Elaboración de un algoritmo predictivo sobre la evolución del precio de las criptomonedas

TRABAJO DE FIN DE GRADO

Por
Manuel Pastor Cobo y Pablo de Torre Barrio

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Ingeniería del Software
Facultad de Informática

Directores: María Isabel Pita Andreu y José Alberto Verdejo López

Elaboración de un algoritmo predictivo sobre la evolución del precio de las criptomonedas

Madrid, 2018–2019

Agradecimientos

Queríamos agradecer a Alberto e Isabel el apoyo que nos han dado durante todo el desarrollo de nuestro trabajo y la atención e interés que han depositado en nosotros.

También queremos agradecer a nuestras familias, que nos han permitido estudiar esta carrera y nos la han financiado.

Por último, queremos agradecer a Andrew Ng. de la Universidad de Stanford y a Microsoft por sus excelentes cursos sobre redes neuronales que nos han ayudado muchísimo a la hora de introducirnos en este mundo.

Sobre TEFLON

Teflon(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado o Trabajos de Fin de Máster. La versión usada es la 1.3.

V:1.3 Overleaf V2 with pdfLaTeX, margin 1in, NO-bib

Contacto
Autor: David Pacios Izquiero
Correo: dpacios@ucm.es
ASCII: asciifdi@gmail.com
Despacho 110 - Facultad de Informática

Índice general

Índice de figuras

Listings

Resumen

En este proyecto trataremos de comprobar si una red neuronal es capaz de predecir el precio de una criptomoneda. Esta tarea es complicada porque los precios de las criptomonedas son muy volátiles, lo que supone un reto a la hora de desarrollar la red neuronal.

Comenzaremos por una fase larga de aprendizaje en la que realizaremos cursos de Inteligencia Artificial, tanto básicos como más avanzados. También nos ayudaremos del trabajo previo de otros investigadores y estudiantes.

Ya introducidos en el mundo de la inteligencia artificial, pasaremos a recopilar los datos sobre criptomonedas necesarios para entrenar a nuestra red. Les aplicaremos un preprocesamiento para facilitar el aprendizaje y, una vez estén listos los datos, procederemos al desarrollo de toda la infraestructura necesaria para nuestra red.

Durante este desarrollo nos encontraremos con múltiples complicaciones, como pueden ser la falta de datos o los problemas de formato. Para solucionarlos recopilaremos estos datos de diferentes plataformas y probaremos bibliotecas específicas para el formateo de los mismos.

Con toda la estructura de la red definida pasaremos a una fase de prueba y error en la que comprobaremos cuáles son los factores que más influyen a la hora de realizar una predicción en este ámbito.

Finalmente, expondremos posibles mejoras que podremos llevar a cabo en un futuro si conseguimos los medios necesarios.

El software generado a lo largo del trabajo está disponible en un repositorio público de GitHub. La dirección de este repositorio es la siguiente:
https://github.com/PabloDeTorre/NeuralBTCPredictor

Palabras clave

- Inteligencia artificial
- Redes neuronales
- Criptomonedas
- Predicciones

Abstract

In this project we will try to verify if a neuronal network is able to predict the evolution of a cryptocurrency price. This is a complex task because of the volatility of the cryptocurrencies prices, which is a challenge when developing a neuronal network.

We will start with a long learning phase in which we will take basic Artificial Intelligence courses and, later on, more advanced ones. We will also help ourselves with the previous work of other researchers and students.

Once introduced in the world of Artificial Intelligence, we will collect the data about cryptocurrencies needed to train our network. We will apply a preprocessing to make learning easier. Once the data is ready, we will proceed to develop the entire necessary infrastructure for our network.

During this development we will encounter multiple complications, such as be the lack of data or format problems. To solve them we will collect these data from different platforms and we will test libraries specialized in data formatting.

With the entire structure of our neuronal network defined, we will go through a trials phase. In this stage, we will check which are the factors that influence the most when making a prediction in this area.

Finally, we will discuss possible improvements that we can carry out in the future if we get the necessary resources.

All the software created for this project is available in the following GitHub repository: https://github.com/PabloDeTorre/NeuralBTCPredictor

Keywords

- Artificial intelligence
- Neuronal networks
- Cryptocurrencies
- Predictions

Capítulo 1

Introducción

1.1. Contexto

1.1.1. Criptomonedas

Nacimiento

El concepto de "moneda criptográfica" o "criptomoneda" lo publicó Wei Dan por primera vez en 1998 en la lista de correo "cypherpunks" en un artículo llamado "b-money, an anonymous, distributed electronic cash system". La idea se fundamenta en la creación de un nuevo tipo de medio de cambio basado en la criptografía, la cual se utiliza para garantizar la seguridad en las transacciones y evitar monedas duplicadas o estafas.

El objetivo que se busca es crear transacciones seguras, descentralizadas y anónimas.

La criptografía es la encargada de asegurar que las monedas que recibes son auténticas, aunque no conozcas al emisor.

Qué son

Actualmente las transacciones de criptomonedas se realizan mediante conexiones "peer-to-peer" o p2p, muy parecidas a las utilizadas en la red de Torrent. Es decir, estas transacciones no están respaldadas por ningún tipo de entidad u objeto físico que garantice que esas monedas valen lo que dicen, sino que el usuario emisor y el receptor consideran que otras personas van a estar dispuestas a pagar esa cantidad por esas monedas.

Debido a esto, el valor de las criptomonedas es extremadamente volátil y muy dependiente de la confianza que depositen las personas en ellas, por lo que elementos como las redes sociales afectan directamente al precio de estas.

Las monedas se almacenan en *carteras*. Una cartera es una dirección única por usuario y tipo de moneda. Las casas de intercambio de criptomonedas crean una cartera para cada usuario y criptomoneda con la que se pueda comerciar en su página web. Este tipo de carteras se llaman carteras en caliente, porque suelen tener bastante movimiento de entrada y salida. A las carteras que se utilizan para acumular monedas durante grandes

periodos de tiempo se las denomina carteras en frío y lo más común es que sean independientes de una casa de intercambio.

La gran mayoría de estas monedas, para funcionar de una forma descentralizada, utilizan para la transmisión y almacenamiento de transacciones una red "blockchain" o una *cadena de bloques*. Esta es una estructura de datos formada por bloques, unos conjuntos que contienen la información que se quiere almacenar y metainformación de otro bloque anterior a él codificada. Gracias a esto, si se quiere editar la información de un bloque, hay que modificarla en todos los bloques siguientes a este, lo que hace la transacción mucho más segura. La tecnología de cadena de bloques aplicada a las criptomonedas permite crear bases de datos públicas que contengan los datos históricos de todos los movimientos que se han realizado de un tipo de moneda. Por lo que, en todo momento, cualquier usuario puede consultar si una cartera es sospechosa o no. Además, si se llega a un consenso dentro de la red de usuarios de una criptomoneda, es posible asegurar que los datos de la red son fiables, por lo que no es necesario depender de una entidad externa que aporte esa confianza, como es el caso del dinero fiduciario actual y los bancos.

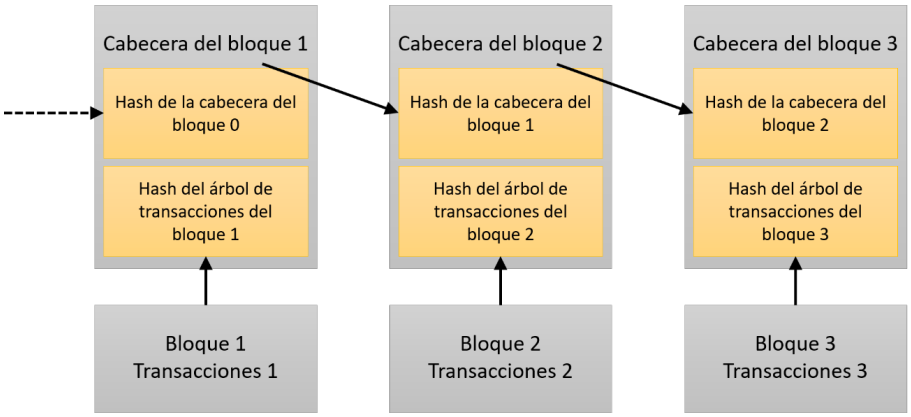

Figura 1.1: Funcionamiento de la cadena de bloques.

El anonimato sería posible a todos los niveles de no ser porque las casas de intercambio obligan, por motivos legales, a dar los datos personales en su plataforma en cuanto un usuario sobrepasa un cierto límite de dinero o de transacciones al día.
Este anonimato se adquiere gracias a que las carteras de monedas no contienen ningún tipo de información sobre la persona propietaria de esa cartera y las transacciones se realizan entre direcciones de carteras de una misma moneda y no entre usuarios.

Bitcoin

La primera implementación del concepto de criptomoneda fue el Bitcoin, actualmente la criptomoneda más importante del mercado y poseedora de más del 57 % del capital invertido en este tipo de monedas. También fue la primera moneda en aplicar la cadena de bloques y en registrar todos los movimientos que se hacen entre carteras.

También, Bitcoin, es la criptomoneda que se puede utilizar en más casas de intercambio y que, a corto plazo, es la más asentada en la economía actual. Es por esto que hemos

decidido desarrollar nuestro trabajo de fin de grado con datos relativos a esta moneda.

1.1.2. Redes neuronales

Qué son

Las redes neuronales tienen como objetivo replicar el funcionamiento de una red neuronal real mediante software.

Cada uno de los nodos de una red neuronal se llama neurona artificial y es la encargada de ejecutar las operaciones que sean necesarias. Las conexiones entre neuronas son las que contienen el resultado de esas operaciones y el peso, o la importancia, que hay que darle a ese resultado sobre el resultado global de la red.

Las redes se estructuran en *capas de neuronas*, existiendo siempre un mínimo de dos capas: una capa de entrada y una capa de salida. En la capa de entrada se almacenan todos los datos que van a ser procesados. Y en la capa de salida se muestran los resultados de las operaciones. Muy habitualmente también tienen, como mínimo, otra capa llamada capa oculta, en la que se aplican las operaciones necesarias para dar la salida que se haya solicitado.

Además, se pueden aplicar funciones entre capas que modifiquen el resultado de una capa o para limitar los valores que se permiten pasar a la siguiente capa. Estas funciones se conocen como *funciones de activación*.

Una de las características más importantes de las redes neuronales es la capacidad de aprender de sí mismas, es decir, no hay que programar una serie de reglas que deben cumplir, sino que es la propia red la que decide qué necesita para ajustarse al contexto en el que se encuentra.

Para poder utilizar la red primero es necesario entrenarla. Con los resultados de este entrenamiento se crea un modelo. Para llevar a cabo este entrenamiento se utilizan conjuntos de datos de los que ya conoces la respuesta esperada. Esos datos se separan en dos grupos, el de entrenamiento y el de validación. Los datos de entrenamiento son los que utiliza la red para aprender y los de validación para comprobar el buen funcionamiento de la red.

Para comprobar cómo de bien o de mal está funcionando la red debemos mirar la función de pérdida, que comprueba cuánto se asemejan los resultados de la red durante la fase de aprendizaje a los resultados reales. Nuestro objetivo será minimizar el resultado de esta función.

Una vez que podemos considerar que la red ya se ha ajustado lo suficiente a los datos, de forma que devuelve resultados suficientemente fiables como para utilizarlos, se exporta un modelo que contiene los pesos que ha asignado la red a cada una de las variables de los datos de entrada. Este modelo es el que se utiliza con datos reales para calcular los resultados deseados.

Tipos y aplicaciones

Las redes neuronales se utilizan para resolver una gran cantidad de tareas, como pueden ser la realización de predicciones, el reconocimiento de caras o de voz, la detección de anomalías o la creación de bots automáticos para juegos como el ajedrez.

Se han desarrollado diferentes tipos de redes neuronales para resolver cada uno de estos problemas.

La **red neuronal monocapa** o **perceptrón simple** (SLP por sus siglas en inglés) es la red neuronal más simple que podemos encontrar. Está compuesta tan solo por una capa de entrada que envía los datos directamente a la capa de salida, donde se realizan algunas operaciones simples.

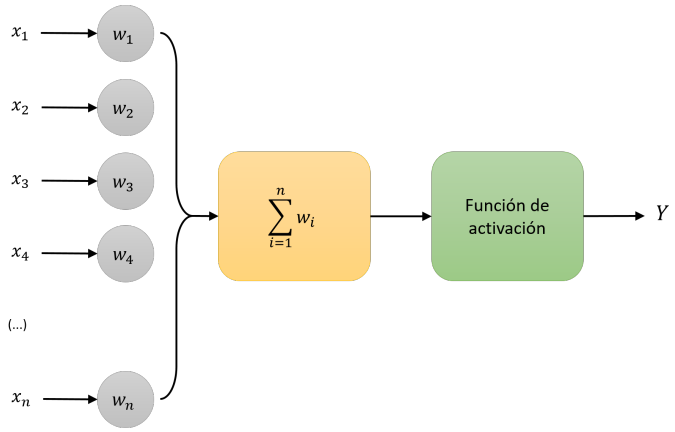

Figura 1.2: Perceptrón simple.

La **red neuronal multicapa** o **perceptrón multicapa** (MLP por sus siglas en inglés) es una extensión del perceptrón simple. Su principal diferencia es la existencia de capas ocultas entre la capa de salida y la de entrada.

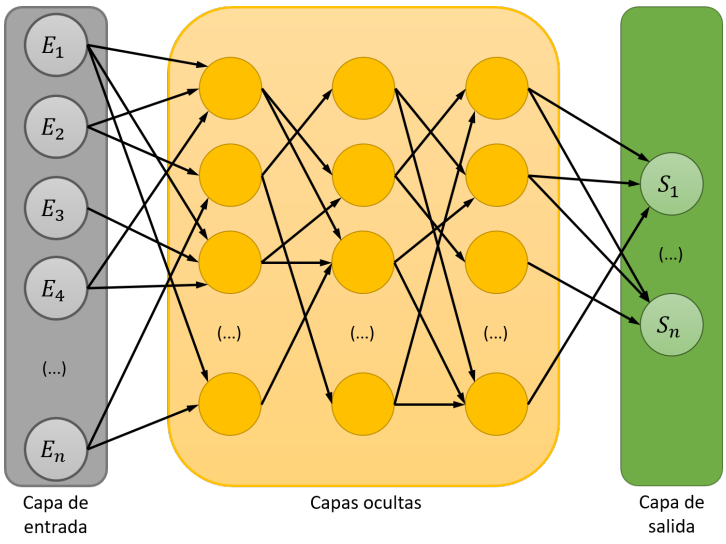

Figura 1.3: Perceptrón multicapa.

La **red neuronal convolucional** (CNN por sus siglas en inglés) es una red de neuronas especializadas. Esto es así porque no todas las neuronas están conectadas entre sí, sino que se especializan en un pequeño grupo. De esta forma se consigue disminuir la complejidad computacional y la cantidad de neuronas que se necesitan para ejecutar la red.

Las redes convolucionales son muy buenas para el análisis de imágenes. Por lo tanto, algunos de los usos de las redes convolucionales es el reconocimiento de rostros o la conducción automática de automóviles.

La estructura de estas redes constan de una serie de capas **convolucionales** y **reductoras** intercaladas y una **capa clasificadora** al final.

Las capas convolucionales están formadas por neuronas que se encargan de crear un mapa de características de una zona de los datos de entrada, es decir, convierten una zona de la matriz de entrada en el resultado de aplicar una serie de operaciones de suma y producto definidas por un elemento denominado **filtro**. Las capas reductoras son las encargadas de extraer las características más habituales. De esta forma se reducen el número de parámetros, simplificando los cálculos, aunque supone una pérdida de precisión. Finalmente, la capa de clasificación, que es la que realiza la separación de los datos en clases. Está formada por neuronas conectadas todas entre sí, existiendo una neurona por clase en la que se quieren clasificar los datos.

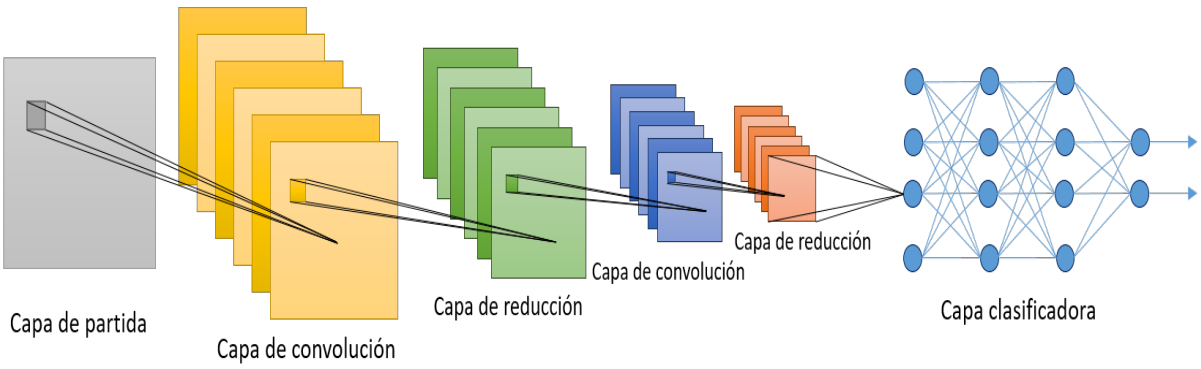

Figura 1.4: Red neuronal convolucional.

La **red neuronal recurrente** (RNN por sus siglas en inglés) es una red neuronal muy parecida al perceptrón multicapa. La principal diferencia se encuentra en que las neuronas se interconectan, además de con las neuronas de la siguiente capa, consigo mismas. De esta forma crean ciclos en los que su salida sirve de entrada adicional, de forma que cuando los datos atraviesan definitivamente esa capa, la red se ha ajustado mejor a ellos.

Este tipo de red neuronal es excelente para realizar predicciones de series temporales. Algunos ejemplos son la predicción de precios o la escritura predictiva.

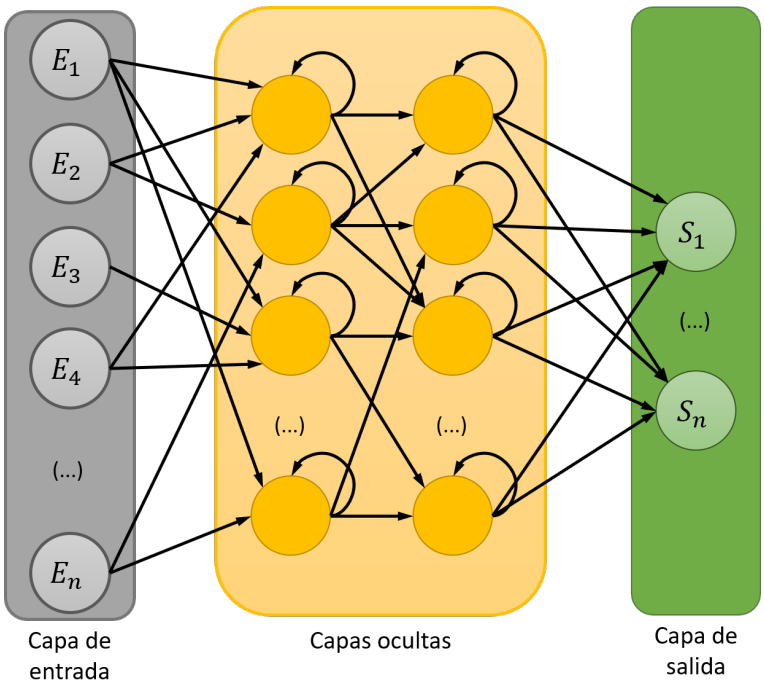

Figura 1.5: Red neuronal recurrente.

Finalmente, las **redes de base radial** (RBF por sus siglas en inglés) son redes que calculan los resultados basados en la distancia de la salida de una función a un centro. El resultado de la red es una combinación de todas las funciones de activación que han utilizado las neuronas.

Se utilizan para resolver problemas de clasificación al igual que las convolucionales.

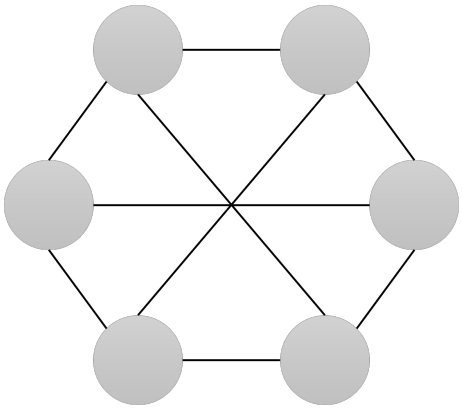

Figura 1.6: Red de base radial.

1.2. Motivación

Hemos decidido realizar este trabajo porque nos permite combinar dos campos que nos despiertan un gran interés, como son las redes neuronales y el mundo de las criptomonedas. Buscamos aumentar nuestro conocimiento en redes neuronales, ya que en nuestros

estudios tan sólo disponemos de una asignatura optativa y una pequeña introducción en una asignatura obligatoria.

Pensamos que las criptomonedas van a suponer un cambio en la economía en el futuro y que es un buen momento para invertir, ya que es un mercado que actualmente se encuentra en fase de recuperación. Las monedas ahora mismo tienen precios bajos y están entrando grandes inversores tradicionales en el mercado, como es el caso de la familia Rockefeller, que se asoció con Coinfund a través de Venrock, una empresa suya de capital riesgo[4]. Gracias a estos inversores se está asentando una base de confianza para el resto de población.

Finalmente, queremos estudiar la viabilidad del desarrollo de las redes neuronales de predicción de precios en un mercado tan volátil como el de las criptomonedas.

1.3. Objetivos

Nuestros objetivos son los siguientes:

- Desarrollar una red neuronal capaz de predecir si el precio de una criptomoneda va a subir o bajar utilizando los datos históricos de esa criptomoneda.
- Comprobar cuál es la mejor configuración de la red neuronal en este contexto y comprobar la viabilidad de la misma.
- Comprobar cuáles son las variables que mayor impacto tienen en el precio de una criptomoneda.

Capítulo 2

Introduction

2.1. Context

2.1.1. Cryptocurrency

Criptocurrencies birth

The concept of "cryptographic currency" or "cryptocurrency" was published by Wei Dan for the first time in 1998 on the mailing list "cypherpunks". It was mentioned in an article called "b-money, an anonymous, distributed electronic cash system". The idea is based on the creation of new types of medium of exchange based on cryptography. Cryptography will guarantee security in transactions and will avoid duplicate currencies or scams.

The goal is to create secure, decentralized and anonymous transactions.

Cryptography is used to ensure the coins you receive are authentic, even when the sender is unknown.

What are cryptocurrencies?

Currently, cryptocurrency transactions are made through "peer-peer" or p2p connections, similar to those used in the Torrent network. This implies that no entity or physical object guarantees the currencies value. The value is based on that the issuing user and the receiver consider that other people will be willing to pay that amount for those currencies.

The cryptocurrencies value is extremely volatile because it depends on the trust people place in them. This is the reason that elements such as social networks directly affect their price.

Cryptocurrencies are stored in wallets. A wallet is a unique address per user and type of currency. The cryptocurrency exchange houses create a portfolio for each user and cryptocurrency with which it is possible to trade on their website. This type of wallets are called hot wallets because they tend to have a lot of movement in and out. The wallets used to accumulate currencies for long periods of time are called cold storage wallets and they are, usually, independent of an exchange house.

The vast majority of these currencies, to operate in a decentralized manner, use a "block-

chain" network or chain of blocks for the transmission and storage of transactions. This is a data structure formed by blocks. A block contains the data that you want to store and encoded meta information from the previous block. To edit the information of a certain block, it is necessary to modify it in every subsequent block. This makes the transaction safer. The blockchain technology applied to cryptocurrencies allows the creation of public databases that contain the historical data of all movements that have been made of a currency type. At any time, any user can check if a portfolio is suspicious or not. In addition to this, if a consensus is reached within the cryptocurrency user network, it is possible to ensure that the network data is reliable, so it is not necessary to depend on an external entity that provides that trust, as is the case of current fiduciary money and banks.

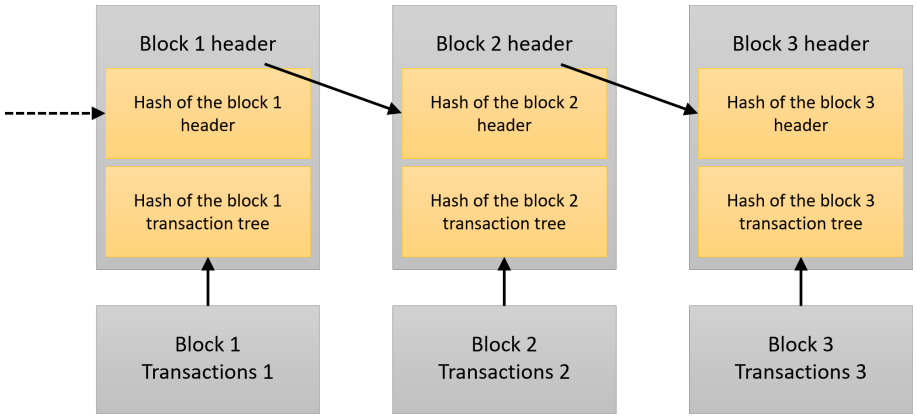

Figura 2.1: Blockchain's working process.

Theoretically, anonymity would be possible at all levels. But, due to legal reasons, exchange houses oblige to provide personal information on their platform as soon as a user exceeds a certain limit of money or transactions per day. Anonymity is acquired because currency wallets do not contain any information about the wallet owner. Transactions are made between wallets addresses of the same currency and not between users.

Bitcoin

Bitcoin was the first implementation of the concept of cryptocurrency. Currently, it is the most important cryptocurrency market. 57 % of the capital invested on cryptocurrencies is invested on Bitcoin. It was also the first currency to apply the blockchain and to record all the movements made between wallets.

Bitcoin is the cryptocurrency that can be used in more exchange houses and it is, short term, the most established cryptocurrency in the current economy. This is the reason why we will be using Bitcoin data for our final degree project.

2.1.2. Neuronal networks

Neuronal networks

Neuronal networks aim to replicate the functioning of a real neuronal network through software.

Each node of a neuronal network is called artificial neuron and it is the responsible for executing the necessary operations. The connections between neurons contain the result of those operations and the weight, or importance, that must be given to that result on the overall result of the network.

The networks are structured in *neurons layers*. They have a minimum of two layers: an input layer and an output layer. The data to be processed is stored in the input layer. The output layer shows the operations results. Neuronal networks, usually have, at least, another layer named hidden layer. The hidden layer performs the operations needed to get the requested output.

Functions can be applied between layers to modify the result of a layer or to limit the values that are allowed to pass to the next layer. These functions are the *activation functions*.

One of the most important characteristics of neuronal networks is the ability to learn from themselves. It is not necessary for the user to program a series of rules to be followed, but the network decides what is necessary to adjust to the context.

The first step to using the network is to train it. With the results of the training, a model is created. To carry out this training, datasets are used, of which you already know the expected response. Data is divided in two groups, the training data and the validation data. Training data is those used by the network to learn. Validation data is used to check the proper functioning of the network.

To check the network performance, we check the loss function. The loss function verifies, during the learning phase, how close are the results of our network to the actual results. Our goal will be to minimize this function output.

Once the network returns reliable results, we consider the network sufficiently adjusted to the data. Then, we export a model containing the weights assigned by the network for each of the input data variables. This model is used with real data to calculate the desired outputs.

Neuronal networks types and use

Neuronal networks are used for a large number of tasks: predictions making, faces or voice recognition, anomalies detection or automatic bots creation for games such as chess.

Different types of neuronal networks have been developed to solve each of these problems.

The **single layer** or **single layer perceptron** (SLP) neuronal network is the simplest neuronal network. It is composed of an input layer that sends data directly to the output layer, where some simple operations are performed.

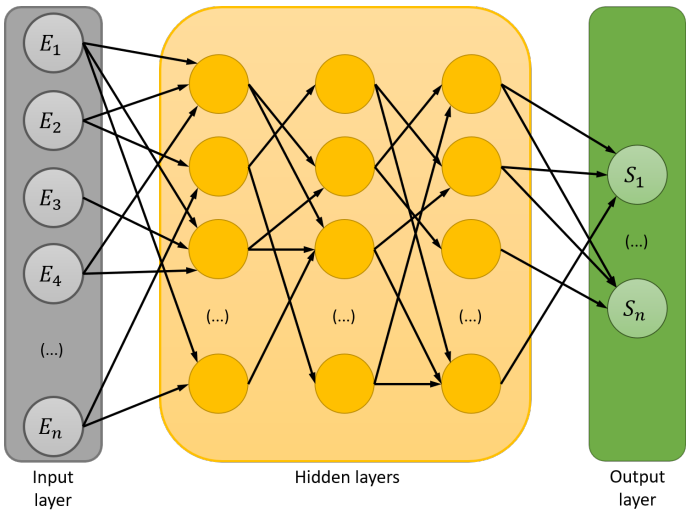

Figura 2.2: Simple perceptron.

The **multilayer neuronal network** or **multilayer perceptron** (MLP) is an extension of the simple layer perceptron. It has hidden layers between the output layer and the input layer.

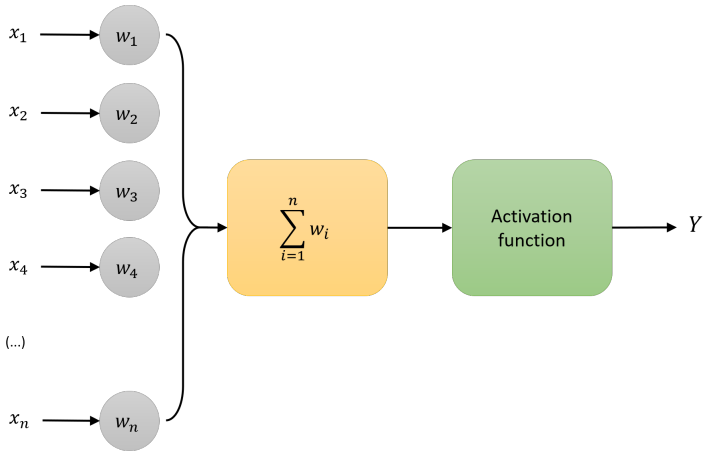

Figura 2.3: Multilayer perceptron.

The **convolutional neuronal network** (CNN) consist of specialized neurons. Not every neuron connects to each other, but they specialize in small groups instead. This reduces the computational complexity and the number of neurons needed for the network to work.

Convolutional networks are very good for image analysis. Therefore, some of the uses of convolutional networks is face recognition, or self-driving cars.

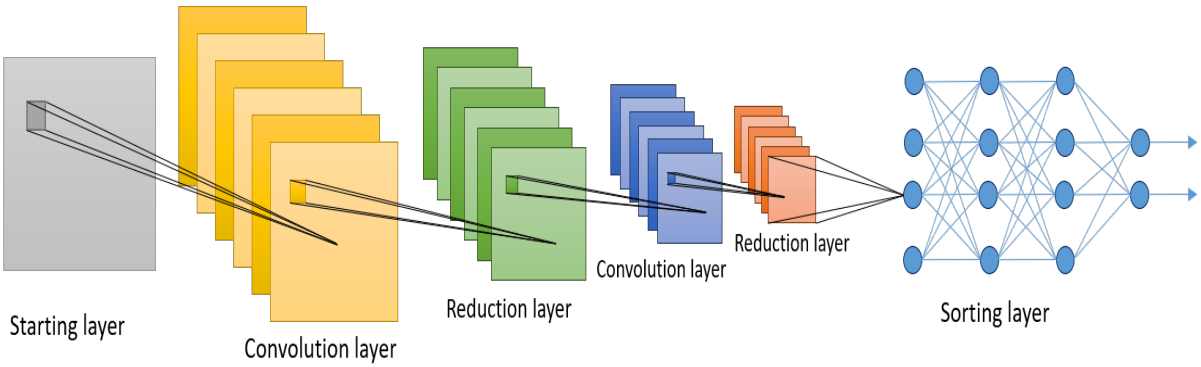

Figura 2.4: Convolutional neuronal network.

The **recurrent neuronal network** (RNN) is a neuronal network very similar to the multilayer perceptron. In a recurrent neuronal network, neurons interconnect with each other, as well as with the neurons in the next layer.

In this way they create cycles in which their output serves as an additional input. When data goes through that layer, the network has adjusted better.

This kind of neuronal network is excellent for making predictions of time series.

Some examples are price prediction or predictive writing.

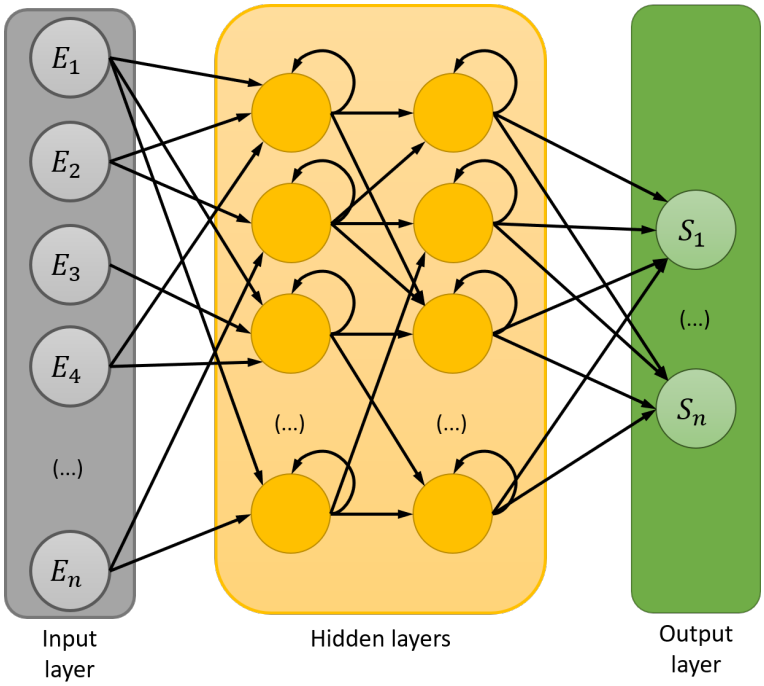

Figura 2.5: Recurrent neuronal network.

Finally, **radial base networks** (RBFs) are networks that calculate the results based on the distance from the function output to a center. The result of the network is a combination of every activation functions used by neurons.

They are used to solve classification problems.

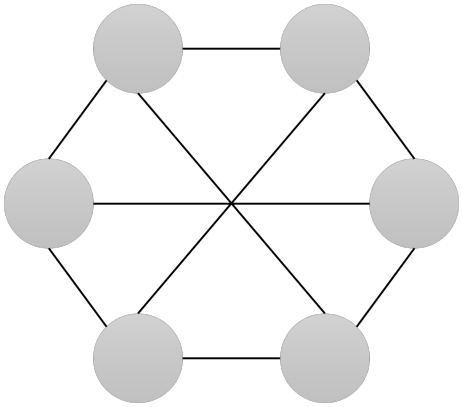

Figura 2.6: Radial base network.

2.2. Motivation

We have decided to carry out this work because it allows us to combine two fields that arouse great interest, such as neuronal networks and cryptocurrencies.

We seek to increase our knowledge in neuronal networks. In our current degree studies we just have an optional subject and a short neuronal networks introduction in a compulsory subject.

Our opinion is that cryptocurrencies are going to change economy in the future and, now, it is a good time to invest, since this market is currently in recovery phase. Cryptocurrencies have low prices at this moment and large traditional investors - like the Rockefeller family, which has partnered with Coinfund through Venrock, a venture capital company[4] - are entering the market. Thanks to these investors, a trust base for the rest of the population is establishing.

Finally, we want to study the feasibility of developing neuronal networks to predict prices in a market as volatile as cryptocurrencies.

2.3. Goals

These are our goals:

- To develop a neuronal network capable of predicting if the price of a certain cryptocurrency will increase or decrease using cryptocurrency historical data.
- Research the best neuronal network configuration for this context and check if its performance is good enough to make worthwhile to use it in a real context.
- Check which are the variables with the greatest impact on a cryptocurrency price.

Capítulo 3

Aprendizaje

Antes de meternos de lleno en la realización de nuestro trabajo, necesitamos hacer una investigación sobre los contenidos que vamos a tratar relacionados con las redes neuronales. Esto se debe a que, en el momento de comenzar nuestro proyecto, ninguno de los participantes habíamos cursado ninguna asignatura relativa a este temario.

A lo largo del segundo cuatrimestre del presente curso 2018/2019, en la asignatura de Ingeniería del conocimiento, hemos podido ver algunos conceptos y representaciones de redes neuronales. Por otro lado, ambos hemos formado parte del programa Alumno 4.0, realizado en una colaboración entre la Facultad de Informática y Microsoft, junto con varias empresas asociadas. En este programa hemos visto distintas formas de trabajar con inteligencia artificial, entre las que se incluyen algunas de las necesarias para nuestro trabajo.

Para la preparación del trabajo hemos decidido realizar el curso "Neuronal Networks and Deep Learning"[5], impartido por la universidad de Standford en la plataforma online de educación Coursera. Este curso forma parte de un catálogo bastante variado, siendo este el más básico sobre redes neuronales. Una vez terminado este curso decidimos completarlo con parte del temario de otro de los cursos de esta misma plataforma, "Machine Learning"[6].

Estos cursos se dividen por semanas. En la primera semana enseñan a representar una red neuronal, como la de la figura 1.5, e introducen al auge del "Deep Learning" y a la explicación de los diferentes tipos de redes neuronales que existen. Entre estos tipos pudimos ver las redes neuronales recurrentes, que son las que utilizamos en nuestro proyecto final puesto que, como comentábamos en la sección 1.1, sirven para realizar predicciones de precios.
En el siguiente apartado de la primera semana se empieza a tratar el concepto de *regresión logística*, utilizada para los experimentos en los que la salida es binaria, es decir, la salida será "1" si se cumple lo que estamos buscando y "0" si no se cumple. Para ello utilizamos el vector X, donde guardamos los datos de la entrada. En nuestro caso serán todos los datos relacionados con el precio del Bitcoin. También necesitamos un vector Y, que representa la salida. Nosotros consideramos que la salida será "1" si el valor de la criptomoneda ha aumentado o se ha mantenido de un instante de tiempo al siguiente, y "0" si no.

En este curso también nos dan a conocer los componentes de una red neuronal. Una red neuronal puede tener distintas entradas en la capa inicial, o capa de entrada. Estas

entradas pasan por entre una y n capas ocultas. Una capa oculta contiene un vector de pesos que irá variando para dar mayor importancia a los datos que aumenten el ajuste de la red creada al conjunto de datos. El flujo de la imagen anterior se realiza más de una vez, lo que permite que la matriz de pesos se vaya actualizando y vaya mejorando los resultados a cada vuelta que da. Los resultados del proceso realizado por todas las capas ocultas se juntarán en la capa de salida. En ella obtenemos la probabilidad de que la salida se cumpla, es decir, que devuelva un "1". A cada ejecución que se hace sobre la red se le llama "época".

A la hora de inicializar la matriz de pesos, en el curso recomiendan que se haga de forma aleatoria, así evitamos que en dos nodos de una capa se obtenga un mismo resultado, lo que haría que los pesos de esos nodos evolucionaran de la misma forma e impidieran mejorar los resultados.

Otro de los conceptos vistos es el de la función de activación, que se dedica a devolver una salida en función de la entrada que le ha llegado a esa capa. Por ejemplo, la función tangente hiperbólica (tanh) devuelve valores entre -1 y 1, donde los valores que se aproximan a los extremos lo hacen en forma de asíntota.

En la última semana del curso enseñan cómo podemos asegurarnos de que las matrices que utilizamos tienen las dimensiones correctas. A lo largo de la descripción del desarrollo del trabajo comentaremos cómo llevamos a cabo las trasformaciones necesarias de los datos para que la red los aceptara. Por otro lado, explican también la importancia de los hiperparámetros, que son aquellos parámetros que no afectan de forma directa a los datos, pero sí a la capacidad de aprendizaje de la red neuronal. Hay varios hiperparámetros configurables, de los que hablaremos en profundidad a lo largo de esta memoria para comentar los resultados que hemos obtenido sobre ellos y las conclusiones que hemos sacado. Algunos de los hiperparámetros que manejamos son: el número de épocas de la ejecución, la función de activación y el número de unidades de una capa oculta.

A lo largo del curso se explica el funcionamiento interno de una red neuronal de respuesta binaria, que se basa en el algoritmo de regresión logística que hemos comentado al principio de este capítulo. Para minimizar la pérdida de la red neuronal, combinamos el uso de la regresión logística con el modelo del gradiente descendente, que busca encontrar el valor mínimo de una función (figura 3.1). Para ello tomamos un valor y le restamos la derivada de la función en ese mismo valor, multiplicándolo después por un número α, que representa cuánto nos desplazamos en la búsqueda del mínimo[7].

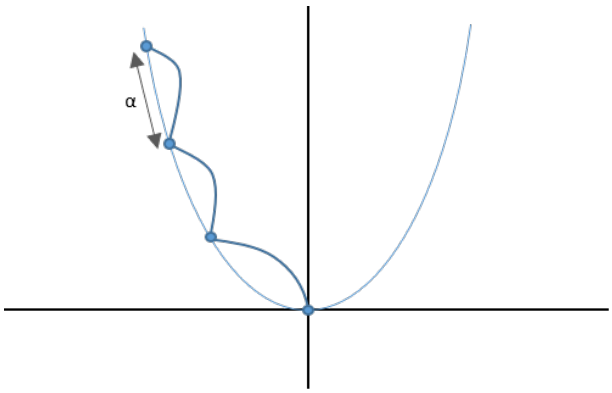

Figura 3.1: Representación del gradiente descendente.

Aunque toda la explicación del funcionamiento interno es de mucha utilidad para la comprensión del trabajo a realizar, los contenidos y demostraciones impartidas en el curso no tienen una representación explícita dentro del código que hemos generado. En cambio, mucha de la información vista en este curso nos ha sido de gran utilidad, en especial el aprendizaje de la terminología utilizada, lo que nos ha permitido comprender la información que hemos buscado a posteriori por internet, ya sea tanto en la documentación oficial, como en artículos especializados. Para poder configurar la red de forma más sencilla, ha sido muy relevante el conocimiento adquirido sobre los hiperparámetros, así como de otros conceptos como la función de pérdida de la red, para comprender mejor los resultados que devuelve, o de los módulos **NumPy** y **Pandas**. Utilizaremos Pandas para tratar los datos y analizarlos, y NumPy para trabajar con matrices, lo que nos servirá para dar el formato necesario a los datos que utilicemos. Más adelante haremos una descripción detallada de estos módulos.

Una vez finalizado este curso, dimos comienzo a la implementación del código del proyecto, y a lo largo del desarrollo empezamos el programa Alumno 4.0 de Microsoft, en el que nos enseñaron a usar su herramienta de Machine Learning. Esta herramienta se llama Machine Learning Studio (o ML Studio)[8], que nos permite crear soluciones de Machine Learning sin necesidad de programar, solamente uniendo y colocando elementos visuales.

Hemos hecho pruebas con esta aplicación para la recopilación de datos, cargando varias tablas de la página oficial de Bitcoin[9], previamente descargadas en formato CSV, y uniéndolas, ya que cada una de las tablas que se pueden extraer solo tienen dos columnas: la fecha y la información relativa al indicador que corresponda. A continuación, adjuntamos una figura reducida, ya que, al haber utilizado 37 tablas distintas sería complicado ver los módulos utilizados y el trabajo realizado.

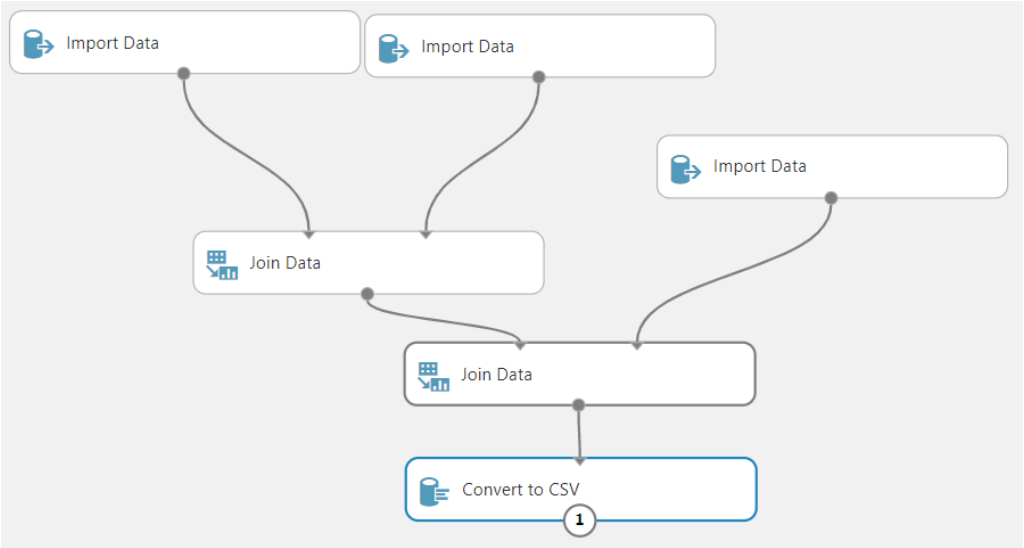

Figura 3.2: Unión de 3 tablas en una sola, que exportamos a formato CSV. Captura realizada sobre Machine Learning Studio.

Por otro lado, a lo largo del programa también hemos hecho uso de ML Studio para la aplicación de algoritmos de Machine Learning. Por ejemplo, realizamos pruebas sobre un experimento proporcionado por la propia aplicación, "Quantile Regression: Car price prediction". En este experimento se extraen datos sobre varias características de una serie de vehículos y se pretende realizar un modelo predictivo. Para ello se hace uso del módulo Fast Forest Quantile Regression, que realiza predicciones en función de la distribución de los valores del conjunto de datos. Este modelo de regresión se aplica después al módulo Tune Model Hyperparameters, que se encarga del proceso de prueba y error para el modelo seleccionado.

El resultado del proceso anterior da lugar al modelo predictivo ya entrenado, el cual se probará con una serie de datos nuevos de los que ya conocemos el resultado. Esta prueba se hace en el módulo Score Model, que nos devolverá los resultados de la ejecución del modelo predictivo.

De las charlas recibidas a lo largo del programa sacamos una idea que explicaremos en la sección de respuesta no binaria del desarrollo, la creación de un script que pruebe distintas combinaciones de los hiperparámetros de la red. Una vez terminada la ejecución de estas pruebas podríamos revisar de forma manual cuáles han dado mejores resultados.

Aquí termina nuestra fase de aprendizaje, donde los cursos de la plataforma Coursera han sido de gran utilidad para nosotros. El siguiente paso ha sido recopilar una serie de documentos de los que también hemos hecho uso a lo largo del desarrollo del proyecto.

3.1. Recopilación de documentos

Destinamos la semana del 11 de febrero a recopilar toda la documentación que consideramos relevante, a leerla y a apuntar aquellos datos que nos pudiesen ayudar a desarrollar este trabajo.

Del proyecto "Automated Bitcoin Trading via Machine Learning Algorithms"[10] obtuvimos dos APIs con las que empezar a recopilar datos sobre el precio del Bitcoin: la API de Coinbase[11] y la de OKCoin[12]. Por desgracia, no pudimos recopilar todos los parámetros que se mencionan en el artículo, ya que solo los proporcionan las APIs si tienes una licencia de pago.

En el artículo "Applying Deep Learning to Better Predict Cryptocurrency Trends"[13] fue donde vimos que el optimizador adam era el que mejor funcionaba con los datos de criptomonedas y que la función de activación ReLU daba resultados mejores que otras.

El documento que más nos ha ayudado ha sido el trabajo de fin de carrera de Sean McNally, del National College of Ireland, "Predicting the Price of Bitcoin using Machine Learning"[14]. Al igual que nosotros, Sean McNally trata de desarrollar una red neuronal capaz de predecir la evolución del precio del Bitcoin. Gracias a este documento y al curso que realizamos de la universidad de Stanford en Coursera, supimos que necesitábamos una red neuronal recurrente y que la ejecución de una red en GPU era mucho más eficiente que en una CPU. También pudimos obtener una tercera API, la de Poloniex[15], una plataforma de intercambio de criptomonedas.

Capítulo 4

Preparación previa al desarrollo

4.1. Entorno

A la hora de elegir el entorno de ejecución hemos tenido en cuenta varias cosas. Por un lado, tiene que darnos la posibilidad de utilizar una GPU para la paralelización del proceso. Por otro lado, debe permitirnos la instalación de los paquetes necesarios para llevar a cabo nuestro proyecto.

A nivel de sistema operativo, nos hemos decidido por una distribución de Linux, ya que nos da mucho más control sobre los paquetes y casa muy bien con el lenguaje de programación utilizado, **Python**. Sobre el lenguaje de programación hablaremos en la sección siguiente.

Antes de elegir exactamente la distribución que vamos a utilizar, tenemos que decidir el equipo. Rápidamente descartamos la posibilidad de utilizar los ordenadores de los laboratorios de la facultad, ya que no nos permiten instalar paquetes y nos generaría mucha dependencia de horarios. Tampoco podíamos utilizar una máquina virtual ya que dificultaba la paralelización mediante GPU.

Otra de las posibilidades que se nos presentó fue la de hacer uso de la nube, en un entorno preconfigurado que nos facilitase esa etapa del desarrollo, pero fue también descartada porque no podíamos asumir el gasto que supondría su uso.

Finalmente hemos decidido utilizar uno de nuestros equipos personales, puesto que nos permite hacer todo lo que necesitábamos y nos da plena libertad sobre su uso.

Las especificaciones del equipo utilizado son las siguientes:

- **Procesador:** Intel(R) Core(TM) i7-7700K de 4.20GHz, 4200MHz, con 4 procesadores principales y 8 procesadores lógicos.
- **Placa base:** MSI Z270 GAMING M5
- **Memoria RAM:** 16GB DDR4 2666MHz
- **Tarjeta gráfica:** Nvidia GTX 1080 8GB GDDR5X

A este equipo le hemos añadido una partición con Ubuntu 16.04 de 40GB de almacenamiento, ya que esta cantidad nos parece más que suficiente para el trabajo que queremos

realizar.

Una vez elegido el equipo y el sistema operativo pasamos a la elección del IDE que vamos a utilizar para programar la red neuronal. En este apartado nos decidimos directamente por Visual Studio Code, entorno con el que estamos ambos familiarizados y que nos ofrece extensiones de gran utilidad como Live Share, que nos permite trabajar juntos sobre el mismo código e incluso ejecutar de forma remota la shell del miembro que está trabajando en local. Esta herramienta nos da la posibilidad de trabajar en paralelo sobre los mismos archivos sin preocuparnos de tener que hacer una mezcla de los contenidos después. Una extensión que también nos es de utilidad es Excel Viewer, para visualizar archivos con formato CSV directamente en Visual Studio Code, o Prettier, para poder tabular los archivos automáticamente y hacernos más sencilla la lectura de archivos Python y JSON.

Ahora que ya tenemos listo el entorno pasamos a configurarlo. Dentro de esta configuración tenemos que actualizar los drivers de Nvidia y añadir los paquetes necesarios para que **Tensorflow** se ejecute en su versión para GPU.

Tensorflow es una biblioteca de código abierto que se utiliza para desarrollar y entrenar modelos de Machine Learning[16].

Poner en funcionamiento el módulo para la GPU no fue una tarea sencilla ya que una vez terminado el tutorial de instalación de la página oficial de Tensorflow, aún nos daban problemas de incompatibilidad. Esto se debe a que no hay un instalador directo para este módulo y muchas de las versiones de las bibliotecas que se utilizan son incompatibles entre sí. Entre estas bibliotecas se encuentran **CUDA** y **CuDNN**.

Tanto CUDA como CuDNN son paquetes de Nvidia que nos permiten crear aplicaciones aceleradas por GPU de alto rendimiento. Con estos kits de herramientas podemos desarrollar, optimizar e implementar aplicaciones en sistemas acelerados por GPU[17][18].

Para hacer compatibles las versiones de cada una de las bibliotecas decidimos hacer uso de Anaconda, un módulo de Python que sirve para desplegar paquetes de ciencia de datos y aprendizaje automático en Python de una forma sencilla[19].

También nos fue de mucha utilidad el artículo de Medium “Install Tensorflow-GPU to use Nvidia GPU using anaconda on Ubuntu 18.04 / 19.04 do AI!”[20], en el cual encontramos una configuración que no nos da conflictos y nos permite ejecutar nuestro primer modelo de red neuronal. Medium es una plataforma de blogs de distintos ámbitos, siendo la tecnología uno de ellos.

Ahora que hemos detallado todos los aspectos necesarios del entorno que hemos utilizado, procedemos a hablar del lenguaje de programación que hemos utilizado, Python, y los motivos que nos han llevado a utilizarlo.

4.2. Lenguaje

Python es un lenguaje de programación que permite trabajar rápidamente e integrar sistemas de manera efectiva. En particular, es muy útil para nosotros ya que tiene funcionalidades especializadas de análisis numérico y aprendizaje automático[21].

Antes del comienzo del presente curso (2018/2019) ninguno de los miembros del grupo había utilizado Python a excepción de la asignatura optativa "Cloud y Big Data", donde se hace uso de la biblioteca spark para el tratamiento de conjuntos de datos de gran tamaño[22]. Igualmente, no profundizamos en este lenguaje hasta este curso, en el que lo hemos podido utilizar, tanto en la asignatura de Ingeniería del Conocimiento, como en el programa Alumno 4.0 de Microsoft, mencionado anteriormente.

En la asignatura de Ingeniería del conocimiento no se nos obligaba a utilizar este lenguaje, pero decidimos hacer uso de él para las prácticas en las que teníamos que implementar algoritmos de Inteligencia Artificial, y para practicar de cara a este TFG. Por otro lado, el temario del programa de Microsoft contenía una introducción y explicación del lenguaje y nos permitía crear nuestros propios módulos en Python dentro de la herramienta ML Studio.

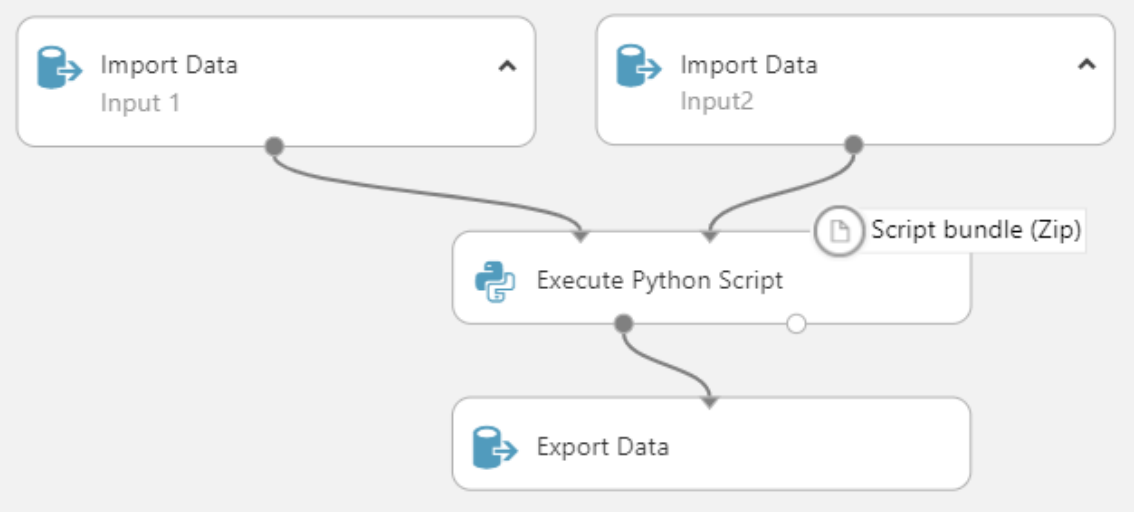

Figura 4.1: Representación del módulo de Python para ML Studio, junto con su código interno. Captura realizada sobre Machine Learning Studio.

Otro de los motivos principales por los que hemos decidido utilizar este lenguaje ha sido por la gran cantidad de información que hay en internet, desde la página oficial de Tensorflow hasta artículos de Medium en los que se explican diferentes formas de realizar una red neuronal con Tensorflow en Python[20][23].

Python también nos ofrece algunas bibliotecas de gestión de datos que nos han resultado de gran utilidad a lo largo del proyecto. Hemos utilizado en gran medida tanto Pandas como NumPy.

Pandas es una biblioteca de código abierto de Python que proporciona estructuras de datos de alto rendimiento y fáciles de usar, además de herramientas de análisis de datos[24]. Por otro lado, NumPy está orientado a la computación científica sobre Python[25], pero nosotros lo utilizaremos básicamente por ser un contenedor eficiente de datos genéricos.

4.3. Elección de la biblioteca de Deep Learning

Necesitamos una red neuronal capaz de persistir los datos de ciclos anteriores. La red va a analizar secuencias de datos muy dependientes entre sí, por lo que si no recordara qué ha sucedido en los instantes anteriores sería imposible realizar predicciones fiables.

Las redes neuronales que cumplen con estas características son las redes neuronales recurrentes, ya que pasan al ciclo siguiente datos de los ciclos anteriores. Por lo tanto, vamos a utilizar una red neuronal recurrente para nuestro trabajo.

Para el desarrollo de nuestra red neuronal vamos a hacer uso de **Keras**, una API de redes neuronales de alto nivel[26].

Nos hemos decantado por Keras, por encima de otras bibliotecas más potentes como Tensorflow[16] o Theano[27], porque permite la creación fácil y rápida de prototipos gracias a su modularidad y extensibilidad y tiene implementaciones específicas para CPU y GPU. Además, las redes neuronales desarrolladas sobre Keras se pueden extender añadiendo muy pocas líneas de código y pueden ejecutarse sobre Theano y Tensorflow.

Esta biblioteca está pensada para aprender a desarrollar redes neuronales antes de pasar a plataformas más complejas, así que es perfecta para nosotros, ya que nuestros conocimientos de Machine Learning previos a desarrollar este trabajo eran muy bajos.

Queríamos una biblioteca que nos permitiera plasmar lo que íbamos aprendiendo de una forma sencilla y Keras lo cumple a la perfección.

4.4. Funcionamiento de los módulos

Los módulos de Keras que utilizamos a lo largo del proceso son Model, de la biblioteca models, para la creación del modelo, y varios pertenecientes a la biblioteca layers, entre los que se encuentran algunos como Activation, Input y Dense, para la configuración de las capas del modelo.

La explicación de estos módulos la veremos a medida que vayan saliendo durante el capítulo del desarrollo del proyecto.

4.4.1. Tipos de arquitecturas recurrentes

Como hemos mencionado en el capítulo anterior, Keras dispone de implementaciones optimizadas tanto para CPU, como para GPU, de las arquitecturas de redes neuronales más comunes.

Nosotros, al ir a desarrollar una red neuronal recurrente, nos hemos centrado en las dos arquitecturas recurrentes que tiene implementadas: la LSTM (del inglés Long Short-Term Memory) y la GRU (del inglés Gated Recurrent Unit). Hemos obviado otra de las arquitecturas de estas redes porque el mecanismo de aprendizaje de la LSTM y de la GRU manejan mejor los datos históricos, por lo que es más difícil que el aprendizaje de la red se estanque.

LSTM

La arquitectura LSTM, al ser recurrente, dispone de conexiones que retroalimentan las capas de la red, permitiendo computar los mismos problemas que una máquina de Turing.

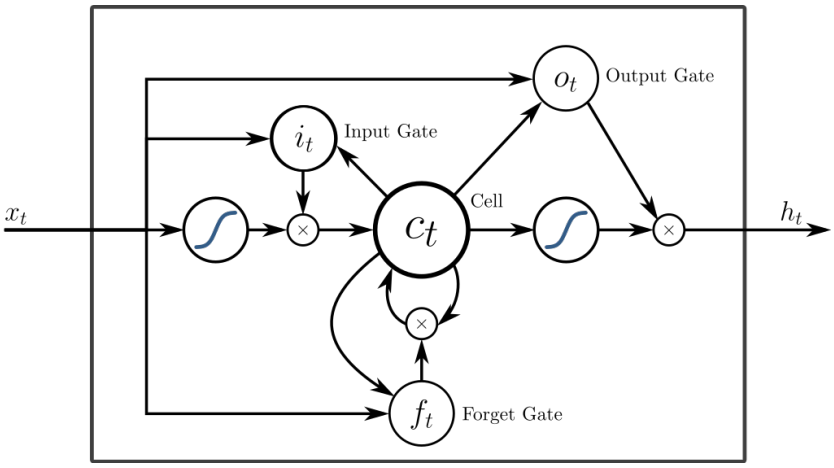

Figura 4.2: Unidad LSTM[1].

En una red neuronal recurrente tradicional, los datos no persisten mucho tiempo porque el sistema de persistencia de una RNN está compuesto por una sola capa, que calcula el peso que van a tener esos datos históricos. El problema surge cuando se necesitan datos de muchos ciclos atrás para realizar la predicción del ciclo siguiente. En las redes LSTM del sistema de persistencia se encargan cuatro capas que interactúan entre sí. Por tanto, las redes LSTM solucionan el problema del largo plazo con el que se encuentran las redes neuronales recurrentes tradicionales.

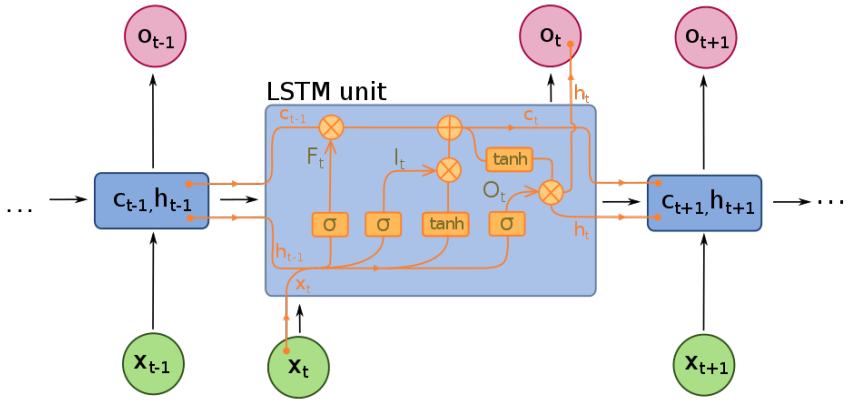

Figura 4.3: Estructura interna de una unidad LSTM[2].

La parte más importante de una red LSTM es la línea de estado, en el diagrama la línea C_{t-1}-C_t. Esta línea es como una cinta transportadora que está a lo largo de toda la unidad con cambios mínimos o nulos.

Para decidir qué información hay que añadir o eliminar de la línea de estado, las LSTM utilizan unas estructuras denominadas puertas. Las puertas son una forma de dejar pasar

la información solo en determinadas ocasiones. Están formadas por operaciones sigmoideas y una operación de multiplicación. El resultado de las operaciones son números entre cero y uno, siendo cero no dejar pasar información y uno dejar pasar toda la información.

Las LSTM tienen tres de estas puertas para controlar la información que fluye a través de la línea de estado.

La primera puerta que interactúa con la información es la llamada "puerta del olvido" (forget gate en inglés). En la imagen anterior la línea F_t. Es la encargada de decidir qué información histórica hay que desechar.

El siguiente paso es decidir qué nueva información se va a añadir a la línea de estado y aquí es cuando interviene la "puerta de entrada" (input gate en inglés). La línea I_t en la imagen.

Una vez la red sabe qué datos va a quitar y cuáles va a añadir hay que actualizar la línea de estado. Para ello multiplica la línea de estado anterior por la salida de la puerta del olvido, que son valores entre cero y uno de todos los datos, indicando qué porcentaje o qué peso tiene en la matriz de datos. Una vez olvidados los datos se suman los nuevos y pasa a la última fase: decidir cuál va a ser la salida.

La salida se calcula en la puerta de salida (output gate en inglés). En la imagen es la línea O_t. Es una versión cribada de la nueva línea de estado actualizada. Se aplica una operación sigmoidea que decide qué partes de la línea de estado van a formar parte de la salida y, después, se aplica a la línea de estado una tangente hiperbólica para asegurar que los valores se encuentran entre -1 y 1. Por último se multiplica la salida de la tangente hiperbólica por la salida de la operación sigmoidea, de forma que solo vamos a sacar los datos que se han elegido.

GRU

Las redes GRU son una simplificación de una red LSTM común. Si las LSTM estaban compuestas por tres puertas, las GRU tienen dos. Una puerta llamada "puerta de actualización", R_t en la imagen inferior, que es el resultado de combinar la puerta del olvido y la puerta de entrada de la LSTM tradicional, y la puerta de salida, Z_t en la imagen inferior.

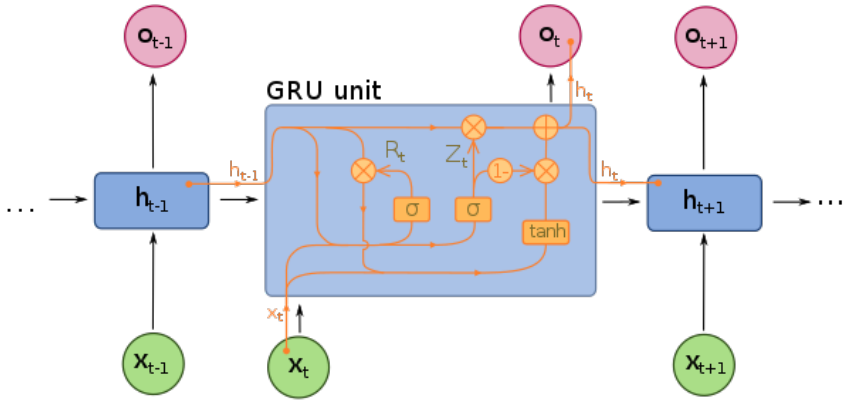

Figura 4.4: Estructura interna de una unidad GRU[3].

En general el rendimiento de las GRU es equivalente al de una LSTM. Suelen tener una menor complejidad, ya que los datos atraviesan menos puertas. Por tanto, entrenan más rápido y necesitan menos datos para aprender, pero recuerdan peor las secuencias largas de datos, al realizar una criba de datos menos exhaustiva.

Como nosotros vamos a trabajar con secuencias temporales muy largas, que contienen datos en intervalos muy cortos de tiempo relativos a varios años, nos resulta una arquitectura menos atractiva que la LSTM porque necesitamos que recuerde la mayor cantidad de entradas posibles. Sin embargo, es la mejora en eficiencia lo que nos hace tenerla en cuenta a la hora de desarrollar nuestra red neuronal.

4.4.2. CPU vs GPU

Keras tiene implementaciones tanto para CPU como para GPU de todos sus tipos de redes neuronales.

En general, la ejecución de una red neuronal en una GPU va a ser siempre mucho más rápida, ya que una CPU dispone de muchos menos núcleos que una GPU. Cuanto mayor sea el conjunto de datos a procesar, mayor será la diferencia en rendimiento entre uno y otro.

Las implementaciones específicas de Keras dependen de CuDNN, la biblioteca de Nvidia para desarrollo de redes neuronales. CuDNN depende de la plataforma de computación en paralelo CUDA, la cual sólo funciona en tarjetas Nvidia. Existen implementaciones menores para tarjetas gráficas de otros fabricantes como AMD, pero son aún muy jóvenes y están muy poco pulidas. Es por esto que, para realizar nuestro trabajo, hemos necesitado una tarjeta Nvidia.

Capítulo 5

Desarrollo

5.1. Creación de la API de recopilación de datos

Lo primero que debemos hacer para empezar el desarrollo es recoger los datos históricos necesarios y de utilidad sobre el Bitcoin, que es la criptomoneda en particular que vamos a tratar. Después de mirar varias APIs nos decidimos por Poloniex, para nuestro primer modelo de red neuronal. Esto se debe a que, de las APIs gratuitas que encontramos en un principio, era la que mejores datos nos proporcionaba y nos dejaba hacer más peticiones.

Decidimos hacer una petición que nos devolviera una fila por hora desde el 1 de enero de 2016 hasta el día actual. Las columnas que nos devolvían eran: valor de cierre, fecha, valor máximo alcanzado en ese periodo, valor mínimo, valor de apertura, volumen de Bitcoins transferidos en ese periodo y volumen medio por transacción.

Fue bastante complicado encontrar datos útiles, ya que la gran mayoría de las APIs que hay en internet son de pago y su uso gratuito te limita en gran medida las funcionalidades y peticiones que puedes llevar a cabo.

Para extraer los datos de la API hicimos uso de la biblioteca **urllib**, que nos proporciona una interfaz de alto nivel para buscar datos en internet mediante URLs[28].

Para la extracción de los datos montamos una URL por cada año que queríamos recuperar y tratar, ya que la API no nos permitía realizar el número de peticiones necesarias para extraer todos los datos de una sola vez. Para almacenar estos datos convertimos la respuesta de la API en un DataFrame de Pandas, para así poder cambiarle el nombre a las columnas, y después, convertirla en un archivo CSV. Este archivo es el que almacenamos y cargamos a la hora de ejecutar la red neuronal.

```
1 # connect to Poloniex's API
2 url2016 = 'https://poloniex.com/public?command=returnChartData
    &currencyPair=USDT_BTC&start=1451606400&end=1483227000&
    period=1800'
3 url2017 = 'https://poloniex.com/public?command=returnChartData
    &currencyPair=USDT_BTC&start=1483228800&end=1514763000&
    period=1800'
4 url2018toNow = 'https://poloniex.com/public?command=
```

```
    returnChartData&currencyPair=USDT_BTC&start=1514764800&end
    =999999999999&period=1800'
5
6 # parse json returned from Poloniex to Pandas DF
7 openUrl = urllib2.urlopen(url2018toNow)
8 r = openUrl.read()
9 openUrl.close()
10 d = json.loads(r.decode())
11 df = pd.DataFrame(d)
12
13 original_columns=[u'close', u'date', u'high', u'low', u'open',
    u'volume', u'weightedAverage']
14 new_columns = ['Close','Timestamp','High','Low','Open','Volume
    ','WeightedAverage']
15 df = df.loc[:,original_columns]
16 df.columns = new_columns
17 with open('data/bitcoin.csv', 'a') as f:
18     df.to_csv(f, index=False, header=False) #Delete headers=
       False when creating the file for first time
```

Listing 5.1: Llamada a la API de Poloniex y almacenamiento en un documento CSV.

5.2. Respuesta binaria

Ahora que tenemos los datos almacenados vamos a proceder a crear un nuevo conjunto de datos que contenga los resultados esperados. La respuesta que dará la red neuronal será una respuesta binaria, siendo el valor de esta nueva columna un 1 en caso de que aumente o se mantenga el valor de cierre entre un momento del tiempo y el inmediatamente anterior, siendo cada uno de estos "momentos" intervalos de una hora. En el caso de que el valor del momento anterior sea inferior al actual, el valor de la columna resultado será 0.

Para explicar toda la ejecución de la red, que va desde la lectura del documento CSV con todas las filas hasta el entrenamiento de la misma, vamos a detallar a continuación todo el proceso de forma secuencial.

Lo primero que vamos a hacer, como hemos comentado antes, es cargar el documento CSV en una variable. Para ello vamos a utilizar la función read_csv que nos ofrece Pandas. Una vez con los datos cargados, vamos a eliminar las columnas que no nos interesan como, por ejemplo, la fecha, puesto que al extraerse las filas ya ordenadas, no nos hace falta.

Nuestro conjunto de datos de entrada se llamará "inputs" y el conjunto de datos de salida se llamará "outputs". También editaremos cada fila para que, además de contener los datos que ya teníamos de antes, contengan varias de las filas anteriores. A esta medida la llamamos window_len, la cual podemos editar a través de código, y hace que cada fila contenga un array de dimensiones *nº de columnas*window_len*. Esto nos permite añadir más información a cada fila. La idea de añadir este parámetro la cogimos del artículo de Medium "How to predict Bitcoin and Ethereum price with RNN-LSTM in Keras"[23],

del cual, más adelante, también sacaremos alguna idea para mejorar nuestro modelo. En la misma función aplicaremos la transformación con window_len y añadiremos el valor de la columna de salida.

```
def createInputsAndOutputs(data, window_len):
    # -window_len first rows that we can't use
    # -1 last row also we can't use
    inputs = np.zeros((data.shape[0] - window_len, data.shape[
        1], window_len))
    outputs = np.zeros((data.shape[0] - window_len, 1))

    for i in range(data.shape[0] - window_len - 1):
        inputs[i] = data[i:i+window_len].T

        # Price decreased
        if(data[i+(window_len-1),0] > data[i+window_len,0]):
            outputs[i] = 0
        # Price did not change or increased
        else:
            outputs[i] = 1

    return inputs, outputs
```

Listing 5.2: Creación de los inputs y outputs.

Con los datos en el formato deseado para poder entrenar la red, llega el momento de construir el modelo que vamos a entrenar, para ello tendremos que crear el flujo de ejecución de la red. Este flujo se compone de todas las capas que queramos añadir, la primera capa que tenemos será la de los inputs, y la última capa, es una capa Dense, que es una capa que conecta entre sí todos los nodos de la capa anterior unificándolos.

Todas las capas que utilizamos pertenecen al módulo layers, de Keras, y todas estas capas deben ir acompañadas de una función de activación, que definen las posibles salidas de cada capa. Estas funciones de activación podremos editarlas a nuestra elección, al igual que las capas de la red, exceptuando la capa de entrada[29].

A la hora de compilar el modelo lo configuraremos para utilizar la función de pérdida mse (mean squared error), que calcula el promedio de errores al cuadrado. La función de pérdida debe minimizarse, y los pesos de cada neurona se irán actualizando para conseguirlo.

También debemos elegir un optimizador, que es el encargado de ajustar los pesos según los resultados de la función de pérdida. En un principio utilizaremos el optimizador adam, ya que ofrece ventajas como sus bajos requisitos de memoria, su sencillez a la hora de implementarlo y el poco ajuste que suelen necesitar los hiperparámetros cuando se utiliza, entre otras.[30].

Finalmente elegiremos las métricas que queremos que se muestren durante la ejecución, en este caso serán mae (mean absolute error), que mide el error medio del modelo, y acc (accuracy), que mide el porcentaje de ajuste del modelo a los datos. El modelo devuelto

por esta función es de tipo Model, y pertenece al módulo models de Keras.

```
def createModel(input_shape, output_size):
    inp = Input(shape=input_shape)

    lstm = CuDNNLSTM(
        units=32,
    )(inp)

    activationlstm = LeakyReLU(alpha=0.1)(lstm)
    dropout = Dropout(rate=0.25)(activationlstm)

    dense = Dense(units=output_size)(dropout)
    activation = Activation('sigmoid')(dense)

    model = Model(inputs=inp, outputs=activation)
    model.compile(loss='mse', optimizer='adam', metrics=['mae'
        ,'acc'])
    model.summary()
    return model
```

Listing 5.3: Construcción del modelo.

Con el modelo construido ya solo nos queda entrenarlo. Para hacerlo utilizaremos la función fit del propio modelo, la cual nos devolverá los resultados de la ejecución. Los resultados que devuelve son los siguientes:

- **loss**: Resultado de la función de pérdida en el conjunto de entrenamiento.
- **mae**: Error medio absoluto en el conjunto de entrenamiento.
- **acc**: Precisión. Porcentaje de aciertos en el conjunto de entrenamiento. El valor devuelto está entre 0 y 1 en vez de entre 0 y 100.
- **seg/ep**: Segundos por época.
- **v_loss**: Resultado de la función de pérdida en el conjunto de validación.
- **v_mae**: Error medio absoluto en el conjunto de validación.
- **v_acc**: Precisión en el conjunto de validación. El valor devuelto está entre 0 y 1 en vez de entre 0 y 100.

Para configurar el entrenamiento debemos especificar algunos parámetros más:

- Número de épocas (epochs): las veces que se ejecuta la red sobre el conjunto de datos. En cada época irá aprendiendo y ajustándose cada vez más al conjunto de entrenamiento.
- Tamaño del lote (batch_size): el número de datos que trata la red al mismo tiempo. Un número menor indica una ejecución más exhaustiva y, por lo tanto, más larga y precisa.

- Tamaño del conjunto de validación (test_size): dentro del conjunto de datos, estos se dividen en conjunto de entrenamiento y conjunto de validación, el cual se ejecuta sin conocer los resultados de cada fila. El tamaño del conjunto de validación se elige especificando el porcentaje del mismo sobre el conjunto total. Todas las filas que no pertenezcan a este pertenecerán al conjunto de entrenamiento.

Con el entrenamiento completamente configurado, podremos pasar a ejecutarlo. Durante la ejecución de la red tendremos los datos de éxito (accuracy) tanto para el conjunto de entrenamiento como para el de validación, si es que existe. No es necesario que haya un conjunto de validación, al menos en un principio, ya que puedes querer ver si el ajuste al conjunto de entrenamiento es suficientemente bueno antes de probarlo con un conjunto de validación.

Las primeras configuraciones que probamos no hacían uso de un conjunto de validación, siendo todos los datos parte del conjunto de entrenamiento, ya que para empezar hemos priorizado el aprendizaje de la red hasta que obtengamos resultados lo suficientemente buenos como para añadir el conjunto de validación.

Ahora vamos a comentar las ejecuciones realizadas sobre este modelo y la evolución del mismo. Los principales cambios realizados se basan en los datos utilizados y en los parámetros de configuración de la red. La configuración inicial es la siguiente:

- window_len: 48 (ya que así englobamos los dos días anteriores para cada dato)
- nº de épocas: 100
- batch_size: 32
- Función de activación: ReLU
- Optimizador: adam
- Una capa LSTM de 32 unidades

Esta configuración nos ofreció un ajuste al conjunto de entrenamiento del 54 %, que, siendo esta una red neuronal de respuesta binaria, indica que ha sido una respuesta aleatoria.

Viendo que la red ya podía ser ejecutada nos dedicamos a buscar métodos para aumentar su ajuste. Decidimos hacer un cambio de API para extraer los datos, pasando de utilizar Poloniex a utilizar CoinAPI[31], una nueva API que hemos encontrado y que utilizaremos durante todo el desarrollo que nos queda. Esta API nos aporta una nueva columna que muestra el total de transacciones realizadas en ese periodo.

Los datos que pedíamos seguían dividiéndose en intervalos de una hora, pero esta vez, revisando los datos de forma manual, detectamos muchos datos repetidos a lo largo de varias horas, lo que puede deberse a la falta de información por parte de la API. Por este motivo decidimos eliminar todo el año 2016 de los datos extraídos.

Solo con estos cambios, sin realizar ninguno más en la configuración detallada anteriormente, la red pasó de un 54 % de ajuste a un 65 %, lo que puede confirmarnos que el año 2016 ejercía de lastre para el aprendizaje del modelo.

Revisando algunos artículos, nuevamente en Medium, en el artículo comentado anteriormente "How to predict Bitcoin and Ethereum price with RNN-LSTM in Keras"[23], encontramos la posibilidad de añadir columnas con nueva información. Estas columnas son la volatilidad, que nos muestra la variación del precio con respecto al de apertura, el close off high, que indica la distancia del precio de cierre al más alto dado durante el periodo y el cambio, que es la diferencia entre el precio de apertura y el de cierre. A pesar de que los valores de estas columnas dependen directamente del valor de otras, la mejora fue significativa. Esta mejora puede deberse a que los datos entran a la red neuronal preprocesados. Su valor con respecto al precio es menor al normalizarse, lo que facilita el aprendizaje de la red.

```
1 def add_volatility(data):
2     kwargs = {'Change': lambda x: (x['Close'] - x['Open']) / x
          ['Open'],
3               'CloseOffHigh': lambda x: 2*(x['High'] - x['
                  Close']) / (x['High'] - x['Low']) - 1,
4               'Volatility': lambda x: (x['High'] - x['Low']) /
                    (x['Open'])}
5     data = data.assign(**kwargs)
6     return data
```

Listing 5.4: Cálculo de las nuevas columnas y su adición al conjunto de datos.

Nuevamente, sin cambiar ninguno de los datos de la configuración, solo con las nuevas columnas añadidas, pasamos del último 65 % de ajuste a un 75 %. Además de la mejora conseguida, nos dimos cuenta de que el ajuste seguía aumentado en gran medida incluso en las últimas épocas, lo que no pasaba en los casos anteriores, donde el resultado dejaba de aumentar bastante antes de llegar al final de la ejecución. Por este motivo decidimos aumentar el número de épocas de 100 a 200, consiguiendo el siguiente resultado:

- **loss**: 0.1079
- **mae**: 0.219
- **acc**: 0.8499

Con estos buenos resultados sobre la mesa, decidimos realizar variaciones en algunos de los parámetros de configuración de la red, en concreto del batch_size y del window_len. Como comentamos anteriormente, un batch_size de menor tamaño significa un entrenamiento más exhaustivo. Esto se confirmó probando con uno mayor, lo que nos llevó a un peor resultado, en concreto de un 69,6 % de ajuste. Decidimos mantener su valor en 32. No realizamos pruebas con valores inferiores a 32 debido al gran incremento en el tiempo de ejecución que esto suponía.

Por otro lado, probamos variando también el valor de window_len, empeorando también los resultados tanto al aumentarlo como al disminuirlo. Esto puede deberse a que, si los datos están muy separados en el tiempo hay poca relación entre ellos, y si la cantidad de datos es muy reducida, no aportan suficiente información. Debido a este empeoramiento decidimos mantener su valor en 48.

Otra variación que comprobamos fue en el número de unidades de la capa LSTM que

estábamos utilizando, en la cual no vimos cambios significativos en los resultados, manteniéndose entre el 84 % y el 86 % de ajuste.

Antes de utilizar un conjunto de validación decidimos aumentar el tamaño del conjunto total. Para ello, pasamos de coger los datos en intervalos de una hora a cogerlos en intervalos de 5 minutos, con lo que obtuvimos aproximadamente 233 mil filas, doce veces más que las que teníamos anteriormente. Como sabíamos que esto incrementaría considerablemente el tiempo de ejecución, decidimos poner en funcionamiento la ejecución sobre la GPU. Este fue el momento en el que decidimos utilizar anaconda, para poder utilizar el módulo tensorflow-gpu sin que nos dieran problemas de compatibilidad, como comentamos en la sección dedicada al entorno de ejecución. Para conseguir mejorar el tiempo de ejecución pasamos de utilizar capas LSTM a capas CuDNNLSTM[32], que hace uso de una implementación de CuDNN para aprovechar mejor el uso de los núcleos de la tarjeta gráfica.

Para las próximas ejecuciones decidimos aumentar el valor del parámetro batch_size de 32 a 128, debido al aumento de datos para reducir el tiempo de ejecución. Para la primera ejecución después de todos los cambios anteriores, la configuración utilizada fue la siguiente:

- window_len: 48
- nº de épocas: 200
- batch_size: 128
- Función de activación: ReLU
- Optimizador: adam
- Una capa CuDNNLSTM de 32 unidades

Con esta nueva configuración, el resultado bajó a un 75,59 % de ajuste al conjunto de entrenamiento.

Al haber cambiado el intervalo entre cada una de las filas de una hora a 5 minutos, mantener el valor de window_len en 48 ya no agrupaba en cada fila los valores de los últimos 2 días, sino que recogía las últimas 4 horas. Por esto decidimos cambiar su valor a 288 para volver a englobar esos últimos dos días para cada fila del conjunto de datos. Esta medida nos hizo recuperar los buenos resultados de antes, en concreto un resultado de 83,52 % de ajuste.

Con los buenos resultados obtenidos y una red con más datos y en apariencia mejor estructurada que al principio, decidimos ejecutar el modelo también sobre un conjunto de validación, dedicando a este el 20 % del tamaño del conjunto total.

- **loss**: 0.1008
- **mae**: 0.2035
- **acc**: 0.8713
- **seg/ep**: 4
- **v_loss**: 0.2181
- **v_mae**: 0.3386
- **v_acc**: 0.7151

También nos dimos cuenta de que ya no eran necesarias 200 épocas para alcanzar los mejores resultados, por lo que decidimos volver a ejecutar 100 épocas. Estos resultados han sido muy satisfactorios para nosotros y, a partir de este momento, decidimos buscar nuevas formas para mejorar los resultados.

El primer cambio que probamos fue la normalización de los datos utilizados ya que, como comentamos anteriormente, simplifican a la red su procesamiento. Esta medida significó un empeoramiento de los resultados, pasando a resultados alrededor del 70 % en el conjunto de entrenamiento y del 50 % sobre el conjunto de validación. Esto puede deberse a una pérdida de precisión en los datos cuando pasan a estar entre 0 y 1. Por tanto, decidimos eliminar la normalización de los datos.

A continuación, probamos a añadir una segunda capa CuDNNLSTM de 32 unidades, añadiendo así profundidad a la red, por lo que los datos se procesarían más veces y, como consecuencia, la red se podría ajustar más a ellos. También decidimos hacer cambios en el optimizador del modelo, probando entre varios de los optimizadores que nos ofrece Keras. Los optimizadores probados han sido: adam, rmsprop, nadam, sgd y adagrad.

Con el optimizador adam, que era el que habíamos estado utilizando hasta el momento, la segunda capa añadida no aportó ninguna mejora a la red. Ninguno de los nuevos optimizadores hizo que mejorasen los resultados previos, aunque sí observamos que todos los optimizadores de la familia del adam (nadam, adagrad y adam) obtenían resultados muy similares. Después de estas pruebas decidimos mantener el optimizador adam.

Posteriormente decidimos cambiar la función de activación de la última capa de una ReLU a una LeakyReLU que, como se puede ver en la figura 5.1, al asignar valores próximos a 0 para los datos negativos, pero no 0, impide que haya neuronas que no estén trabajando. A pesar del cambio, tampoco hubo ninguna mejora en los resultados.

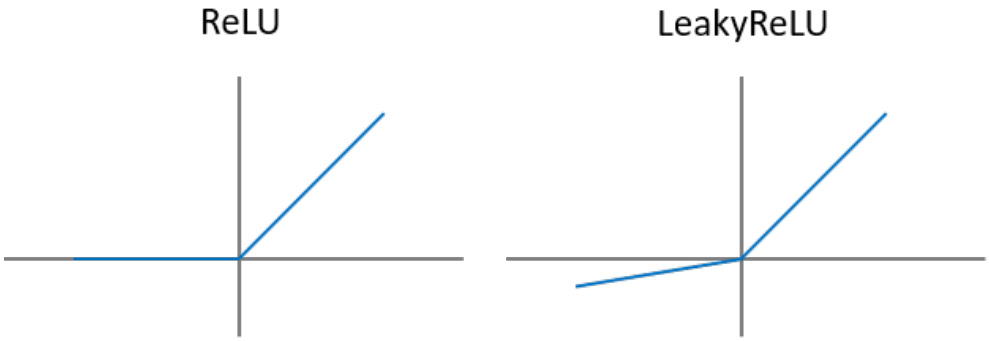

Figura 5.1: Diferencia entre la función ReLU y LeakyReLU.

Como los cambios en la configuración no nos estaban dando ningún resultado, decidimos añadir nuevas columnas al conjunto de datos.

Inicialmente añadimos la columna EMA (Exponential Moving Average), que indica la media móvil exponencial. Pese a utilizar varios métodos para calcularla, entre ellos uno propio de Pandas, ninguno nos dio mejora alguna en los resultados. Posteriormente añadimos la columna MACD (Moving Average Convergence Divergence), que indica la media móvil de convergencia y divergencia, que es la diferencia entre el EMA de un periodo corto y el de un periodo largo, que normalmente suelen ser 12 y 26 periodos, respectivamente.

Finalmente añadimos las columnas MACD y RSI (Relative Strength Index), el cual nos muestra cómo de fuerte es un precio con respecto al anterior. Estos nuevos datos tampoco significaron ningún cambio relevante en los resultados del conjunto de entrenamiento y del conjunto de validación.

A falta de ideas para conseguir una mejora en los resultados, decidimos hacer una revisión del curso de Redes Neuronales y Deep Learning que realizamos en la plataforma Coursera. Después de esta revisión no encontramos ninguna idea notable para aplicar al modelo. Lo más significativo que encontramos fue la recomendación de utilizar una función de activación sigmoide en la última capa, ya que esta devuelve los valores entre 0 y 1 y es preferible para redes neuronales de respuesta binaria.

Por otro lado, añadimos al modelo una capa de dropout del 25 %, lo que hace que ese porcentaje de las unidades de una capa no se utilice durante cada una de las ejecuciones de la red. Las unidades inutilizadas se eligen de forma aleatoria para evitar que la red neuronal se ajuste demasiado al conjunto de entrenamiento, lo que impediría que se pudiera ejecutar sobre nuevos datos. A este fenómeno se le conoce como sobreentrenamiento.

Debido a la poca variación en los resultados obtenidos decidimos dar por finalizado el desarrollo de la red neuronal de respuesta binaria. La configuración final de la red fue la siguiente:

- window_len: 288
- nº de épocas: 100
- batch_size: 128
- Función de activación: sigmoide
- Optimizador: adam
- Una capa CuDNNLSTM de 32 unidades

5.2.1. Conclusión

Esta configuración nos ha dado unos resultados finales de, aproximadamente, el 85 % de ajuste al conjunto de entrenamiento y del 70 % de ajuste al conjunto de validación. Estos resultados pueden deberse principalmente a la simplicidad de la red, siendo una red de respuesta binaria más propensa a obtener buenos resultados. A lo largo de la próxima sección veremos cómo evolucionan los resultados complicando la red a seis posibles salidas en vez de las dos posibles que hemos contemplado en este apartado.

También queremos recalcar la dificultad de encontrar datos de mejor calidad, ya que es necesario invertir una cantidad de dinero bastante elevada para poder obtenerlos. Sería un trabajo interesante poder utilizar alguna de ellas, ya que pueden llegar a contener muchísimos más datos de utilidad.

5.2.2. Conclusion

This configuration results have an adjustment of around 85 % to the training set and 70 % to the validation set. The simplicity of the network may be the reason for these results since a binary response network is more likely to produce good results. Throughout the next section, we will see the results evolution when the network complexity increases to six possible outputs instead of the two outputs we have contemplated in this section.

We want also to emphasize the difficulty of finding better quality data, since it is necessary to invest a large amount of money to obtain them. In the future, it would be an interesting job to use quality data. Quality data contains many more useful data.

5.3. Respuesta no binaria

Una vez concluimos que los resultados de nuestra red con respuesta binaria eran sólidos y no tenían mucho margen de mejora decidimos crear una red con una salida más compleja. Esta idea la desarrollamos durante el transcurso del curso de Microsoft al que asistimos.

El objetivo era indicar si va a subir o bajar el precio en el instante siguiente, al igual que en la binaria, pero con tres niveles cada una de las respuestas: bajo, medio y alto.

El nivel bajo indica una subida/bajada de menos de 50 dólares con respecto al instante anterior. El nivel medio indica una subida/bajada mayor o igual a 50 dólares y menor de 200 dólares con respecto al anterior. Y el nivel alto indicaba una diferencia de precio de 200 dólares o más.

Este tipo de salida supone un gran impacto en el entrenamiento de la red, ya que los datos que antes servían para entrenar una respuesta binaria, ahora tienen que entrenar una respuesta con seis salidas posibles.

Para configurar la red solo tuvimos que reescribir el método “createInputsAndOutputs”, que pasó a llamarse “createNonBinaryInputsAndOutputs”, de forma que tuviera en cuenta esta modificación.

```
1  def createNonBinaryInputsAndOutputs(data,window_len):
2      # -window_len first rows that we can't use
3      # -1 last row also we can't use
4      inputs = np.zeros((data.shape[0] - window_len, data.shape[
           1], window_len))
5      outputs = np.zeros((data.shape[0] - window_len, 1))
6
7      for i in range(data.shape[0] - window_len - 1):
8          inputs[i] = data[i:i+window_len].T
9
10         # Price down more than 200$
11         if(data[i+window_len,0] - data[i+(window_len-1),0] <=
               -200):
12             outputs[i] = 1
13         # Price down between 200$ and 50$
```

```
14          elif(data[i+window_len,0] - data[i+(window_len-1),0]
              <= -50):
15              outputs[i] = 2
16          # Price down between 50$ and 0$
17          elif(data[i+window_len,0] - data[i+(window_len-1),0]
              <= 0):
18              outputs[i] = 3
19          # Price up between 0$ and 50$
20          elif(data[i+window_len,0] - data[i+(window_len-1),0]
              <= 50):
21              outputs[i] = 4
22          # Price up between 50$ and 200$
23          elif(data[i+window_len,0] - data[i+(window_len-1),0]
              <= 200):
24              outputs[i] = 5
25          # Price up more than 200$
26          else:
27              outputs[i] = 6
28
29      return inputs, outputs
```

Listing 5.5: Estado del método createInputsAndOutputs tras la modificación en la salida de la red neuronal.

Al principio ejecutamos la red con la misma estructura que estaba configurada para la respuesta binaria salvo el uso de una función de activación LeakyReLU en vez de una sigmoide, ya que el problema ya no era binario, por lo que la configuración pasó a ser la siguiente:

- window_len: 288
- nº de épocas: 100
- batch_size: 128
- Función de activación: LeakyReLU
- Optimizador: adam
- Una capa CuDNNLSTM de 32 unidades

Cuando ejecutamos la red vimos que no era capaz de ajustarse correctamente, dando unos resultados muy pobres:

- **loss**: 0.1108
- **mae**: 0.8025
- **acc**: 0.4214
- **seg/ep**: 4
- **v_loss**: 2.4198
- **v_mae**: 1.2223
- **v_acc**: 0.3074

Pensamos entonces que el problema se podía encontrar en que la red no tenía la complejidad necesaria como para procesar datos tan complejos y ajustarse a ellos. Es por esto que probamos a añadir primero una capa CuDDNLSTM más y, en la siguiente ejecución, una tercera capa CuDNNLSTM.

Al añadir una capa sí obtuvimos una mejora en los resultados, que pasaron de un 42 % de ajuste al conjunto de entrenamiento a un 49 %, añadiendo dos segundos más al tiempo por época. Aun así, siguen siendo números muy bajos que no permiten trabajar con ellos, por lo que probamos añadiendo una capa más.

Cuando incluimos la tercera capa oculta comprobamos que, no solo no mejoraba, sino que empeoraba, pasando de un 49 % a un 48 %, añadiendo un segundo más al tiempo por época.

Decidimos añadir datos que habíamos descartado cuando la red neuronal devolvía resultados binarios, los indicadores MACD y RSI. Pensamos que ahora que disponen de muchas menos filas por salida al pasar de dos a seis, si añadíamos más columnas ayudaría al entrenamiento de la red. De nuevo, no solo no mejoró, sino que empeoró, pasando a tan solo un 44 % de ajuste al conjunto de entrenamiento.

Por último, quisimos ver si el problema residía en la memoria a largo plazo de las capas LSTM, que guardara datos de hace mucho tiempo que no fueran suficientemente relevantes para predecir el instante siguiente, y eso hiciera que fallara la predicción. Para ello elegimos la implementación de la versión simplificada de una CuDNNLSTM, la CuDNNGRU. Cambiamos las dos capas LSTM por capas GRU y ejecutamos la red neuronal. El resultado fue equivalente al que sacó la red con tan solo una capa CuDNNLSTM, ya que tuvo un porcentaje de ajuste de solo el 43 %, con 6 segundos por época, dos más que en la configuración de una capa.

Para descartar completamente que el problema se encontraba en la configuración de la red que teníamos, creamos un script que ejecutaba la red con una combinación de los siguientes hiperparámetros:

```
1     batch_size_array = [64, 128, 256]
2     epochs = 100
3     window_len = 288
4     activation_array = ["leakyRelu", "elu", "selu", "softplus"
          , "softsign", "relu", "tanh", "sigmoid", "exponential",
          "linear"]
5     units_array = [64, 128]
6     test_size = 0.2
```

Listing 5.6: Hiperparámetros utilizados en la ejecución del script.

Este script ejecutaba una por una todas las combinaciones existentes sobre una red formada por dos capas CuDNNLSTM, el tipo de red neuronal recurrente que mejores resultados nos había proporcionado.

```
1 def createModel(input_shape, output_size, units, activation):
2     inp = Input(shape=input_shape)
```

```
3
4       lstm1 = CuDNNLSTM(
5           units=units,
6           return_sequences=True
7       )(inp)
8
9       if activation == "leakyRelu":
10          activation1 = LeakyReLU(alpha=0.1)(lstm1)
11      else:
12          activation1 = Activation(activation)(lstm1)
13      dropout1 = Dropout(rate=0.25)(activation1)
14
15      lstm2 = CuDNNLSTM(
16          units=units
17      )(dropout1)
18
19      if activation == "leakyRelu":
20          activation2 = LeakyReLU(alpha=0.1)(lstm2)
21      else:
22          activation2 = Activation(activation)(lstm2)
23      dropout2 = Dropout(rate=0.25)(activation2)
24
25      dense = Dense(units=output_size)(dropout2)
26      if activation == "leakyRelu":
27          activation3 = LeakyReLU(alpha=0.1)(dense)
28      else:
29          activation3 = Activation(activation)(dense)
30
31      model = Model(inputs=inp, outputs=activation3)
32      model.compile(loss='mse', optimizer='adam', metrics=['mae'
            ,'acc'])
33      model.summary()
34      return model
```

Listing 5.7: Método que crea el modelo de la red neuronal.

Cada vez que terminaba el entrenamiento de una combinación, escribíamos los resultados en un archivo .csv.

Al finalizar el script pudimos comprobar cuáles eran las mejores combinaciones de hiperparámetros para nuestro tipo de datos. Los conjuntos de hiperparámetros que mejor ajuste nos proporcionaron fueron:

1. batch_size: 64, activation: selu y units: 128. Con un 64,3 % de ajuste al conjunto de entrenamiento.

2. batch_size: 64, activation: linear y units: 128. Con un 64,2 % de ajuste al conjunto de entrenamiento.

3. batch_size: 64, activation: LeakyReLU y units: 128. Con un 63,9 % de ajuste al conjunto de entrenamiento.

Como esperábamos los resultados con un batch_size menor fueron mejores, al igual que los resultados con mayor número de unidades. Pero el hiperparámetro más decisivo resultó ser la función de activación que utilizamos.

En el primer caso los resultados pueden deberse a la función de activación selu, que significa Scaled Exponential Linear Unit[33]. Esta función tiene el mismo objetivo que la LeakyReLU: reducir las probabilidades de que algunas de las neuronas de la red mueran a causa de pesos muy bajos. Mientras que la LeakyReLU es lineal en sus valores negativos, la selu aplica un factor escalar y otro exponencial, de esta forma, impide que las neuronas mueran (gracias al escalar), y al mismo tiempo soluciona un problema que tiene la LeakyReLU: el problema del desvanecimiento del gradiente[34]. El desvanecimiento del gradiente se da cuando una red recalcula muchas veces la matriz de pesos, ya que está calculando la derivada de datos menores a uno, por lo que cada vez que recalcula, la derivada es menor, hasta estar muy próxima a cero. Esto hace que aprenda menos. Para solucionarlo la selu aplica un factor exponencial que hace que llegar a ese punto sea mucho más difícil. El segundo caso nos sorprendió más, ya que aplicar una función de activación lineal es equivalente a no utilizar ninguna.
Finalmente, los resultados del tercer caso son los esperados, ya que la función de activación LeakyReLU es la que mejores resultados globales nos ha dado y el resto de hiperparámetros son iguales a los anteriores.

5.3.1. Conclusión

Viendo los resultados que nos dio la red neuronal no binaria y una vez descartado un problema en la configuración de la red, concluimos que el problema residía en el conjunto de datos.

Los principales problemas que tiene el conjunto de datos son: el pequeño tamaño, que impide que se ajuste mejor al no tener datos suficientes por cada posible salida; y que los datos que contiene puede que no sean lo suficientemente relevantes como para ser capaz de predecir la evolución del precio en base a ellos.

El problema del tamaño del conjunto ahora mismo no lo podemos solucionar, ya que hemos recopilado todos los datos históricos de la evolución de la moneda que nos permitía la versión gratuita de la API, con los intervalos de tiempo más pequeños posibles.

Con respecto a la relevancia de los datos, la API que hemos elegido es la que nos proporciona todos los datos que nos dan las demás APIs gratuitas que hay y columnas extra, como el número de transacciones en cada intervalo de tiempo.

Por tanto, nuestra conclusión final es que, teniendo en cuenta las restricciones actuales: que no hay más datos históricos y que ninguna API gratuita proporciona más columnas por dato, entrenar una red neuronal que prediga la evolución del precio de una criptomoneda con resultados no binarios no es viable.

5.3.2. Conclusion

After studying the results from the non-binary neural network and once we ruled out the problem on the network configuration, we reach the conclusion that the problem resides

in the used dataset.

The main problems we found with the dataset are its small size, which prevents it from adjusting better by not having enough data for each possible output; and that the data it contains may not be relevant enough to be able to predict the price evolution based on them.

The problem with the dataset size we cannot solve it right now. We have compiled all the historical data of the currency evolution allowed by the free version of the API, with the smallest possible time intervals.

With respect to the data relevance, the API we have chosen is the one that provides all the data that the other free APIs plus extra columns, such as the number of transactions in each time interval.

Therefore, our conclusion is that, taking into account the current restrictions: that there is no more historical data and that no free API provides more columns per data, training a neural network that predicts the evolution of a cryptocurrency price with non-binary results is not viable.

5.4. Desarrollo futuro

Viendo que los datos que podemos obtener de forma gratuita de una API no son muy relevantes de cara a predecir la evolución del precio de una criptomoneda hemos estado investigando posibles eventos o indicadores que tengan una gran correlación con el precio del Bitcoin.

Gracias al artículo de Jethin Abraham, Daniel Higdon, John Nelson y Juan Ibarra "Cryptocurrency Price Prediction Using Tweet Volumes and Sentiment Analysis"[35] hemos podido comprobar que el número de tweets publicados relacionados con el Bitcoin y los resultados de las tendencias de Google están altamente relacionados con la evolución del precio del Bitcoin.

El coeficiente de correlación de Pearson, o Pearson R indica que la cantidad de tweets publicados relativos al Bitcoin tienen una correlación del 84,1 %, mientras que las tendencias de Google tienen una correlación del 81,7 %.

Una de las posibles mejoras a implementar en nuestra red neuronal es la recopilación histórica de estos datos y su posterior utilización en el entrenamiento de la misma.

5.4.1. Barrera monetaria

Uno de los principales problemas a la hora de desarrollar este trabajo es que si quieres más datos que los que son públicos solo los puedes obtener invirtiendo dinero mediante el pago de licencias de APIs, Twitter Developers o Google Developers.

Es por esto que, si no logramos disponer de un presupuesto para poder pagar estas licencias, la mejora de la red neuronal va a ser muy complicada.

Aportaciones individuales

Teniendo en cuenta que el ámbito de nuestro proyecto era completamente nuevo para nosotros, al principio necesitamos investigar sobre los conceptos básicos relativos a las redes neuronales y las criptomonedas. Para ello decidimos que lo mejor era una primera fase de trabajo individual en la que cada uno recopilara información sobre un tema y, posteriormente, lo pusiera en común con el otro.

Después decidimos empezar una fase de aprendizaje ya enfocado a redes neuronales. Esto comprende los cursos que realizamos en la plataforma Coursera, y la asistencia al curso de Microsoft.

Una vez tuvimos una base sobre la que empezar a trabajar, nos dividimos algunas tareas, teniendo en cuenta que al final de cada día de trabajo poníamos todo en común y desarrollábamos este trabajo en base a las conclusiones que habíamos sacado cada uno de forma individual.

Manuel Pastor Cobo

Mi primera tarea individual en este trabajo fue la recopilación de documentos que pudieran sernos útiles para el desarrollo de nuestro proyecto, ya estuvieran relacionados con redes neuronales, con criptomonedas o con la obtención de datos. Esta fase me llevó alrededor de una semana, en la cual conseguí documentos de los principales repositorios de artículos científicos y tecnológicos, como puede ser arxiv[36] o IEEE Xplore[37]. También utilicé las bases de datos de artículos e investigaciones de algunas universidades como la Universidad de Stanford o el Instituto Tecnológico de Masachusets (MIT).
Una vez recopilados, nos repartimos los documentos y cada uno nos leímos los que nos correspondían, poniendo en común las conclusiones cuando terminamos esta tarea.

Mientras mi compañero se encargaba de investigar las bibliotecas de Python que necesitábamos para el formateo de datos y su funcionamiento, yo me encargué de investigar cuáles eran las mejores APIs de obtención de datos de criptomonedas. En esta fase fue cuando vi que nos interesaba desarrollar nuestro trabajo sobre Bitcoin ya que, aparte de ser la criptomoneda predominante en el mercado, cualquier API de datos de criptomonedas te permite extraer información sobre ella. Las APIs que mayor cantidad de datos nos daban eran la de Bitcoincharts[38], la de OKCoin[12], la de Poloniex[15] y la de CoinAPI[31].
La API de Bitcoincharts la descarté rápidamente ya que, aunque nos daba todos los datos históricos del Bitcoin, su documentación era muy pobre y cada fila solo contenía la fecha, el precio y el volumen de monedas movido en ese intervalo de tiempo. En esta misma API encontré que también te permitía obtener más datos por columnas si, en vez de descargarte el archivo CSV con todos los datos históricos, realizabas una petición http a

su API. El problema que me encontré fue que no te permitía solicitar datos que fueran de hace más de cinco días.
La de OKCoin es una API REST que también rechacé rápidamente. El principal motivo fue que no te permitía solicitar más de 2000 entradas de datos al día, por lo que no nos iba a ser posible obtener todos los datos que íbamos a necesitar en este trabajo.
La API de Poloniex nos permitía recopilar 50000 filas en cada petición que realizábamos con la licencia gratuita, pudiendo realizar una de estas peticiones cada seis segundos. Esta API nos resultó muy atractiva y fue con la que realizamos la demo inicial de nuestra red neuronal, aunque posteriormente la cambiáramos por la de CoinAPI, que nos permitía hacer una petición al día de tantos datos como quisiéramos.

Una vez terminé la recopilación de datos me centré en estudiarme los apuntes sobre Python que había hecho mi compañero y en hacerme múltiples ejemplos probando la sintáxis básica y las bibliotecas de tratamiento de datos que había encontrado durante su fase de investigación.
Ya con los datos obtenidos y con una base de Python ambos nos centramos en ver cómo realizar el tratamiento, ya que ambos teníamos problemas a la hora de formatearlos con una estructura que resultara sencilla.

El siguiente paso que decidimos hacer fue el curso básico sobre redes neuronales de Coursera, impartido por un profesor de la Universidad de Stanford. El desarrollo de este curso está descrito en el capítulo 3 de esta memoria.
Después del curso básico quisimos obtener conocimientos un poco más profundos sobre redes neuronales, por lo que decidimos estudiarnos algunos de los capítulos del curso avanzado de redes neuronales que impartía el mismo profesor del curso anterior en la misma plataforma. Al final, este segundo curso no logró ayudarnos demasiado al centrarse poco en la parte práctica de las redes neuronales.

Cuando consideramos que ya teníamos conocimientos suficientes como para empezar a desarrollar decidimos crear una red neuronal básica para ver si conseguíamos que funcionara y para ver cómo se aplicaban esos conocimientos en Keras.

Una vez tuvimos una primera red de demostración me dediqué a buscar información sobre los indicadores que más se utilizan en criptomonedas para comerciar con ellas. Encontré que la volatilidad, el MACD y el RSI eran los más comunes al ser los que más información relevante daban.
Estos indicadores los incorporamos a nuestro código en la fase de desarrollo, la cual hicimos siempre en grupo, salvo la implementación del script en el apartado de respuesta no binaria. Toda la información sobre esta fase se encuentra en el capítulo 3 de esta memoria.

Finalmente, mientras mi compañero se encargaba de codificar el script que probaba diferentes hiperparámetros en la parte no binaria, yo me encargué de buscar posibles vías de mejora de este trabajo en el futuro. Me centré en buscar trabajos sobre criptomonedas que utilizaran otro tipo de datos. El artículo que más me ayudó fue el de Jethin Abraham, Daniel Higdon, John Nelson y Juan Ibarra “Cryptocurrency Price Prediction Using Tweet Volumes and Sentiment Analysis”[35]. Este artículo muestra cómo utilizan información sobre criptomonedas recopilada de plataformas como Twitter o los datos de tendencias de Google para realizar prediciones de precio.

Aquí termina mi aportación individual a este proyecto y continúa la de mi compañero.

Pablo de Torre Barrio

La primera tarea que he realizado en este trabajo ha sido un acercamiento al lenguaje de programación utilizado, Python. Como comentamos a lo largo de la memoria, ninguno de los miembros hemos hecho uso antes de este lenguaje, por lo que vimos necesario hacer una primera aproximación a él.

Entre los conceptos básicos aprendidos fue de gran importancia el manejo de los arrays, de los que hemos hecho un gran uso a lo largo de todo el desarrollo. Una vez asimilados estos conceptos básicos decidimos pasar al tratamiento de datos.

Esta tarea la realicé en conjunto con mi compañero, mientras él se encargaba de acceder a la API seleccionada para obtener los datos que necesitábamos, yo fui haciendo pruebas con un conjunto de prueba para su posterior formateo. Este tratamiento se basa principalmente en la selección de las columnas útiles dentro del conjunto devuelto por la API y de la posterior creación de la columna con los datos de salida.

Posteriomente, decidí dedicar algo de tiempo al estudio de los conceptos de las criptomonedas. Para ello hice uso de algunos de los documentos recopilados por mi compañero, que también me ayudó de forma personal. Estos conceptos nos son útiles para el entendimiento de los datos obtenidos.

Una vez adquirida una base sobre Python y criptomonedas, ambos decidimos comenzar con el aprendizaje sobre redes neuronales, para ello realizamos un curso básico sobre ellas en la plataforma Coursera, el cual está descrito en el capítulo 3 de la memoria. El tiempo dedicado a este curso y al siguiente, más avanzado sobre el temario, nos llevó alrededor de 3 semanas.

El curso avanzado no lo realizamos en su totalidad, sino que elegimos algunos capítulos que consideramos de mayor utilidad para el desarrollo de nuestro trabajo. Finalmente no fueron de gran ayuda al ser demasiado especializados en el funcionamiento interno de las redes neuronales.

Después de las tareas realizadas hasta el momento, nos decidimos a hacer, ambos compañeros de manera conjunta, una demostración implementando una red neuronal básica, haciendo en ella uso de la API de datos creada anteriormente y los conocimientos adquiridos sobre redes neuronales para su configuración mediante Keras.

Con Keras funcionando sobre nuestro equipo, me dediqué a preparar el entorno para su funcionamiento con la GPU. Esto no fue una tarea fácil, como comentamos en la sección 4.1, debido a los problemas de compatibilidad del sistema operativo con las bibliotecas de Nvidia, y fue gracias a la investigación a lo largo de varios artículos donde encontramos una configuración compatible.

Con el desarrollo del trabajo en sí comenzado, en la que ambos debatíamos la configuración y los datos utilizados en cada caso de prueba, y analizábamos juntos la salida, me dediqué especialmente a probar distintas configuraciones de hiperparámetros. Estas configuraciones y las decisiones tomadas sobre ellas se encuentran en las secciones 5.2 y 5.3.

Para el desarrollo de la red no binaria, me encargué de realizar el script de automatización de pruebas que describimos en la sección 5.3. Este desarrollo fue muy sencillo ya que solo fue necesario añadir al código los bucles correspondientes a cada hiperparámetro que íbamos a probar.

Finalmente, con el trabajo concluido y el desarrollo de la memoria bastante avanzado, me encargué de la exportación de la misma a la plataforma Overleaf[39], donde hemos finalizado la redacción y hemos hecho uso de LaTeX[40] para el formato de la memoria. Para su creación hemos hecho uso de la plantilla TeFlon, proporcionada por la Facultad de Informática, y hemos hecho los cambios necesarios para que se adaptara a nuestras necesidades.

Bibliografía

[1] Unidad lstm por https://commons.wikimedia.org/wiki/user:biobserve (raster version previously uploaded to wikimedia) alex graves, abdel-rahman mohamed, y geoffrey hinton (original) eddie antonio santos - https://commons.wikimedia.org/wiki/file:long_short_term_memory.png alex graves, abdel-rahman mohamed, y geoffrey hinton. speech recognition with deep recurrent neural networks. in acoustics, speech and signal processing (icassp), 2013 ieee international conference on, pages 6645–6649. ieee, 2013., cc by-sa 4.0, https://commons.wikimedia.org/w/index.php?curid=59931189.

[2] Estructura interna de una unidad lstm por françois deloche – trabajo propio, cc by-sa 4.0, https://commons.wikimedia.org/w/index.php?curid=60149410.

[3] Estructura interna de una unidad gru por françois deloche – trabajo propio, cc by-sa 4.0, https://commons.wikimedia.org/w/index.php?curid=60466441.

[4] Molly Jane Zuckerman. Brazo de capital de riesgo de rockefeller, venrock, se asocia con coinfund y ejecutivos se enfocan en largo plazo. *cointelegraph*, 2018.

[5] Curso machine learning de coursera impartido por la stanford university. https://www.coursera.org/learn/neural-networks-deep-learning.

[6] Curso machine learning de coursera impartido por la stanford university. https://www.coursera.org/learn/machine-learning.

[7] Usuario BackStreetCode. Regresión lineal con gradiente descendente. *Medium*, 2017.

[8] Microsoft azure machine learning studio. https://azure.microsoft.com/es-es/services/machine-learning-studio/.

[9] Bitcoin. https://bitcoin.com.

[10] Isaac Madan, Shaurya Saluja, and Aojia Zhao. Automated bitcoin trading via machine learning algorithms. *Stanford University*.

[11] Api de coinbase. https://developers.coinbase.com/.

[12] Okcoin. https://www.okcoin.com/docs/en/.

[13] Brandon Ly, Divendra Timaul, Aleksandr Lukanan, Jeron Lau, and Erik Steinmetz. Applying deep learning to better predict cryptocurrency trends. *Augsburg University*.

[14] Sean McNally. Predicting the price of bitcoin using machine learning. *National College of Ireland*.

[15] Poloniex. https://docs.poloniex.com/#introduction.

[16] Tensorflow. https://www.tensorflow.org/.

[17] Cuda. https://developer.nvidia.com/cuda-toolkit.

[18] Cudnn. https://developer.nvidia.com/cudnn.

[19] Anaconda. https://www.anaconda.com/distribution/.

[20] Usuario Harveen Singh. Tensorflow gpu installation made easy: Use conda instead of pip. *Medium*, 2018.

[21] Python. https://www.python.org/.

[22] Apache spark. https://spark.apache.org/.

[23] Usuario Siavash Fahimi. How to predict bitcoin and ethereum price with rnn-lstm in keras. *Medium*, 2018.

[24] Pandas. http://pandas.pydata.org/about.html.

[25] Numpy. http://www.numpy.org/.

[26] Keras. https://keras.io/.

[27] Theano. http://deeplearning.net/software/theano/.

[28] Urllib. https://docs.python.org/2/library/urllib.html.

[29] Apartado de funciones de activación del curso neuronal networks and deep learning. https://www.coursera.org/lecture/neural-networks-deep-learning/activation-functions-4dDC1.

[30] Jason Brownlee. Gentle introduction to the adam optimization algorithm for deep learning. *Machine Learning Mastery*, 2017.

[31] Coinapi. https://www.coinapi.io/.

[32] Capa cudnnlstm de keras. https://www.tensorflow.org/api_docs/python/tf/keras/layers/CuDNNLSTM.

[33] Günter Klambauer, Thomas Unterthiner, and Andreas Mayr. Self-normalizing neural networks. *Johannes Kepler University Linz*, 2017.

[34] Adrián Hernández. Problema del desvanecimiento del gradiente (vanishing gradient problem). *Meta-learning*.

[35] Jethin Abraham, Daniel Higdon, John Nelson, and Juan Ibarra. Cryptocurrency price prediction using tweet volumes and sentiment analysis. *National College of Ireland*.

[36] Arxiv. https://arxiv.org/.

[37] Ieee xplore. https://ieeexplore.ieee.org/Xplore/home.jsp.

[38] Bitcoincharts api. https://bitcoincharts.com/about/markets-api/.

[39] Curso machine learning de coursera impartido por la stanford university. https://es.overleaf.com/.

[40] Latex. https://www.latex-project.org/.

PASCAL
ENERO 2018
Ult. actualización 31 de mayo de 2019
LaTeX lic. LPPL & powered by TEFLON CC-ZERO