UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Práctica 1. Arquitecturas Supervisadas (BPNN) con Python

Asignatura: Sistemas Inteligentes

Resumen:

Implementación de una red neuronal Backpropagation tanto en Keras como desde cero.

Carlos Brito Pérez

Pablo Morales Gómez

Grado en: Ingeniería Informática. 4º Curso

ÍNDICE

INTRODUCCIÓN

El objetivo de esta práctica era familiarizarse con las redes neuronales BackPropagation, para esto debíamos implementar desde cero una. Esta red debía entrenarse para ser capaz de, a través de una serie de parámetros medioambientales y físicos, si un coche tendrá un accidente o no. Además, se nos pedía realizar el mismo trabajo, pero esta vez con ayuda de la librería de Keras y comparar los resultados.

FUNCIONAMIENTO

Para el correcto funcionamiento del programa, lo primero es importar la clase main y ejecutar la función keras_vs_numpy(Option). A esta función hay que pasarle un número como parámetro, este valor sirve para elegir que versión de la red que se va a utilizar.

Option = 0 → keras y numpy, dando una comparativa entre ambas al acabar
Option = 1 → keras
Option = 2 → Numpy

PREPROCESADO DE LOS DATOS

El primer obstáculo que nos encontramos en este trabajo fue el dataset con el que debíamos trabajar que presentaba varias situaciones con las que tuvimos que lidiar que desarrollaremos a continuación.

Inicialmente vimos que el dataset presentaba una serie de columnas con datos de tipo *Categorical* (*"ESTADO_CARRETERA"* o *"TIPO_PRECIPITACION"* entre otros) así que lo primero que hicimos fue asignarle un valor numérico a cada una de las categorías que poseían las columnas implicadas. Además, por la complejidad que entrañaba el someter a este proceso la primera columna (*"FECHA_HORA"*) se decidió prescindir de ella.

Por otro lado, en muchas instancias de la tabla presentaba varias celdas vacías, notamos que este fenómeno tendía a ocurrir con mayor frecuencia cuando nevaba en la carretera en

cuestión. Por este motivo, ya que no queríamos prescindir de la mayor parte de los ejemplos en los que el fenómeno meteorológico fuera la nieve decidimos rellenar estos *missing values* con la media de la columna afectada.

Durante el transcurso del trabajo el rendimiento de nuestra red se vio muy afectada por la alta desproporción que existía entre los casos en los que había ocurrido un accidente (un 5% del total) y aquellos en los que no (el 95% restante). Tanto fue así que la red en todos los casos daba como resultado que no había habido un accidente, incluso con este comportamiento se lograba un error muy bajo y una *accuracy* (una de las métricas empleadas) del 95% a pesar de que evidentemente esta forma de realizar las predicciones es altamente insatisfactoria.

Para tratar de paliar este efecto se adoptaron diversas medidas, siendo una de ellas la normalización de los datos. Esta aproximación no dio los resultados esperados y al final se decidió descartarla (a pesar de eso en el fuente *utils* se puede ver la función con la que se realizaba dicho proceso).

Finalmente, se optó por reducir el número de instancias del tipo *NO_ACCIDENTE,* para esto se tomaba una muestra aleatoria de las mismas. La proporción que se decidió dejar de entre todas las que se probaron fue la del 50% de cada una de las clases. A pesar de que esta es la solución por la que hemos optado al ser la que mejor respuesta nos ha dado nos surge una preocupación. Esta es que al entrenar la red de esta manera se esté generando algún bias, ya que de alguna forma le hemos enseñado a que cuando se enfrente a la situación de realizar predicciones en un entorno real esperará que los casos de accidentes y no accidentes estén de forma proporcionada, cuando nosotros sabemos que realmente no es así (95% frente a un 5%).

RED KERAS

La versión de Keras que realizamos es considerablemente más simple, ya que estamos usando una librería diseñada para tratar con redes neuronales. La parte del código que se encarga de crear la neurona ocupa menos de 10 líneas.

En estas líneas de código se encuentra la creación de capas y sus funciones de activación, en nuestro caso dos capas con la función sigmoide como activación. La segunda de las capas termina

en una sola neurona, esto lo hacemos así porque como resultado obtenemos un solo valor y gracias a usar la función sigmoide este está comprendido entre 0 y 1.
Para el entrenamiento intentamos usar parámetros similares a la red creada por nosotros desde cero. Para conseguir esto usamos como optimizer "adam" ya que tiene un learning rate de 0.001, como loss function "mean_squared_error" y como métricas tanto la "accuracy" como la "precision". Por último, también usamos el mismo número de "epochs=50" y "batch_size=1".

RED NUMPY

ARQUITECTURA DE NUESTRA RED

A la hora de elegir la arquitectura de nuestra red nos encontramos con diferentes objetivos, primero debía cumplir unas características necesarias para el tratamiento de datos, a la vez que buscábamos maximizar el rendimiento sin llegar a comprometer los tiempos de entrenamiento. Con todo esto en mente, optamos por los siguientes hiperparámetros para configurar nuestra red neuronal.

El número de nodos que hay por capa oculta será siempre igual al número de inputs introducidos al inicio de la misma. Todas los inputs, o en su defecto la salida de la anterior neurona, se conectan con todas las neuronas de la siguiente capa donde se realiza la sumatoria pesada y se aplica la función de activación (sigmoide). Aunque inicialmente optamos por una sola capa oculta, con el tiempo pudimos observar que al añadir una extra conseguíamos mejorar sustancialmente la respuesta de la red neuronal por lo que al final hemos terminado con 2.

En lo que respecta a la tasa de aprendizaje el valor con el que trabajamos es de 0.001, hemos probado varias opciones pero esta es una de las mejores para nuestro caso. En lo que respecta a las épocas decidimos dejarlas en 50. Inicialmente hacíamos tan solo 10 pero con el tiempo nos dimos cuenta de que, especialmente en la que no usábamos keras, no era suficiente para que la red se entrenase lo suficiente como para llegar a unos valores mínimamente aceptables.

En la propagación del error para la corrección de los pesos (que hacemos después de procesar una de las instancias) nos encontramos al ver el desarrollo matemático que usted nos proporcionó con una pregunta que nunca logramos resolver por la falta de oportunidades para coincidir. Es en el cálculo de la delta para una capa oculta genérica. No supimos exactamente

discernir si se refería con la delta de u que aparecía, a literalmente la obtenida al calcular la modificación de pesos para la última capa de la red o en cambio se trataba de la última capa a la que habíamos modificado los pesos (no necesariamente la capa de salida). Optamos finalmente por esta última opción, ya que creímos que a la hora de modificar los pesos de estas capas intermedias tenía una lógica mayor usar la delta de aquella capa más próxima a los pesos que deseamos alterar, al poder albergar esta una mayor información sobre qué valores debería tener su vecina en comparación con la capa de salida que puede estar separada de esta capa genérica por un número indeterminado de capas.

SHUFFLES

Con objetivo de no aumentar el bias que ya acarreamos desde el preprocesado de los datos, a lo largo del entrenamiento de la red neuronal se realizan varios barajeos de los mismos.

Algunos de ellos los realizamos al usar la función *sample()* de la librería *pandas*. Esta selecciona de forma aleatoria una cantidad previamente especificada (entre 0 y 1) de las instancias de un dataset, si esta cantidad es un 1 se mezclarán todas las filas del dataset en un nuevo orden. Este tipo de barajado lo hacemos fundamentalmente a la hora de hacer los sets de entrenamiento y validación o en el preprocesado de los datos.

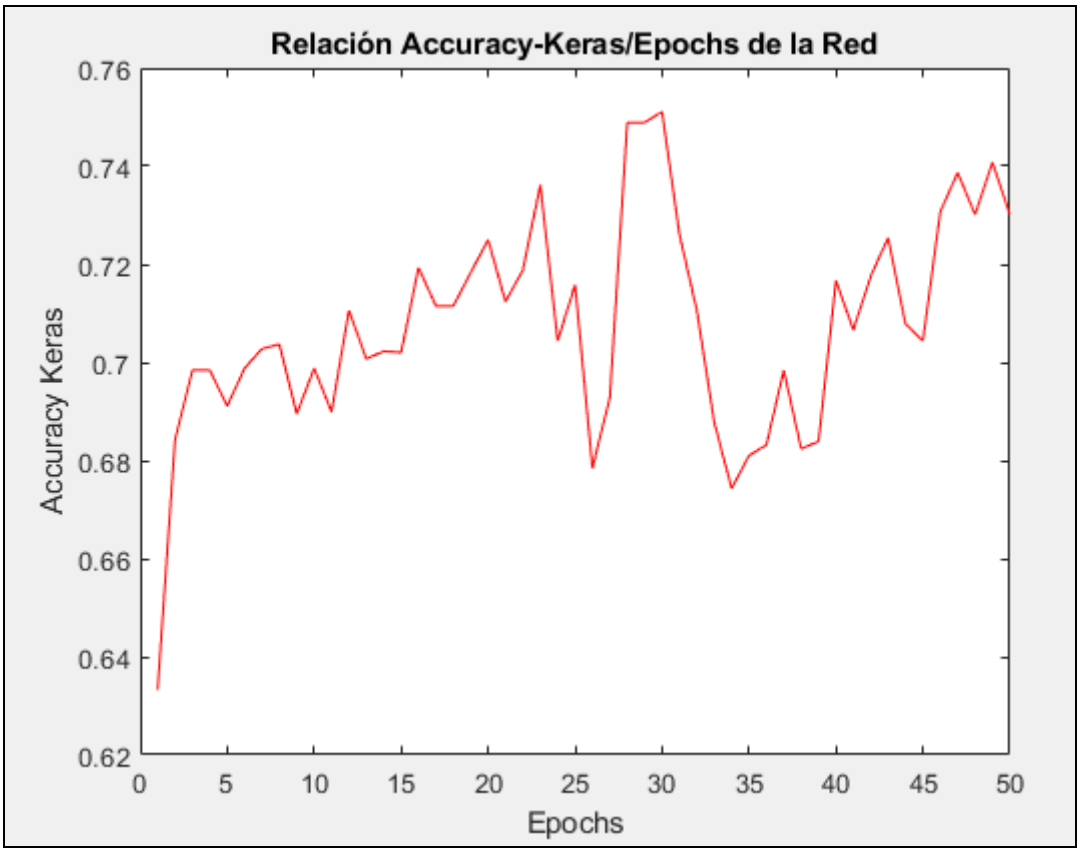
Además, ya dentro de la función de entrenamiento en cada época se produce un desorden de los elementos con los que estamos tratando de preparar a la red. Para esta situación empleamos la función *numpy.random.permutation()* que nos permite modificar en el mismo orden de forma aleatoria dos listas o arrays diferentes (esto nos es muy útil para poder mantener asociados los valores que se introducen con los que se espera que prediga la red).

SALIDA

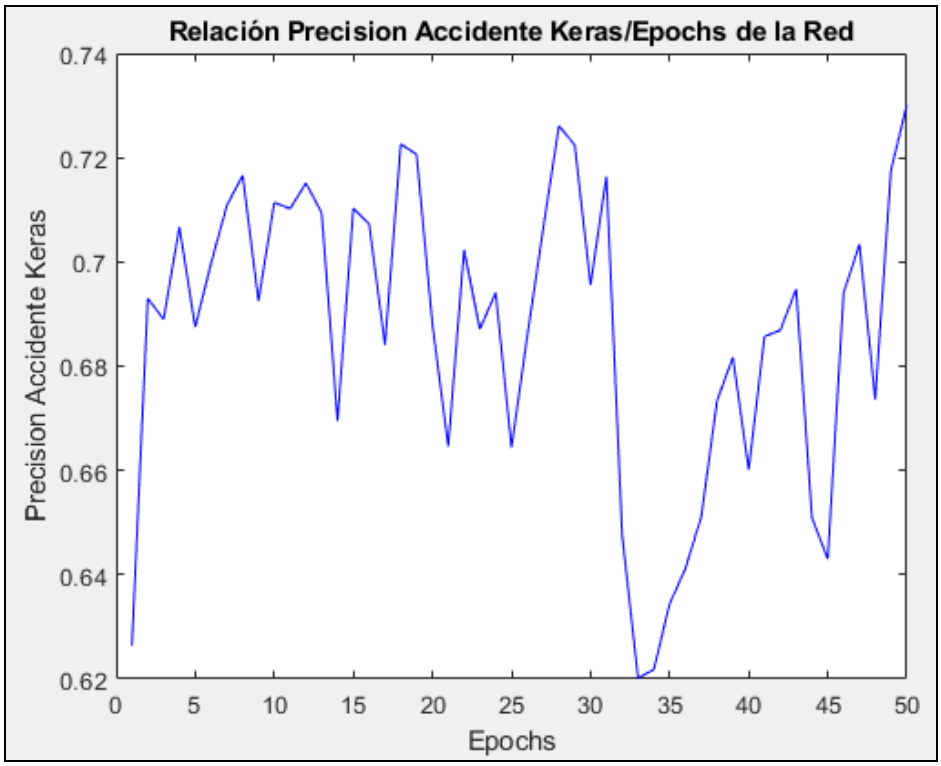
La salida de la red se hace con únicamente una neurona de salida, de esta forma conseguimos que cada ejemplo tenga una sola salida como respuesta. Además, al usar como función de activación la sigmoide, conseguimos que dicho valor esté comprendido entre 0 y 1. Siendo 1 una gran probabilidad de accidente y 0 una baja probabilidad de accidente. Sabiendo esto dividimos los valores entre el número de colores del semáforo y les otorgamos un rango a cada uno, de esta manera las respuestas entre [0.0, 0.2) corresponden al verde, [0.2, 0.4) al verde claro, [0.4, 0.6) al amarillo, [0.6, 0.8) al naranja y [0.8, 1.0) al rojo.

GRÁFICAS Y COMPARATIVAS

KERAS

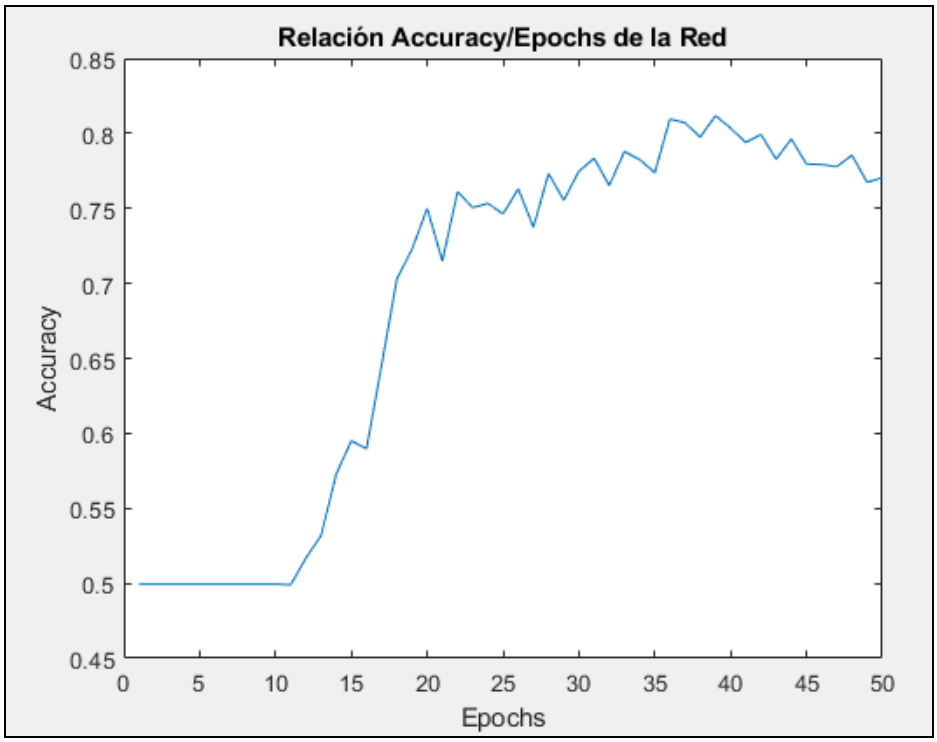


Progreso de la *accuracy* de la red en Keras a lo largo de las épocas.

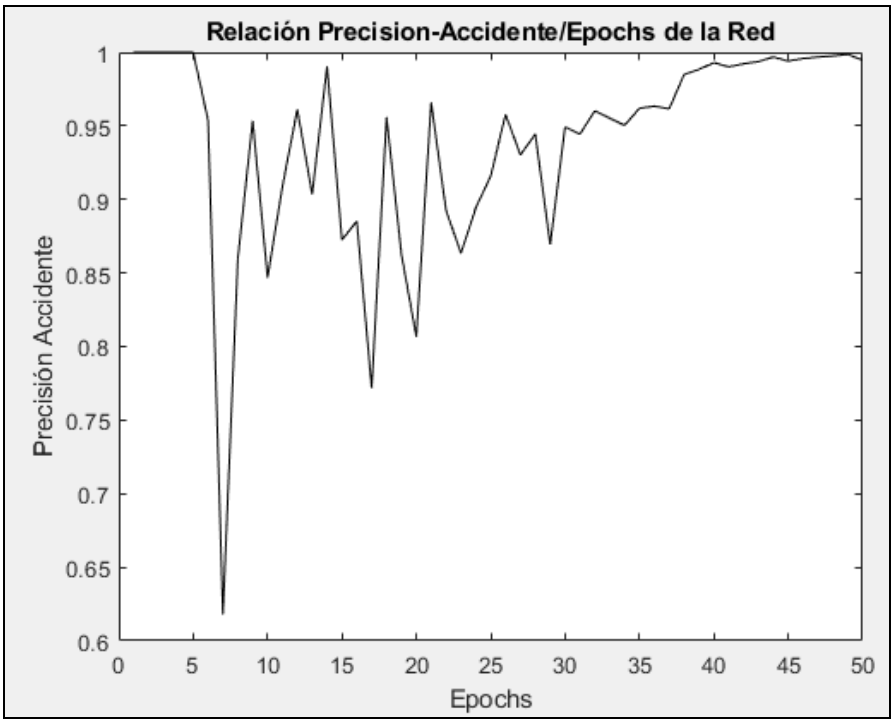


Progreso de la *precision* de la red en Keras a lo largo de las épocas.

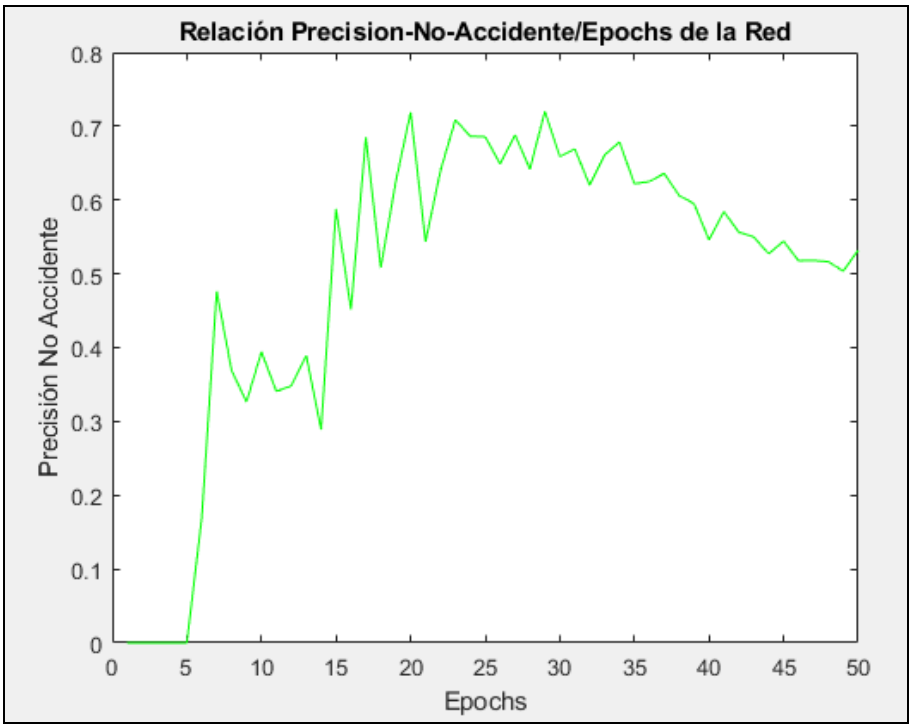
NUMPY



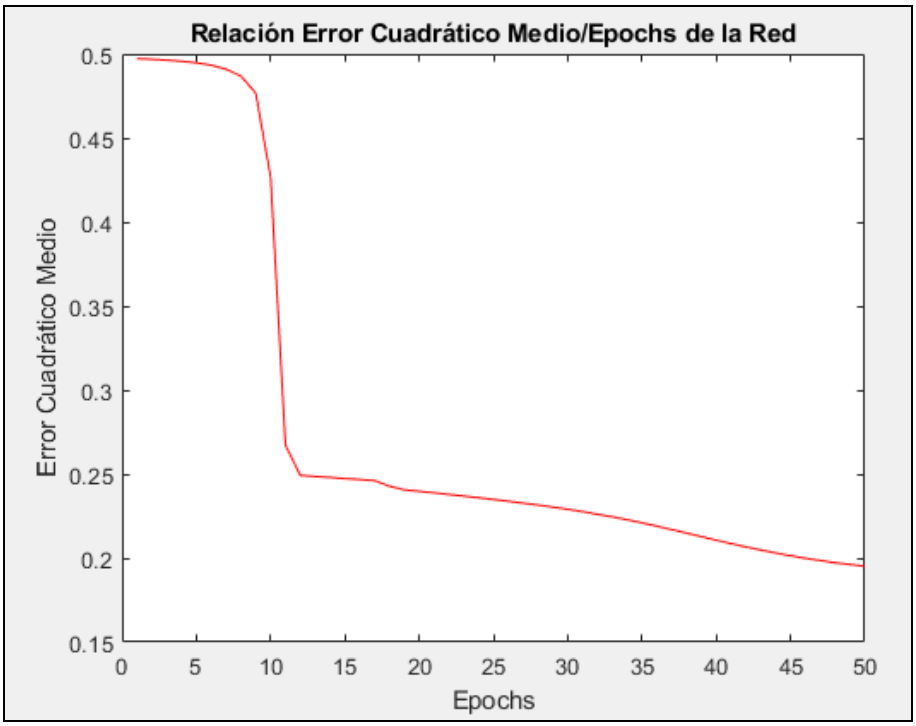
Progreso de la *accuracy* de la red en Numpy a lo largo de las épocas.



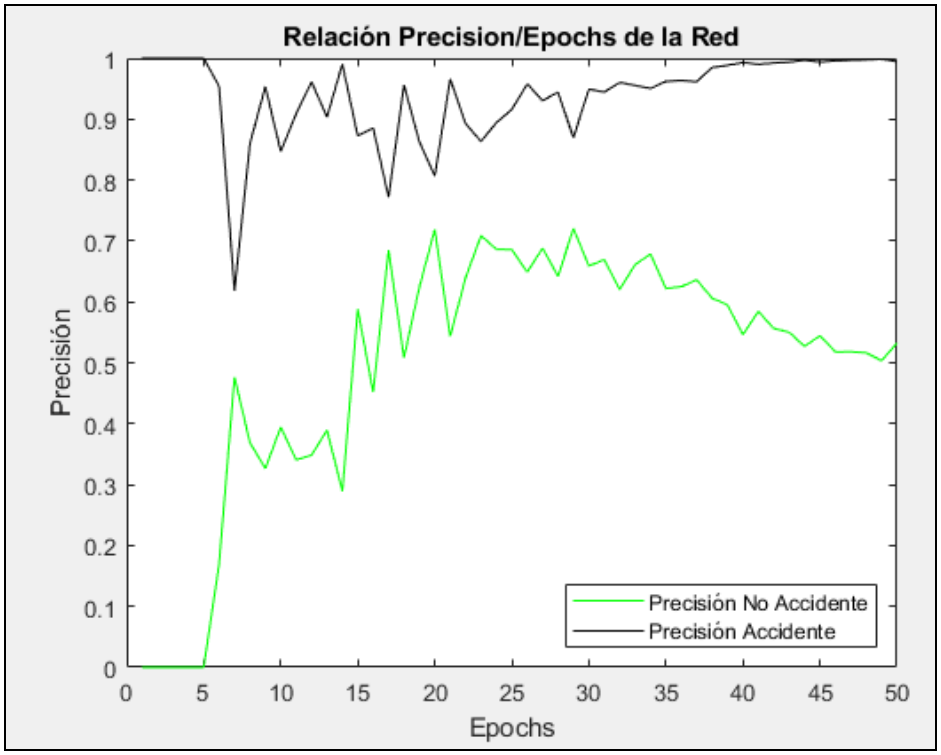
Progreso de la *precision* de accidentes de la red en Numpy a lo largo de las épocas.



Progreso de la *precision* de NO-accidentes de la red en Numpy a lo largo de las épocas.



Progreso del error cuadrático medio red en Numpy a lo largo de las épocas.

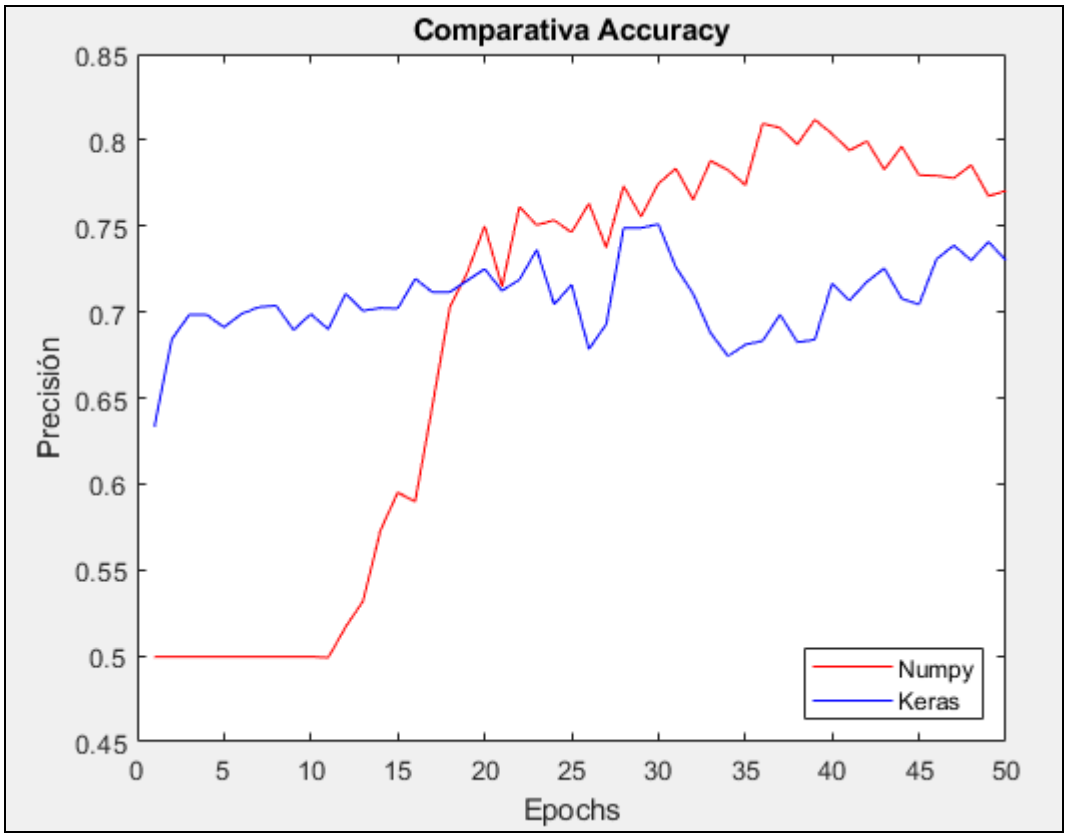


Comparativa de ambas *precision* de la red en Numpy a lo largo de las épocas.

COMPARATIVA

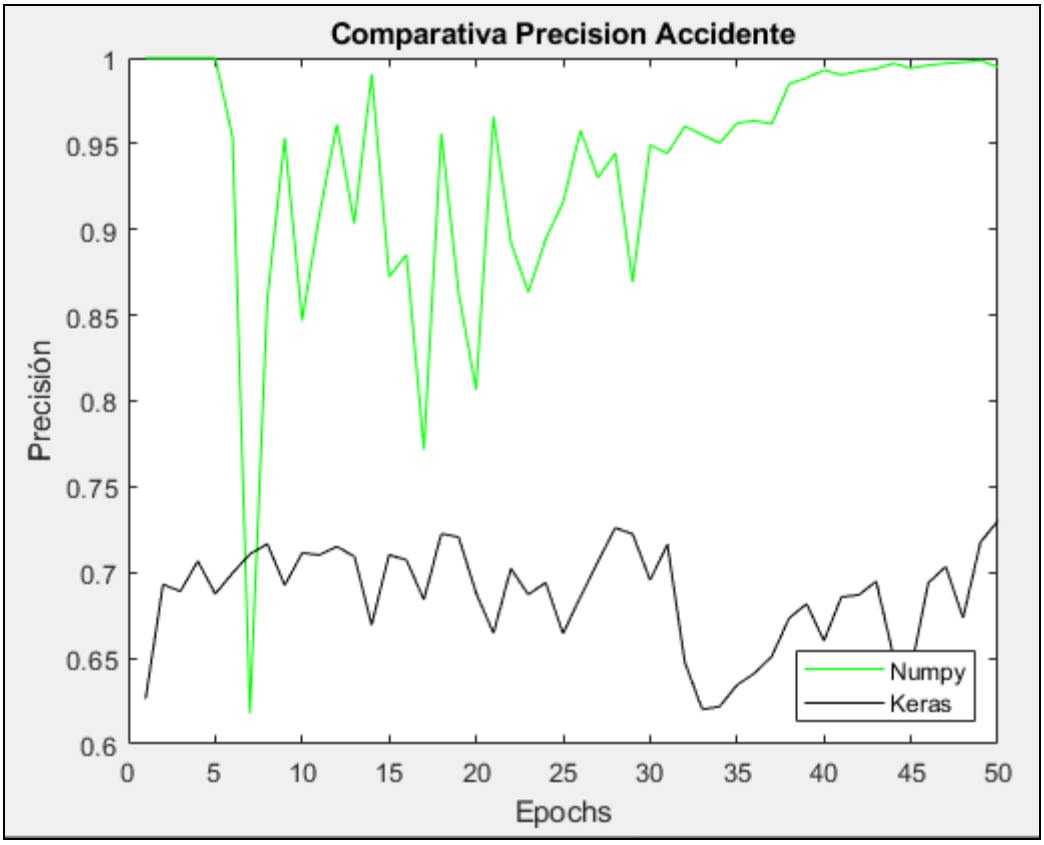
En la siguiente función podemos ver una comparativa entre las *accuracy* de la red en Keras como las de la red en Numpy, en ella podemos observar que, aunque comienza en un punto más bajo la de Numpy consigue alcanzar una *accuracy* superior al finalizar las épocas.

Esto no es ningún indicativo de superioridad de dicha red sobre la otra ya que los resultados en cada ejecución varían, debido a los sets utilizados y el orden de estos.



Comparativa de *accuracy* de la red en Numpy y de la red en Keras.

En cambio, en la función a continuación lo que nos encontramos es una comparativa entre las *precision* de ambas redes. En este caso es curioso resaltar que la precisión de accidentes de la red en Numpy empieza en un 100% y, aunque cae en picado, vuelve a acabar cercana al 100%. Esto lo que nos quiere decir es que siempre que la red detecta un positivo, es decir que determina que va a haber un accidente es muy probable que lo haya, en cambio no asegura que lo que no detecta como accidente no lo vaya a ser.



Comparativa de *precision* de la red en Numpy y de la red en Keras.