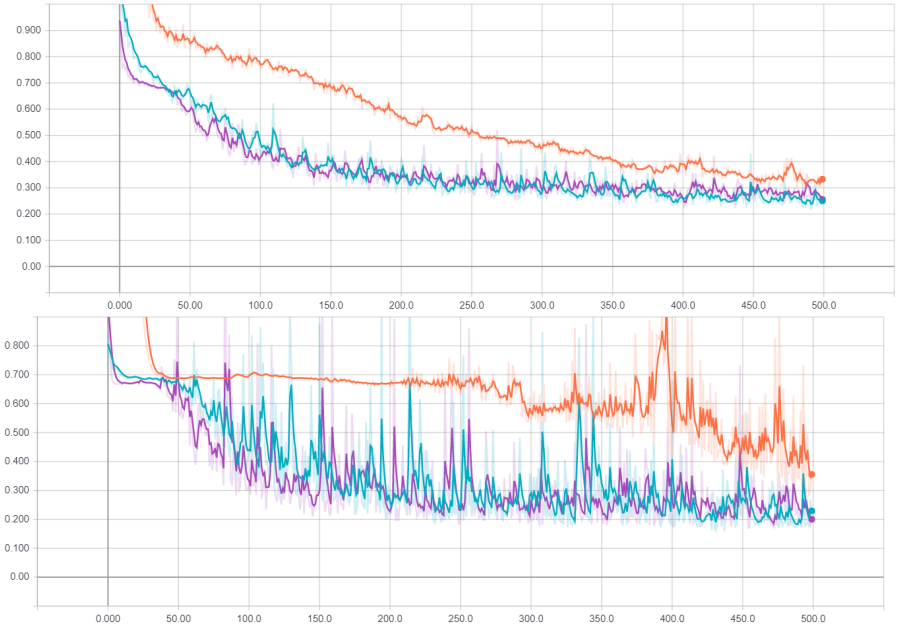


Figura 2: Función de coste para el conjunto de entrenamiento y de validación respectivamente

Fuente: Tensorboard (logs de Keras)[55]

de coste de manera más lenta para el conjunto de datos de entrenamiento, mientras que las oscilaciones siguen produciéndose para el conjunto de datos de validación. Nótese como la función de coste permanece casi constante durante las primeras iteraciones del entrenamiento para los datos de validación, coincidiendo así con el estancamiento que puede verse también en la gráfica del porcentaje de instancias clasificadas de forma correcta.

Por último, nos centraremos en los diferentes archivos *results.txt*. Al principio de este archivo de texto se encuentra en formato JSON el modelo de red neuronal utilizado por Keras. Este JSON es útil en caso de que queramos reutilizar este modelo en un futuro. Como ejemplo, se ha incluido en el apéndice 6 el JSON del archivo *results.txt* perteneciente al log número 3 (tercer entrenamiento) correctamente formateado. A continuación tenemos la descripción de los hiperparámetros usados para la red neuronal y un resumen del modelo (idéntico al que mostramos en la figura 3.7) realizado por Keras. Se puede observar como en cada uno de los tres entrenamientos realizados existen 1.037 parámetros que se van ajustando durante el entrenamiento. Al final del archivo están los resultados del modelo entrenado al aplicar los datos del conjunto de test: función de coste, accuracy y AUC.

Analizando estos resultados podemos afirmar que aún a pesar de haber realizado undersampling (mismo porcentaje de transferencias fraudulentas y normales, unas 800 en total), el hiperparámetro de *batch size* tiene una fuerte importancia. Para el primer entrenamiento hemos elegido un batch size realmente alto, 500, para la cantidad de datos totales, lo que provoca que a nuestra red neuronal le cueste *adaptarse* al conjunto de datos de entrenamiento. Sin embargo, al reducir el tamaño de lote o batch size, se puede ver como los resultados mejoran de forma notoria, logrando unos altos porcentajes de accuracy en unas 150-200 iteraciones. Todo esto se produce porque el hiperparámetro de batch size es el encargado de regular como se adapta el entrenamiento a las peculiaridades del conjunto de entrenamiento (ver ecuaciones 4.10 y 4.11) Con estas redes obtenemos en esta primera etapa unos resultados muy interesantes, con un AUC superior a 95. Nótese que en este caso tiene sentido considerar la métrica *accuracy* debido al undersampling realizado para los datos de entrenamiento, validación y test. Comentar también que el hecho de que el valor AUC sea superior a *accuracy* se debe a que, a pesar de que estamos en este caso con un dataset balanceado, el valor AUC se encarga también de medir la "*robustez*" del modelo variando los *thresholds* para clasificar las instancias, tal y como vimos en 4.2.2.

4. Análisis de logs del benchmark

En esta sección realizaremos un análisis de dos herramientas del benchmark que he programado para este trabajo: por un lado realizaremos un ajuste de hiperparámetros (*tuning_hyperparams.py*) y luego realizaremos pruebas más exhaustivas utilizando el Playground (*playground.py*). Todos estos resultados se pueden encontrar en el repositorio Github del trabajo [55]. Nótese que en esta ocasión realizaremos los entrenamientos sobre el dataset CCF original sin realizar undersampling, lo cual es un verdadero reto, pues únicamente el 0.17 % de las transferencias son fraudulentas, por lo que no es tarea fácil que nuestra red neuronal pueda aprender a reconocer este tipo de transferencias. El principal problema radica en un posible sobreentrenamiento que en última instancia lleve a nuestros modelos a clasificar con gran probabilidad cualquier transferencia como no fraudulenta.

En primer lugar procederemos a realizar un ajuste de hiperparámetros a partir de los resultados obtenidos en el apéndice 3. Por ello, vamos a probar con una serie de entrenamientos (6 en total) de 100 epochs con *tasa de aprendizaje* de 0.001, *batch size* de tamaño 100, *batch normalization* y optimizador *Adam*. Los hiperparámetros cuya combinación probaremos serán las funciones de activación (tipo ELU y ReLU, veáse figura 4.4 para más información) y la lista de capas y neuronas ocultas: hemos probado con una red de 15 y 5 neuronas respectivamente y otras dos con una única capa oculta de 10 y 3 neuronas respectivamente. Destacar que no se ha utilizado dropout ni regularización L1 o L2 durante estos entrenamientos.

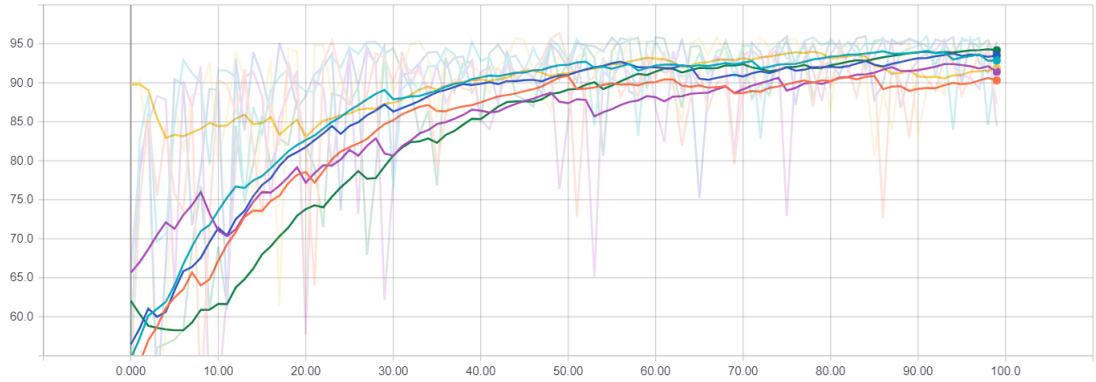


Figura 3: Valores AUC sobre el conjunto de validación para los 6 modelos entrenados

Fuente: Tensorboard (benchmark/resultados/hyptuning.zip) [55]

Conociendo que los resultados en el apéndice 3 fueron realmente buenos con dos capas ocultas, en esta fase de ajuste de hiperparámetros he tratado de reducir el número de neuronas y de capas ocultas para tratar de comprobar si los resultados siguen manteniéndose. Debido a la desproporción entre los dos tipos de transferencias en el dataset

CCF utilizado para estos entrenamientos, solamente tomaremos AUC como métrica de referencia. Por ello, haciendo uso de TensorBoard podemos ver de manera resumida como evoluciona el entrenamiento midiendo el AUC sobre el conjunto de datos de validación (veáse figura 3). Debido a las moderadas oscilaciones de la gráfica[1], se ha optado corregirlas con una media exponencial con un parámetro *smoothing* de 0.9 que he elegido en TensorBoard.

A primera vista, los resultados son realmente similares para los 6 modelos, destacando sobre todo el modelo 6, correspondiente a la línea amarilla, que desde las primeras iteraciones obtiene un valor AUC bastante alto, aunque en posteriores iteraciones es "alcanzado" por el resto de modelos.

Es también interesante ver como la función de coste (*cross-entropy*) sufre también oscilaciones en los distintos modelos. Por ejemplo en el modelo 1 (figura 4) estos picos son bastante claros. El porqué de este comportamiento se debe a que el desbalanceo del dataset CCF provoca que la actualización de pesos en alguna iteración pueda estar realmente sesgada hacia las transferencias normales (que son mayoritarias), por lo que de esta manera los pesos no tendrán en cuenta a la transferencias fraudulentas y no podremos predecirlas de forma correcta. Este problema a la hora de predecir este tipo de transferencias provoca un aumento (en algunos casos bastante importante) de la función de coste.

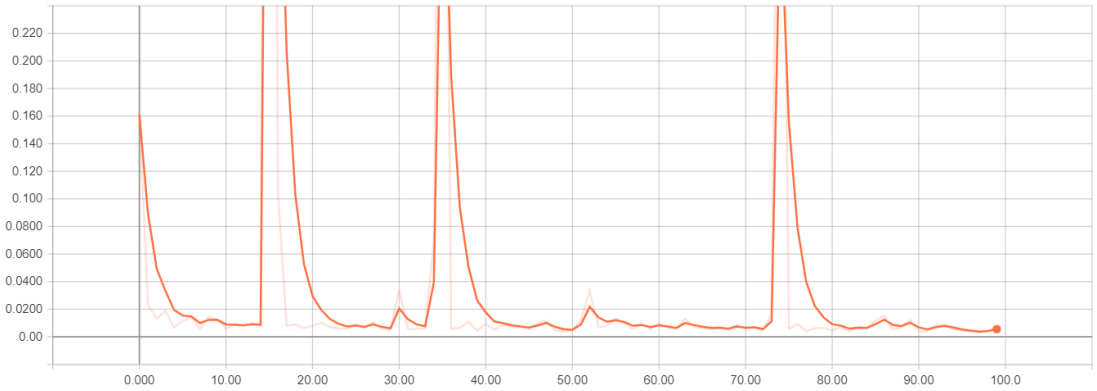


Figura 4: Evolución de la función de coste durante el entrenamiento para el modelo 1

Fuente: Tensorboard (benchmark/resultados/hyptuning.zip) [55]

También queremos destacar en este análisis la posibilidad de interactuar con el gráfo de cómputo creado en TensorFlow para cada uno de los 6 modelos generados. Por ejemplo,

[1] En este caso las oscilaciones en el gráfico de datos de validación son más grandes de lo normal debido al hecho de que estamos en ante un dataset realmente desbalanceado, y donde el entrenamiento está "dominado" en su mayoría por transferencias no fraudulentas.

en la figura 5 se muestra una captura del gráfo de cómputo del modelo 1, donde se pueden advertir en las distintas cajas como la red neuronal creada (*dnn*) posee dos capas ocultas: *hidden1* y *hidden2*. Este grafo es bastante más sencillo que el creado por Keras, debido al hecho de que lo he creado desde cero por medio de los scripts *dnn_binary.py* y *dnn_multiclass.py*.

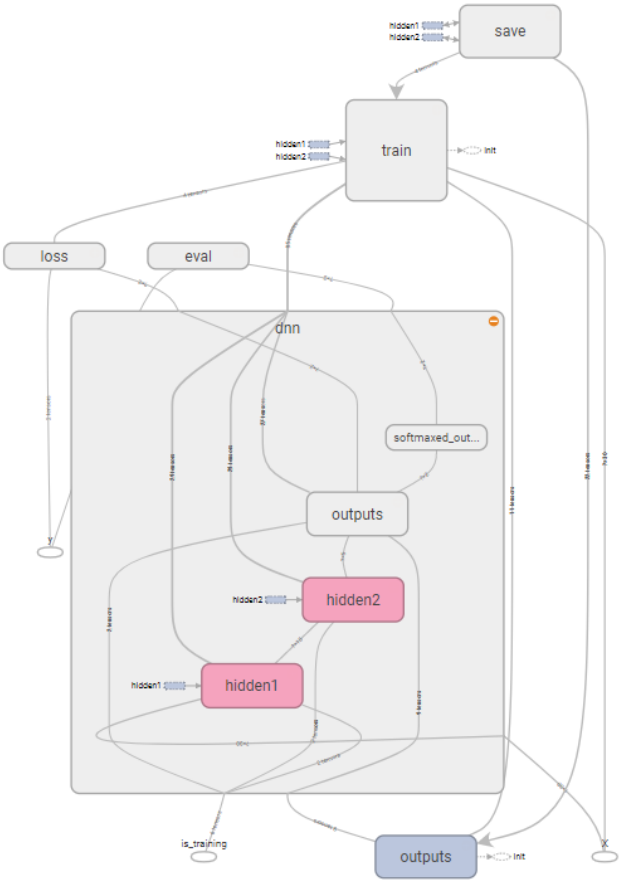


Figura 5: Grafo de cómputo en TensorFlow para el modelo 1

Fuente: Tensorboard (benchmark/resultados/hyptuning.zip) [55]

En la tabla 2 encontramos un resumen de los 6 modelos entrenados, obtenido a partir los archivos *tuning_results.txt*, donde por tiempo y resultados (bastante ajustados todo ellos) el modelo número 6 se convierte en el óptimo. Destacar de nuevo como las redes neuronales que hemos entrenado en el benchmark ofrecen unos resultados realmente buenos a pesar de reducir su complejidad (menos capas y neuronas ocultas) respecto a las entrenadas con Keras. Sin embargo, un hecho bastante sorprendente es que estas redes, a pesar de ser entrenadas bajo un dataset realmente desproporcionado, devuelvan

Tabla 2: Resumen del ajuste de hiperparámetros en el benchmark

Fuente: benchmark/resultados/hyptuning.zip [55]

Modelo	**Iteraciones**	**Capas ocultas**	**Función act.**	**Tiempo entrenamiento**	**AUC test**
1	100	[15,5]	ReLU	6 min 37s	98.86
2	100	[15,5]	ELU	6 min 50s	98.99
3	100	[10]	ReLU	4 min 48s	98.94
4	100	[10]	ELU	4 min 44s	98.82
5	100	[3]	ReLU	4 min 35s	98.91
6	100	[3]	ELU	4 min 32s	99.17

unos resultados tan buenos sin caer en sobreentrenamientos u overfitting[2]. Es posible que todo esto se deba al uso de batch normalization, pues esta técnica aporta una componente de regularización, tal y como señalan sus autores en [71]. También queremos hacer notar que a excepción de los modelos 3 y 4, el resto de modelos que utilizan la función de activación ELU son ligeramente mejores que aquellos con función ReLU.

Todos los resultados relativos al ajuste de hiperparámetros en este ejemplo se encuentran en el archivo *hyptuning.zip* del repositorio Github [55].

Ahora bien, utilizando el modelo 6 realizaremos sobre él un análisis más exhaustivo gracias al archivo *playground.py* del benchmark. Uno de los elementos más interesantes generados por este script es la matriz de confusión, que permite conocer de manera específica como han sido las predicciones realizadas. En la figura 6 mostramos la matriz de confusión para el modelo 6 utilizando los datos de test. En resumen, prácticamente la totalidad de las transferencias normales son correctamente clasificadas (existe un número residual de falsos positivos) mientras que el 77 % de las transferencias fraudulentas son clasificadas correctamente. Aquí existe un mayor número de falsos negativos (es decir, transferencias que hemos clasificado como normales pero son de tipo fraudulento), lo cual es lógico debido a que nuestra red neuronal debe lidiar con el problema del desbalanceo de clases.

No obstante, si hemos dicho que el valor AUC se ocupaba de verificar que estábamos clasificando correctamente observaciones de las dos clases objetivo, ¿por qué tenemos en este ejemplo casi un valor AUC del 100 %? Esto se responde gracias con el concepto de *threshold*. Supongamos una red neuronal con dos neuronas de salida como la de la figura 4.12, que se encargan de predecir si una transferencia es fraudulenta o no. Nuestra matriz de confusión está midiendo resultados con un *threshold* de 0.5, esto es, todas aquellas

[2]El *overfitting* se descarta al observar que la función de coste es decreciente (a lo largo del entrenamiento) sobre los datos de validación y que los valores AUC sobre el conjunto de test siguen siendo bastante altos.

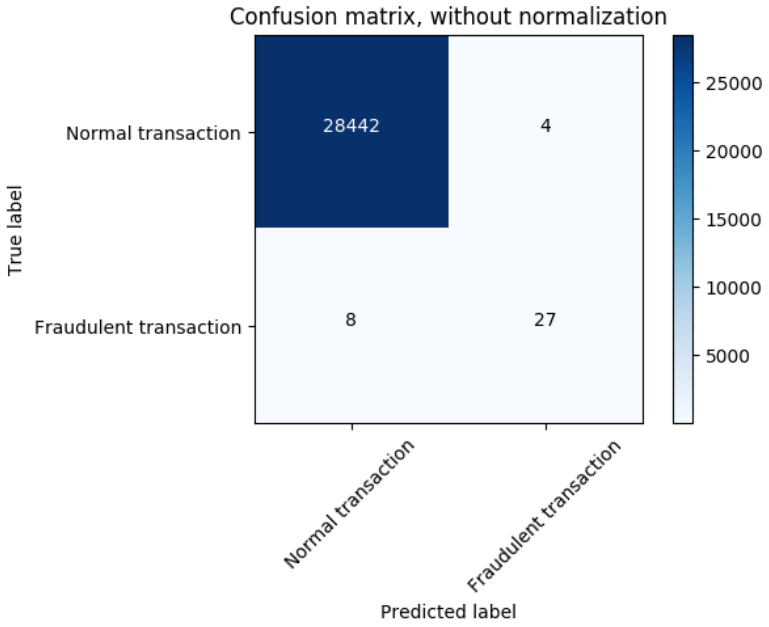


Figura 6: Matriz de confusión para el modelo 6

Fuente: Playground (benchmark/resultados/model6playground.zip) [55]

observaciones para las que la neurona indicadora de transferencias fraudulentas toma valor 0.5 o superior se considerarán transferencias fraudulentas. Es normal que en este problema en particular deseemos tener un número bajo de falsos negativos (transferencias que no hemos podido detectar como fraudulentas), incluso a cambio de tener un alto número de falsos positivos, por lo que si disminuimos este *threshold* exigiremos a esta neurona de salida una menor seguridad (esto es, un valor menor a 0.5) para predecir transferencias de tipo fraudulento, provocando que sea más díficil que exista un alto número de falsos negativos. Esta variación de *thresholds* es lo que se realiza en el cálculo del AUC y lo que garantiza que nuestro modelo realice predicciones con gran acierto para transferencias de ambos tipos. Hemos decidido no incluir en la memoria capturas de curvas ROC dado que los valores AUC son cercanos a 100 y por lo tanto estas curvas son casi imposibles de visualizar.

Tabla 3: Resumen de modelos entrenados por el Playground

Fuente: Logs generados en benchmark/resultados [55]

Modelo	**Tiempo entrenamiento**	**AUC test**	**FN (%)**
modelo 6	3 min 59s	99.1	22.85
modelo regularizado	2 min 09s	99.81	7.14
modelo sin regularizar	2 min 07s	52.09	100
regresión logística	2 min 42s	98.94	32.55

Además de realizar este análisis más exhaustivo sobre el modelo 6, he probado otros modelos sobre el dataset CCF, tal y como podemos ver en la tabla 3. Por un lado he probado dos modelos tomando el modelo 6 pero sin aplicar batch normalization. En el *modelo regularizado* he aplicado una regularización de tipo L1 con un parámetro de regularización (beta) igual a 0.0001. Interesante destacar como aplicando esta regularización se obtienen unos resultados aún mejores al modelo 6, con un porcentaje de falsos negativos realmente pequeño (con un threshold de 0.5). Además, el tiempo de entrenamiento se reduce a la mitad, pues recordemos que al incluir batch normalization se realizan más cómputos y existen más parámetros que entrenar. En el caso de redes neuronales con pocas capas ocultas, como aquí ocurre, el hecho de incluir batch normalization puede traducirse en un aumento del tiempo de entrenamiento, aunque aún así las cifras son bastante razonables para un dataset de cerca de 285.000 observaciones y 30 variables.

Sin embargo, si entrenamos un modelo idéntico al modelo 6 pero sin aplicar batch normalization ni ningún tipo de regularización (lo he denominado *modelo sin regularizar*), los resultados son realmente decepcionantes. El valor del AUC final es cercano a 50, lo que quiere indicar que nuestro modelo ha sufrido un sobreentrenamiento y está realizando predicciones de forma aleatoria, dado que no es capaz de predecir las transferencias de tipo fraudulento. Con este modelo queremos destacar como batch normalization y la regularización de tipo L1 pueden ser de gran ayuda con datasets con clases desproporcionadas, hasta el punto de incluso lograr (como en este caso) resultados realmente buenos.

Por último también he probado a entrenar un modelo por medio de una regresión logística en nuestro benchmark. Los resultados en este caso también han sido bastante buenos, aunque el tiempo ha sido superior al del modelo regularizado. Si accedemos al archivo *log.txt*, el cual recoge un resumen del modelo entrenado y la evaluación de las métricas por cada iteración del algoritmo, se puede observar como en una sola iteración se llega a alcanzar un 92 % de AUC sobre los datos de validación, lo cual nos indica que es posible que modelos de regresión lineal (por ejemplo regresiones logísticas, que son menos complejas matemáticamente que las redes neuronales que estamos considerando) puedan devolver resultados satisfactorios en unas pocas iteraciones.

5. Análisis de entrenamientos y ajuste de hiperparámetros con ML Engine

Con el objetivo de mostrar los conceptos desarrollados en el capítulo 5, he realizado un entrenamiento con ajuste de hiperparámetros y otros dos entrenamientos posteriores, todos ellos utilizando modelos *Wide and Deep* y el dataset CCF. En cada modelo *Wide and Deep* se ha discretizado la variable *amount* del dataset por medio de intervalos (*boundaries*) para la parte *Wide* (regresión lineal). Por otro lado las otras 29 variables continuas del dataset CCF se han utilizado en la parte *Deep* (red neuronal profunda). Con el objetivo de comparar tiempos con los entrenamientos del benchmark, se ha elegido siempre en ML Engine una máquina con GPU (*BASIC_GPU*). Todos los resultados mostrados en esta sección se pueden encontrar en el repositorio Github del trabajo [55].

Para poder realizar el ajuste de hiperparámetros se ha utilizado el archivo *hptuning_config.yaml* y el script Bash *scriptml.sh*. En total se han entrenado 15 modelos con un dataset CCF balanceado por medio del script en R *split.R*. Consultando el archivo YAML, vemos que se ha intentado maximizar el AUC en estas pruebas, corriendo 3 modelos como máximo en paralelo. Comentar que el tiempo total para realizar el ajuste ha sido de unos 50 minutos, tal y como se puede observar con TensorBoard.

Cada uno de los 15 modelos se ha entrenado a lo largo de 700 epochs, variando cada uno de ellos el número de capas de ocultas (entre 1 y 15), así como el número de neuronas en cada una de ellas (empezando con 25 como máximo en la primera capa oculta y decreciendo a partir de ahí). En la figura 7 tenemos un resumen del ajuste de hiperparámetros sobre el conjunto de datos de validación, utilizando AUC como métrica.

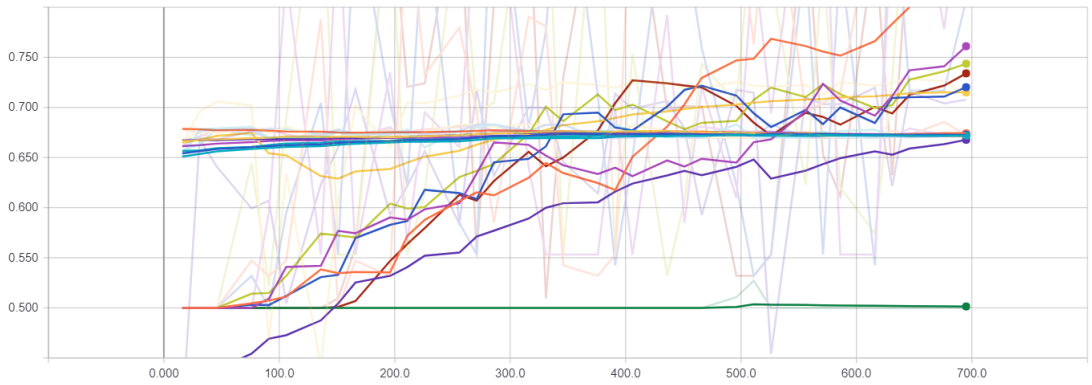


Figura 7: Valores AUC sobre datos de validación para un ajuste de hiperparámetros en ML Engine

Fuente: TensorBoard (cloud/ml_engine/resultados/ml_engine_15_pruebas.zip) [55]

Al igual que ocurría con el ajuste de hiperparámetros realizado en 4, se puede comprobar en la gráfica de TensorBoard como los modelos con valores AUC casi constantes son aquellos bastante complejos (con un gran número de capas ocultas) mientras que los modelos sencillos son los que logran mejores resultados. De hecho, el mejor modelo es el número 14, que posee una sola capa oculta de 2 neuronas. Este modelo es realmente similar al escogido como óptimo en 4, el cual tenía una capa oculta con 3 neuronas, por lo que esto refuerza la hipótesis de que el problema de clasificación asociado al dataset CCF se puede resolver con redes neuronales con una sola capa oculta o con incluso una simple regresión logística. Uno de los principales problemas de ML Engine es que no permite hacer evaluaciones sobre un conjunto de test, tal y como realizamos en nuestro benchmark, por lo que si queremos realizar predicciones con nuestro modelo entrenado necesitamos volver a instanciar un objeto de la clase *tf.contrib.learn.DNNLinearCombinedClassifier* indicando la carpeta del modelo a restaurar.

Además de este ajuste de hiperparámetros, he realizado un entrenamiento con el dataset CCF desbalanceado. Con *split.R* se ha dividido el dataset en un conjunto de entrenamiento (80 %) y otro de validación (20 %) manteniendo en ambos el desbalanceo original de las dos clases. Como podemos ver en la figura 8, solamente se han tardado unos 5 minutos en realizar 500 iteraciones de una red neuronal con dos capas ocultas, de 20 y 15 neuronas respectivamente. Este tiempo es claramente menor al obtenido por nuestro benchmark para el mismo modelo: 40 minutos, por lo que el entrenamiento en local es unas 8 veces más lento. No obstante, en ML Engine el modelo sufre un sobreentrenamiento y devuelve un valor AUC de 50 para el conjunto de validación, a diferencia del modelo del benchmark, que obtiene un AUC de 99 sobre el conjunto de validación.

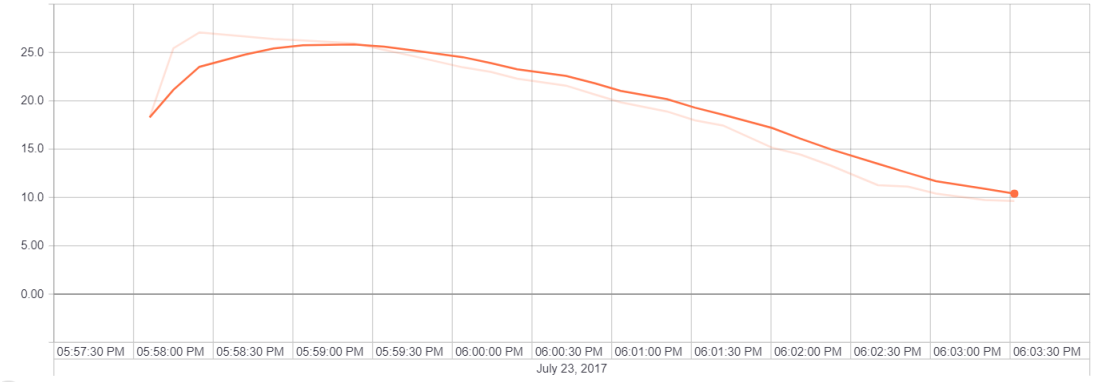


Figura 8: Función de coste a lo largo del tiempo en ML Engine

Fuente: ML Engine (cloud/ml_engine/resultados/ml_engine_no_sampled.zip) [55]

Con el objetivo de facilitar la comparación, se han incluido también los resultados de esta ejecución en el benchmark en el repositorio Github (*playgroundcomp.zip*).

Por último, y con el objetivo de evitar el sobreentrenamiento que hemos sufrido con el dataset CCF al no realizar undersampling, he entrenado el mismo dataset balanceado que en el ajuste de hiperparámetros (unas 1.000 observaciones en total). En este modelo también he usado una estructura de red neuronal con dos capas ocultas de 20 y 15 neuronas respectivamente, la cual sorprendentemente tarda 5 minutos en realizar 500 iteraciones. Es decir, los modelos *Wide and Deep* en TensorFlow tardan el mismo tiempo en realizar un entrenamiento para 1.000 que para 285.000 observaciones[3].

Sin embargo los resultados son realmente decepcionantes, si atendemos a los gráficos de la figura 9. La métrica *accuracy*, que en este caso es equivalente a AUC debido a la proporcionalidad de clases en el dataset, es siempre inferior al 90 %, logrando un porcentaje máximo de aciertos (88 %) en la iteración 391. Si comparamos estos resultados con una ejecución del benchmark con los mismos hiperparámetros y el mismo dataset balanceado (*playgroundsampled.zip*), vemos que el benchmark es capaz de obtener un AUC de 97 sobre datos de test en solo 100 epochs.

En general, hay que admitir que los resultados en ML Engine no son buenos. Aunque hemos visto como los entrenamientos con GPU son bastante más rápidos que en el benchmark local, la poca cantidad de datos utilizados en estas pruebas nos han impedido comprobar el verdadero poder de computación de ML Engine. Los resultados de los modelos *Wide and Deep* con el dataset desbalanceado han mostrado que estos modelos necesitan hiperparámetros adicionales de regularización para prevenir el sobreentrenamiento y que necesitamos conocer más en detalle la API de TensorFlow que hemos utilizado, pues para el dataset CCF balanceado los resultados siguen sido peores a aquellos entrenamientos del benchmark con el mismo dataset pero desbalanceado.

[3] Es posible que esto se deba al hecho de que estos modelos están preparados para realizar ingestas de datos realmente grandes con mecanismos algo complejos, por lo que la mayoría de tiempo se perderá en este proceso.

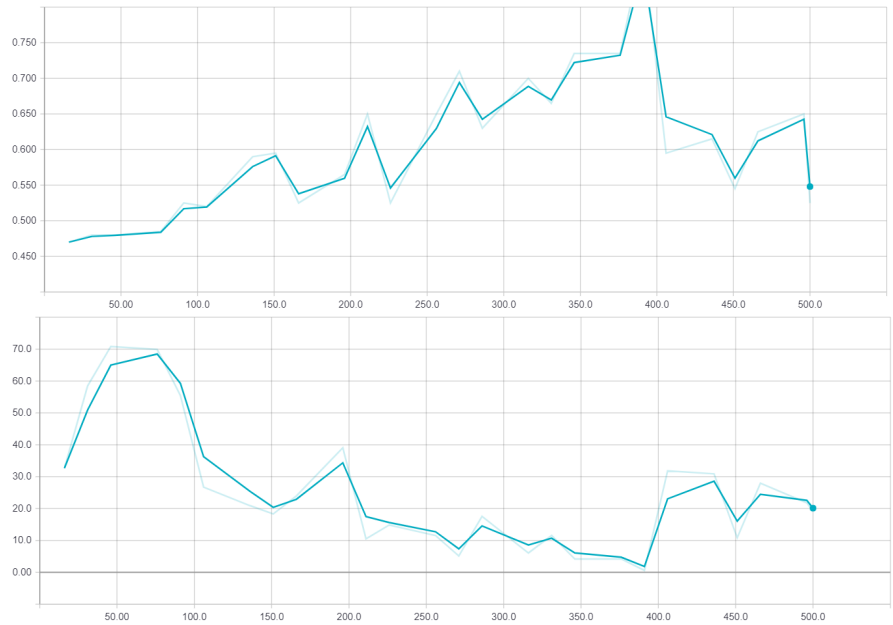


Figura 9: Accuracy y función de coste para un dataset balanceado en ML Engine

Fuente: Tensorboard (cloud/ml_engine/resultados/ml_engine_sampled.zip) [55]

6. Ejemplo de una red neuronal de Keras en formato JSON


```
{
    "class_name":"Sequential",
    "keras_version":"2.0.0-tf",
    "config":[
        {
            "class_name":"Dense",
            "config":{
                "kernel_initializer":{
                    "class_name":"VarianceScaling",
                    "config":{
                        "distribution":"normal",
                        "scale":2.0,
                        "seed":null,
                        "mode":"fan_in"
                    }
                },
                "name":"dense_1",
                "kernel_constraint":null,
                "bias_regularizer":null,
                "bias_constraint":null,
                "dtype":"float32",
                "activation":"linear",
                "trainable":true,
                "kernel_regularizer":null,
                "bias_initializer":{
                    "class_name":"VarianceScaling",
                    "config":{
                        "distribution":"normal",
                        "scale":2.0,
                        "seed":null,
                        "mode":"fan_in"
                    }
                },
                "units":20,
                "batch_input_shape":[
                    null,
                    30
                ],
                "use_bias":true,
                "activity_regularizer":null
            }
```

```
        },
        {
            "class_name":"BatchNormalization",
            "config":{
                "beta_constraint":null,
                "gamma_initializer":{
                    "class_name":"Ones",
                    "config":{

                    }
                },
                "moving_mean_initializer":{
                    "class_name":"Zeros",
                    "config":{

                    }
                },
                "name":"batch_normalization_1",
                "epsilon":0.001,
                "trainable":true,
                "moving_variance_initializer":{
                    "class_name":"Ones",
                    "config":{

                    }
                },
                "beta_initializer":{
                    "class_name":"Zeros",
                    "config":{

                    }
                },
                "scale":true,
                "axis":-1,
                "gamma_constraint":null,
                "gamma_regularizer":null,
                "beta_regularizer":null,
                "momentum":0.99,
                "center":true
            }
        },
        {
            "class_name":"Activation",
            "config":{
```

```
            "activation":"elu",
            "trainable":true,
            "name":"activation_1"
         }
      },
      {
         "class_name":"Dropout",
         "config":{
            "rate":0.5,
            "trainable":true,
            "name":"dropout_1"
         }
      },
      {
         "class_name":"Dense",
         "config":{
            "kernel_initializer":{
               "class_name":"VarianceScaling",
               "config":{
                  "distribution":"normal",
                  "scale":2.0,
                  "seed":null,
                  "mode":"fan_in"
               }
            },
            "name":"dense_2",
            "kernel_constraint":null,
            "bias_regularizer":null,
            "bias_constraint":null,
            "activation":"linear",
            "trainable":true,
            "kernel_regularizer":null,
            "bias_initializer":{
               "class_name":"VarianceScaling",
               "config":{
                  "distribution":"normal",
                  "scale":2.0,
                  "seed":null,
                  "mode":"fan_in"
               }
            },
            "units":15,
            "use_bias":true,
            "activity_regularizer":null
```

```
            }
        },
        {
            "class_name":"BatchNormalization",
            "config":{
                "beta_constraint":null,
                "gamma_initializer":{
                    "class_name":"Ones",
                    "config":{

                    }
                },
                "moving_mean_initializer":{
                    "class_name":"Zeros",
                    "config":{

                    }
                },
                "name":"batch_normalization_2",
                "epsilon":0.001,
                "trainable":true,
                "moving_variance_initializer":{
                    "class_name":"Ones",
                    "config":{

                    }
                },
                "beta_initializer":{
                    "class_name":"Zeros",
                    "config":{

                    }
                },
                "scale":true,
                "axis":-1,
                "gamma_constraint":null,
                "gamma_regularizer":null,
                "beta_regularizer":null,
                "momentum":0.99,
                "center":true
            }
        },
        {
            "class_name":"Activation",
```

```
            "config":{
               "activation":"elu",
               "trainable":true,
               "name":"activation_2"
            }
         },
         {
            "class_name":"Dropout",
            "config":{
               "rate":0.5,
               "trainable":true,
               "name":"dropout_2"
            }
         },
         {
            "class_name":"Dense",
            "config":{
               "kernel_initializer":{
                  "class_name":"VarianceScaling",
                  "config":{
                     "distribution":"uniform",
                     "scale":1.0,
                     "seed":null,
                     "mode":"fan_avg"
                  }
               },
               "name":"dense_3",
               "kernel_constraint":null,
               "bias_regularizer":null,
               "bias_constraint":null,
               "activation":"softmax",
               "trainable":true,
               "kernel_regularizer":null,
               "bias_initializer":{
                  "class_name":"VarianceScaling",
                  "config":{
                     "distribution":"uniform",
                     "scale":1.0,
                     "seed":null,
                     "mode":"fan_avg"
                  }
               },
               "units":2,
               "use_bias":true,
```

```
218                 "activity_regularizer":null
219             }
220         }
221     ],
222     "backend":"tensorflow"
223 }
```