TRABAJO DE FIN DE GRADO

Doble grado en física y matemáticas

UNIVERSIDAD D SEVILLA

Aplicación de redes neuronales al ajuste de parámetros en modelos diferenciales

Autor:

Sergio Murillo García

Tutores:

Francisco Manuel Guillén González

Anna Doubova Krasotchenko

Junio 2023

Por todos las personas que han hecho que estos cinco años sean tan bonitos.

Resumen

El éxito del desarrollo de la inteligencia artificial durante los últimos 15 años ha revolucionado todas las áreas del conocimiento. Debido a la «maldición de la dimensión», que hace referencia al desafío que surge al tratar de resolver sistemas de ecuaciones diferenciales con un gran número de variables, su aplicación a la aproximación numérica de ecuaciones diferenciales juega un papel muy importante en problemas de alta dimensión.

En este trabajo se realizará una introducción a los conceptos básicos de las redes neuronales generales. Se estudiará el algoritmo de diferenciación automática (*backpropagation*), clave de su eficiencia, y se demostrará el Teorema de aproximación universal, base matemática que justifica la utilidad de las redes neuronales en la aproximación de funciones. Se utilizarán las redes neuronales denominadas como *Physical Informed Neural Networks* (PINN) para la resolución de ecuaciones y sistemas diferenciales y el ajuste de parámetros en ellos. Además, se implementarán utilizando Python y sus librerías en problemas concretos.

Palabras clave: redes neuronales, entrenamiento, hiperparámetros, derivación automática, ajuste de parámetros, sistema diferencial, python.

Abstract

The successful development of artificial intelligence over the last 15 years has revolutionised all areas of knowledge. Due to the «curse of dimensionality», which refers to the challenge that arises when trying to solve systems of differential equations with a large number of variables, its application to the numerical approximation of differential equations plays a very important role in high-dimensional problems.

In this work, an introduction to the basic concepts of general neural networks will be given. The automatic differentiation algorithm (*backpropagation*), the key to its efficiency, will be studied and the universal approximation theorem, the mathematical basis that justifies the usefulness of neural networks in the approximation of functions, will be demonstrated. Neural networks known as Physically Informed Neural Networks (PINN) will be used to solve differential equations and systems and to adjust parameters in them. Furthermore, they will be implemented using Python and its libraries in concrete problems.

Keywords: neural networks, training, hyperparameters, automatic derivation, parameter tuning, differential system, python.

Índice general

Capítulo 1

Introducción

El auge de la inteligencia artificial en los últimos tiempos es algo irrefutable. Con la creación de la plataforma «ChatGPT» a finales de 2022 (ver [1]), cualquier persona puede acceder a la potencia de la inteligencia artificial (IA) con un simple clic. Sin embargo, la IA no es algo que se haya desarrollado en cuestión de un año.

La expresión «inteligencia artificial» fue acuñada en 1956 en la Conferencia de Dartmouth por el informático estadounidense John McCarthy. Se entiende por IA un conjunto de capacidades cognoscitivas e intelectuales expresados por sistemas informáticos cuyo propósito es imitar la inteligencia humana para realizar tareas, y que pueden mejorar a medida que recopilan información. Anteriormente, los estudios publicados en 1936 por parte del inglés Alan Turing formalizaron el concepto de «algoritmo» y establecieron las bases teóricas de la computación, es por ello que se le considera «el padre» de la computación moderna (ver [2]).

El desarrollo de la inteligencia artificial ha ido de la mano de cuestiones filosóficas y éticas. En 1950, Alan Touring propone el conocido como «Test de Touring» cuyo objetivo era diferenciar si un interlocutor era una inteligencia artificial o un humano (ver [3]). El programa «Eliza» es considerado como el primer programa informático que pasa el test, fue desarrollado en 1966 por el Alemán Joseph Weizenbaum (ver [4]).

En la historia de la inteligencia artificial se distinguen 3 periodos principales: la «cibernética» en 1940-1960, «conexionismo» en 1980-1990, y la corriente actual, bajo el nombre de «aprendizaje profundo» (*Deep Learning*). Los algoritmos de inteligencia artificial surgen con la idea de imitar cómo el cerebro es capaz de aprender y procesar la información, es por ello que uno de los nombres que recibe la IA es el de «Redes neuronales artificiales» (*Artificial Neuronal Networks*).

Desde el desarrollo por parte de Frank Rosenblatt de la primera red neuronal en 1959, conocida como «perceptrón», la complejidad de las mismas ha ido en un aumento sin límites. Desde el «perceptrón», que consta de una sola neurona intermedia, hasta llegar a tener más de un millón de neuronas (para hacerse una idea del orden de magnitud, el ser humano tiene del orden de 10^{11} neuronas). Además, se estima que el número de neuronas intermedias se duplica cada dos años y medio (ver [5]).

En el ámbito de la resolución numérica de las ecuaciones diferenciales, los métodos clásicos más importantes son los métodos de diferencias finitas y elementos finitos. Estos métodos discretizan las ecuaciones para aproximar la solución, cuando el número de variables aumenta, el número de puntos de la cuadrícula de discretización aumenta de forma exponencial. Esto se conoce como la «maldición de la dimensión» (*Curse of dimensionality*) y hace que la resolución de ecuaciones diferenciales en altas dimensiones sea ineficiente.

La aplicación de la inteligencia artificial a la aproximación numérica de ecuaciones diferenciales no consiste, por tanto, en tratar de encontrar métodos que superen los métodos clásicos en dimensiones bajas, es decir, con pocas variables, sino tratar de resolver ecuaciones diferenciales de alta dimensión, donde los métodos clásicos presentan dificultades.

En el Capítulo 2 se plantean las bases del funcionamiento de las redes neuronales artificiales y sus principales elementos: entrenamiento, hiperparámetros... En el Capítulo 3 se introduce el proceso de derivación automática utilizado en las redes neuronales, que se denomina algoritmo de *backpropagation*. El Capítulo 4 está dedicado al Teorema de aproximación universal, que sirve como fundamento matemático para asegurar que una red neuronal puede aproximar funciones bajo determinadas condiciones.

Una vez realizado el desarrollo teórico, en los siguientes capítulos se implementará en Python algoritmos de redes neuronales aplicados a problemas concretos. En el Capítulo 5 se introducen las redes neuronales utilizadas para la resolución de ecuaciones y sistemas diferenciales (PINN), y se realiza un estudio de cómo afectan los diferentes hiperparámetros en la aproximación obtenida. En el Capítulo 6 se utilizan las redes neuronales para el ajuste de parámetros en ecuaciones y sistemas diferenciales ordinarios. Las redes neuronales pueden adaptarse de forma sencilla a la resolución de ecuaciones en derivadas parciales, estos algoritmos se implementan en el Capítulo 7. Por último, el Capítulo 8 está dedicado a las herramientas informáticas y los códigos utilizados en el proyecto.

Capítulo 2

Redes neuronales

Una rama específica de Aprendizaje Automático (*Machine Learning*) son las redes neuronales artificiales (también llamadas, por simplicidad, redes neuronales). Son un intento de imitar el comportamiento de las neuronas del cerebro en el procesamiento de la información: estructurándose en capas que trabajan con la información que reciben de la capa anterior [6].

Una red neuronal es una colección estructurada de nodos, también llamadas neuronas, conectadas entre sí de una forma concreta, de modo que se obtiene una o varias salidas para unas entradas dadas. Cada una de estas conexiones permite transmitir información, simulando la sinapsis de las neuronas biológicas. Se puede pensar, por tanto, en una red neuronal como una función que, a partir de unas entradas, actúa de una determinada manera para proporcionar las correspondientes salidas.

Dependiendo del objetivo que se pretenda conseguir con la red neuronal, existen muchas arquitecturas diferentes para diseñarlas, algunas de las más conocidas se recogen en la Figura 2.1. Se hace notar que se habla de una red neuronal en el contexto del «aprendizaje profundo» (*Deep Learning*) cuando tiene muchas capas ocultas, que es lo que se observa en la arquitectura b) de dicha figura. El adjetivo «profundo» hace referencia a la noción de profundidad que surge al apilar todas las capas intermedias [12].

Nos centraremos aquí en las que se denominan «redes neuronales prealimentadas» (*feed-forward neural network*), que incluye tanto al perceptrón multicapa como a la red neuronal profunda de la Figura 2.1. Para entender mejor el funcionamiento de una red neuronal tomaremos el ejemplo concreto de una red muy simple, representada en la Figura 2.2.

En este caso sencillo, es fácil obtener la expresión de la función de la red. Se denominará al dato de entrada como t. Para obtener la función se deben seguir las flechas del diagrama, que representan las conexiones entre las neuronas, teniendo en cuenta que, para ir de una neurona a otra multiplicamos el dato de entrada por el peso (*weight*) y sumamos el sesgo (*bias*) correspondiente. Además, al pasar por una neurona de la capa intermedia se aplica una «función de activación» que denominaremos T (ver la Sección 2.2.1). De esta forma, la salida de las neuronas 1

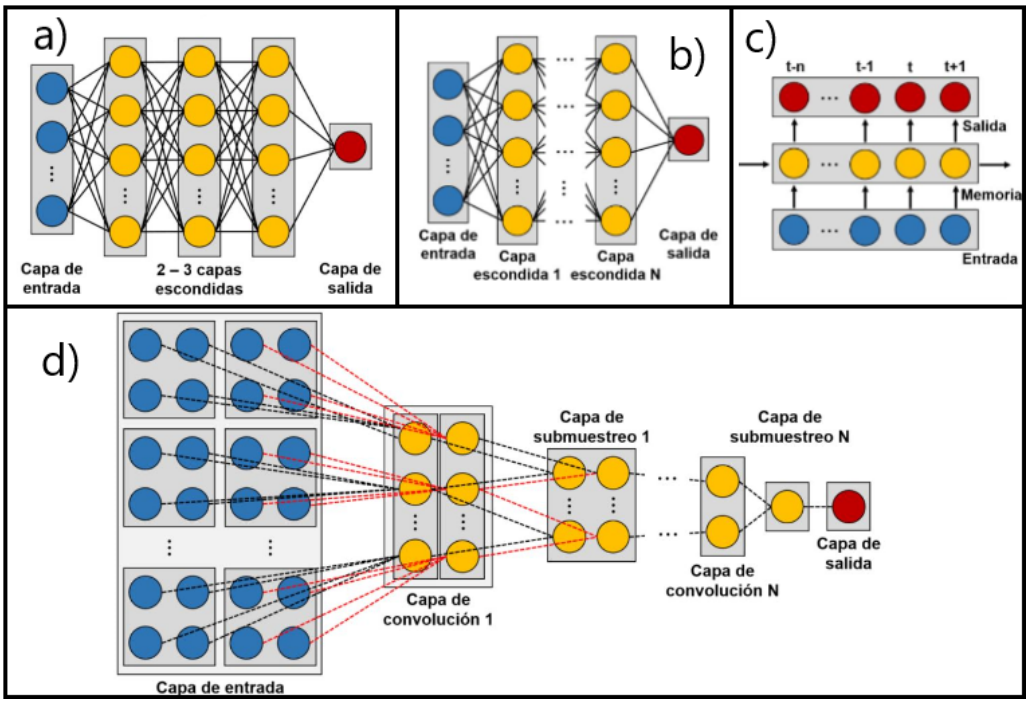


Figura 2.1: Arquitecturas típicas para redes neuronales: a) perceptrón multicapa, b) red neuronal profunda, c) red neuronal recurrente y d) red neuronal convulacional [11].

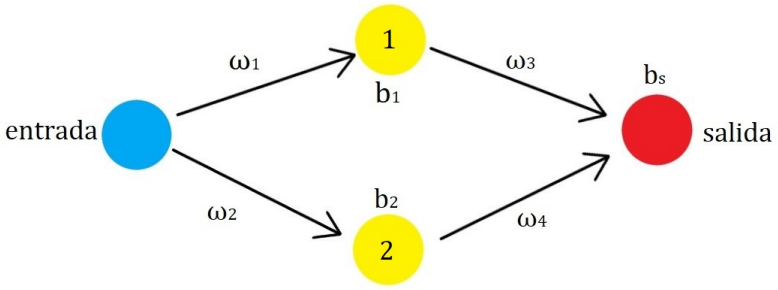


Figura 2.2: Ejemplo de red neuronal con una neurona de entrada, una capa oculta de dos neuronas y una neurona de salida, con sus parámetros: ω_i (pesos) y b_k (sesgos).

y 2 son, respectivamente las funciones

$$y_1(t) = \omega_1 t + b_1,$$
$$y_2(t) = \omega_2 t + b_2.$$

Aplicando ahora la función de activación T a ambas salidas intermedias y_1 e y_2 y teniendo en cuenta los pesos ω_3 y ω_4 y el sesgo b_s para llegar a la neurona de salida, se obtiene la función de la red $y(t)$, que sería la función de salida de la única neurona que tenemos en la capa de salida:

$$y(t) = \omega_3 T(y_1) + \omega_4 T(y_2) + b_s = \omega_3 T(\omega_1 t + b_1) + \omega_4 T(\omega_2 t + b_2) + b_s.$$

En general, para una red neuronal con más nodos y capas, la función de la red se obtiene de forma análoga. El objetivo de la red neuronal será variar los parámetros propios de la red, es decir, pesos $\omega_i \in \mathbb{R}$ y sesgos $b_k \in \mathbb{R}$, de forma que hagan que la minimización de una determinada «función de pérdida» sea óptima (ver Sección 2.2.3).

En rasgos generales, el ajuste de estos parámetros se hace durante lo que se denomina, en el lenguaje de redes neuronales, «entrenamiento» de la red, donde se emplean «datos de entrenamiento» y se trabaja con una «función de pérdida» que se minimizará en un proceso iterativo con un «optimizador», un algoritmo eficiente de optimización. Una vez finalizado el proceso de entrenamiento, la red neuronal está preparada para hacer las predicciones necesarias con otros datos de entrada distintos a los de entrenamiento. A continuación se van a precisar todos estos conceptos y detallar el procedimiento de optimización.

2.1. Entrenamiento

En esta sección haremos una descripción en líneas generales del proceso de entrenamiento de la red neuronal. Como ya hemos adelantado en la sección anterior, el entrenamiento es el proceso en el que se obtienen los valores adecuados de los parámetros internos de la red, también llamados parámetros «entrenables». Debemos ser conscientes de la gran complejidad que supone esta tarea, pues el variar uno de los parámetros en una neurona de la red, afectará de forma global al resto de neuronas conectadas con ella, además, se trabaja, en general, con una gran cantidad de parámetros.

El primer paso para poder obtener los valores óptimos de dichos parámetros es poder medir cómo de lejos está la predicción obtenida en el proceso de entrenamiento de los valores esperados. En este contexto aparece la función de pérdida o *loss function*, que involucra las predicciones y los valores esperados (datos de entrenamiento) y mide el error que se ha cometido.

Una vez se tiene este error, el objetivo es ver como afectaría la variación de cada uno de los parámetros en él, es decir, si este error aumentará o disminuirá al variar un parámetro. Dicha variación se puede determinar usando la derivada del error respecto de cada uno de los parámetros (pesos y sesgos).

Este proceso es, por tanto, un proceso de «ir y venir» por las neuronas. Primero se hacen todos los cálculos con los parámetros internos (la elección del primer conjunto de parámetros con el que se comienza el entrenamiento suele ser aleatorio, ver la Sección 2.2.2), es decir, se recorre hacia delante la red neuronal, es lo que se conoce como *forward propagation*. A continuación, se calcula el error con la función de pérdida y se transmite hacia atrás esta información para ver cómo depende de cada parámetro, obteniendo la derivada de la función de pérdida respecto de los pesos y sesgos, esto se denomina «propagación hacia atrás» (*backward propagation* o *backpropagation*), que veremos con más detalle en el próximo capítulo.

El siguiente paso en el proceso de entrenamiento es actualizar los parámetros. El método básico para hacerlo es el algoritmo de descenso del gradiente (*gradiente descent*). En este punto ya se han calculado las derivadas de la función de pérdida respecto de los parámetros de la red, luego se puede construir el vector gradiente. El gradiente da la dirección de máximo incremento, y su magnitud, cómo de rápido es esta variación, luego es razonable pensar que, como nos interesa la minimización de la función de pérdida, la variación de los parámetros debe hacerse de forma opuesta a dicho gradiente, en otras palabras, en la dirección de descenso del gradiente, de esta forma reduciremos, a priori, el error. Existen algunas variaciones de este método que tratan de aumentar el rendimiento, son lo que se conoce como «optimizadores» y que se verán más en detalle posteriormente.

Este proceso de actualización de los parámetros se realiza de forma iterativa, de manera que, en cada iteración, se está más cerca de los valores óptimos que minimizan el error.

En resumen, se hacen pasar por la red los datos de entrada con los que se hace el entrenamiento (*forward propagation*), se calcula el error con la función de pérdida, se calcula el gradiente con la propagación hacia atrás (*backpropagation*) y se varían los parámetros de acuerdo al descenso del gradiente (*gradient descent*). Este proceso iterativo se repite tantas veces como queramos para reducir el error, dichas iteraciones son lo que se denominan «épocas» o *epochs*.

2.2. Hiperparámetros

Hemos visto que la red tiene unos parámetros internos a optimizar: pesos y sesgos, pero estas no son las únicas variables que determinarán el funcionamiento de la red, el resto de ellas se denominan hiperparámetros, que se refieren a los parámetros externos que el programador debe seleccionar. La mayoría han sido introducidos brevemente, como la función de activación, función de pérdida, optimizador, número de neuronas por capa . . . En lo que sigue, veamos con más detalle cada uno de ellos.

2.2.1. Función de activación

Como hemos introducido anteriormente, la función de activación se aplica a las salidas de las neuronas de las capas intermedias, aunque también puede interesar aplicarla en la capa de salida. El objetivo de esta función es introducir no linealidad en la red neuronal, sin ella, la red sería una función lineal de la entrada.

Veremos más adelante que, a la hora de optimizar la red, será necesario calcular de forma automática las derivadas de esta función, es por ello que se buscan funciones cuyas derivadas se puedan escribir en términos de ellas mismas.

Veamos a continuación algunas de las funciones más usadas en la actualidad [12].

Sigmoide

Su expresión y la de su derivada son las que siguen y su representación gráfica aparece en la Figura 2.3:

$$T : \mathbb{R} \to \mathbb{R},$$

$$T(t) = \frac{1}{1 + e^{-t}},$$

$$T' = T(1 - T).$$

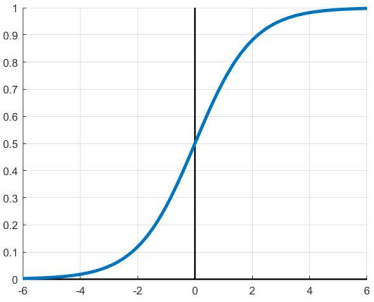


Figura 2.3: Función sigmoide.

Tanh

Se trata de la función tangente hiperbólica definida como sigue y representada en la Figura 2.4:

$$T : \mathbb{R} \to \mathbb{R},$$

$$T(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}},$$

$$T' = 1 - T^2.$$

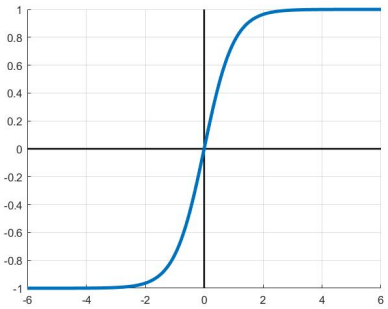


Figura 2.4: Función tanh.

Softmax

Esta función generaliza la función sigmoide a k dimensiones. Si se trabaja con un vector z de k elementos, la función Softmax "comprime" estos valores al conjunto $[0,1]^k$, dejando el valor máximo en la misma posición que la entrada. La expresión de esta función es la siguiente:

$$T : \mathbb{R}^k \to [0,1]^k,$$

$$T(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}} \quad \text{para} \quad j = 1, ..., k.$$

Para su derivada se tiene

$$\frac{\partial T_j}{\partial z_i} = \begin{cases} -T_i T_j & \text{si} \quad i \neq j \\ (1 - T_i) T_i & \text{si} \quad i = j \end{cases} \text{ para } \quad i, j = 1, ..., k.$$

Se utiliza eventualmente en la capa final en redes clasificadoras (redes neuronales cuyo objetivo es clasificar la entrada en 2 o más clases), ya que se puede entender como la probabilidad sobre clases de salida excluyentes, por ejemplo, cuando se trabaja con el reconocimiento de dígitos o una red neuronal encargadas de diferenciar entre perros y gatos. Se puede ver un ejemplo concreto en la Figura 2.5.

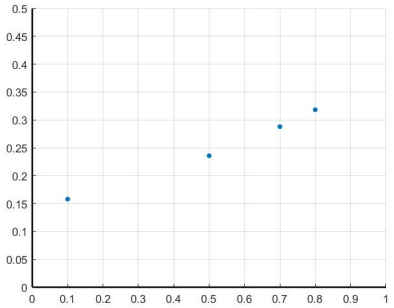


Figura 2.5: Función Softmax para $z = [0.1, 0.5, 0.8, 0.7]$.

ReLU

La función *Rectified Linear Unit* (ReLU) anula los valores negativos y deja sin cambios los positivos. Su expresión y la de sus derivadas son las siguientes:

$$T : \mathbb{R} \to \mathbb{R},$$

$$T(t) = \begin{cases} 0 & \text{si} \quad t < 0, \\ t & \text{si} \quad t \geq 0, \end{cases}$$

$$T'(t) = \begin{cases} 0 & \text{si} \quad t < 0, \\ 1 & \text{si} \quad t \geq 0. \end{cases}$$

Se tiene su representación en la Figura 2.6.

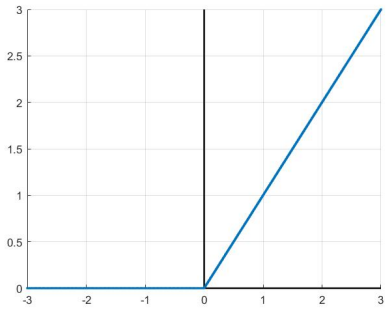


Figura 2.6: Función ReLU.

La utilización de una función de activación u otra depende del problema concreto. En redes neuronales *feed forward* (aquellas en las que las conexiones de las neuronas no forman ciclos) se utilizan, en general, la sigmoide, tanh o ReLU.

2.2.2. Inicialización de parámetros

La inicialización de los pesos y sesgos es importante en el aprendizaje de la red. En general, se realiza de forma aleatoria, ya que en caso de que dos neuronas tuviesen iguales parámetros, ambas se optimizarían siempre de igual forma (tendrían el mismo gradiente asociado) y, por tanto, se estaría perdiendo poder de mejora. Deben ser números no excesivamente grandes, ni excesivamente pequeños, ya que aparecería lo que se conoce como desvanecimiento del gradiente (*vanishing gradient*) y gradiente explosivo (*exploding gradients*), respectivamente (ver [6]).

La inicialización más extendida es la denominada *Xavier normal* o *Glorot normal*. Consideremos las neuronas de una determinada capa, y llamemos f_{in} y f_{out} el número de entradas y salidas de dicha capa. Se toma entonces, para los valores iniciales de los pesos y bias de las neuronas de esta capa, números aleatorios siguiendo una distribución normal, acotados entre $\pm\sqrt{\frac{6}{f_{in}+f_{out}}}$ (ver [13]).

2.2.3. Función de pérdida

La función de pérdida es la que se utiliza en el proceso de optimización y entrenamiento una vez se ha realizado una etapa del entrenamiento de la red. En la literatura también se denomina función de coste, objetivo o de error.

La determinación de esta función depende mucho del problema concreto que se esté tratando y de la función de activación que se utilice. A continuación pasamos a describir algunas de las más comunes.

Error cuadrático medio

El *mean squared error* o «MSE» es el más usado en los modelos de regresión. Su expresión es

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y(x_i) - y_i)^2,$$

donde los (x_i, y_i) son los n datos utilizados en el entrenamiento (y_i es la salida asociada a la entrada x_i) e $y(x)$ representa la función de la red neuronal con una entrada y una salida. En el caso de que la red tuviese j entradas, x_i pasaría a ser un vector con j componentes e igual para y_i en una red neuronal con más de una neurona en la capa de salida.

Error absoluto medio

También puede usarse el *mean absolut error* o «MAE» cuya expresión es

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y(x_i) - y_i|,$$

donde los (x_i, y_i) son los n datos utilizados en el entrenamiento e $y(x)$ representa la función de la red neuronal con una entrada y una salida. De nuevo, en el caso de que la red tuviese j entradas, x_i pasaría a ser un vector con j componentes y análogamente para y_i en una red con más de una salida.

Existen otras muchas funciones de pérdida dependiendo del objetivo de la red, por ejemplo, *categorical cross-entropy* o *sparse categorical cross-entropy* usadas en el contexto de la clasificación de imágenes [12].

2.2.4. Datos de entrenamiento

Ya se ha mencionado que para el proceso de entrenamiento se necesitan datos de nuestro problema específico, estos son el carburante de la red. Por ejemplo, si queremos un sistema que distinga entre fotos de gatos y perros, necesitaremos una gran cantidad de imágenes de ambos animales para que el sistema «aprenda» a diferenciarlos durante el entrenamiento.

El desafío del *Machine Learning* es predecir de forma acertada el comportamiento del sistema para datos que la red no ha visto anteriormente y no solo para los datos usados en el entrenamiento, para los que, tras dicho proceso, el error habrá sido minimizado. Esta habilidad de la red neuronal para predecir de forma correcta para datos «desconocidos» para la red se denomina *generalization* o «generalización».

Normalmente se dividen los datos disponibles en 2 grupos: «datos de entrenamiento» y «datos de prueba». Los datos de entrenamiento, que suponen entorno al 80 %, son los usados en el proceso de entrenamiento de la red para ajustar parámetros y elegir adecuadamente los hiperparámetros, mientras que los datos de prueba se utilizan para comprobar la capacidad de generalización de la red (ver [5]).

Se denominan *training error* y *test error* el error de los datos de entrenamiento y los datos de test, respectivamente. En general, se espera que el error del conjunto de test sea superior al de entrenamiento. Se tienen entonces dos objetivos principales:

1. Minimizar el *training error*.
2. Minimizar la diferencia entre el *training error* y el *test error*.

Estos dos objetivos se corresponden con los dos problemas centrales del *Machine Learning*, conocidos como «subajuste» y «sobreajuste» (*underfitting* y *overfitting*). El *underfitting* es no poder obtener un error de entrenamiento suficientemente bajo y el *overfitting* ocurre cuando la diferencia entre ambos errores es grande, es decir, cuando la red no es capaz de predecir adecuadamente valores no vistos.

Se define en este contexto la «capacidad» de la red como la habilidad de la misma para ajustar más o menos variedad de funciones. Por ejemplo, una red con una alta capacidad podría ajustar polinomios de grado 20, mientras que una red con poca capacidad ajustaría hasta polinomios lineales. Naturalmente, esto dependerá del número de parámetros entrenables de la red, aspecto que se tratará con mas detalle en el próximo apartado.

Por último, observamos que es interesante saber que, en ocasiones, se trabaja con cantidades enormes de datos, es por ello que el ordenador puede tener problemas para trabajar con todos ellos al mismo tiempo en la memoria. Se realiza entonces el entrenamiento tomando «lotes» (*batches*) del total de datos de entrenamiento. El tamaño óptimo depende de muchos factores, entre ellos, de la memoria del ordenador, como ya hemos comentado.

2.2.5. Número de neuronas y capas

El número de parámetros internos de la red neuronal se ve reflejado en el número de neuronas y capas. En el apartado anterior se han introducido los conceptos de *underfitting* y *overfitting*. Veamos un ejemplo para poner de manifiesto qué puede ocurrir con la red neuronal cuando se tienen estos fenómenos.

Supongamos que tenemos una red cuya salida son polinomios de grado 9 y otra red cuya salida es una función lineal. Supongamos también que la solución exacta del problema es una función cuadrática (el ajuste buscado) y que contamos con 6 datos de entrenamiento (valores que satisfacen la solución cuadrática). Es claro que la red lineal no podrá ajustar bien los datos. Mientras que la red de grado 9 sí que es capaz de representarlos de forma adecuada, pero además, es capaz de representar otras muchas funciones que también pasan de forma exacta por los puntos, ya que hay más parámetros que datos. Esto se puede observar mejor en la Figura 2.7 [5].

Como se observa en dicha figura, para la red lineal el error de entrenamiento es grande, pues la red no es capaz de ajustar los datos, se tiene el *underfitting*. En la imagen de la derecha de la Figura 2.7, se puede ver que la red de grado 9 ajusta de forma exacta los datos de entrenamiento, es decir, se tiene error de entrenamiento 0. Pero es claro que el error de test será muy grande, pues la curva no sigue una ecuación cuadrática. Luego, no tendrá buena capacidad de generalizar, se da el *overfitting*. En este caso la capacidad apropiada sería la del centro, donde no hay un número de parámetros excesivos, ni demasiado pocos.

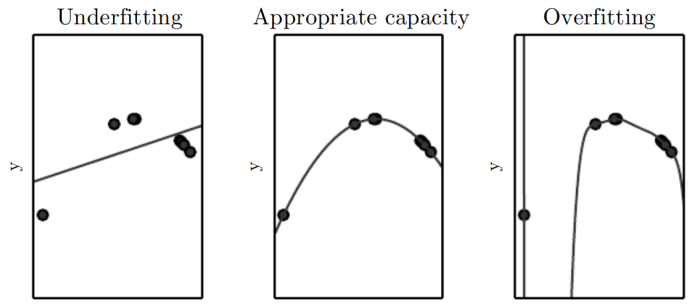


Figura 2.7: Ejemplo de *underfitting* (izquierda, red lineal); aproximación adecuada (centro, red de grado 2) y *overfitting* (derecha, red de grado 9). Los puntos son los datos de entrenamiento y la curva la salida de la red [5].

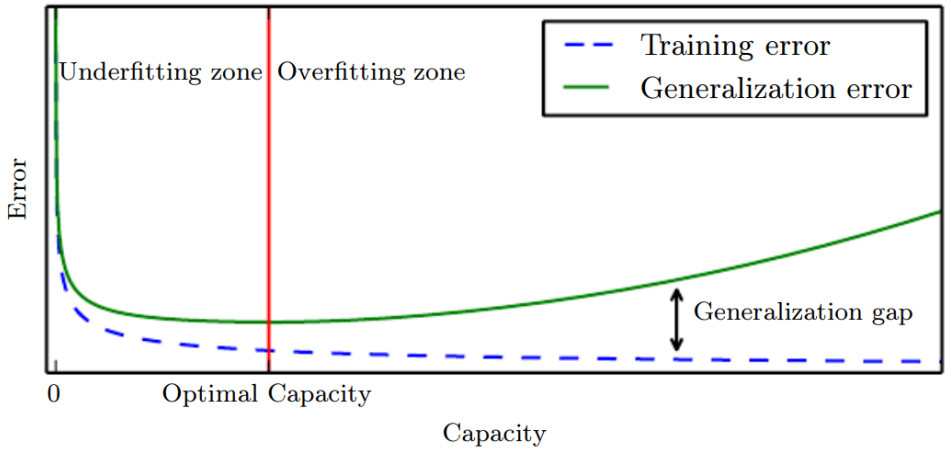


Figura 2.8: Relación típica entre los errores de entrenamiento y de test (*generalization error*) y la capacidad [5].

En conclusión, el número de parámetros no puede ser demasiado elevado, ni demasiado bajo. Se debe ajustar en relación a la cantidad de datos de los que se disponga. En la Figura 2.8 se puede ver la relación habitual existente en las redes para la capacidad óptima de un problema dado.

2.2.6. Número de épocas

Como ya se ha visto, el número de épocas o *epochs* es el número de veces que los datos de entrenamiento pasan por la red y se actualizan los parámetros. En otras palabras, es el número de veces que se resuelve el problema de optimización asociada a la función de pérdida. En general, el error de los datos de entrenamiento disminuye con las épocas, pero llegados a un punto, la mejora puede no ser rentable comparado con el tiempo necesario. Por otro lado, un número de épocas por debajo del óptimo puede limitar el potencial del modelo.

En general, el número de épocas adecuado se decide haciendo diferentes pruebas, en función del tiempo y de los recursos computacionales que se tenga.

2.2.7. Optimizador

Se ha introducido anteriormente que el optimizador será el encargado de minimizar la función de coste, encontrando los valores adecuados para los parámetros internos del modelo (pesos y sesgos). También, se ha mencionado que estos optimizadores se basan en el algoritmo del descenso del gradiente: cada parámetro interno se actualiza restándole la derivada de la función de error respecto de dicho parámetro multiplicado por un factor que se denomina «factor de entrenamiento» (*learning rate*) del que hablaremos en el apartado siguiente.

Existen variantes del algoritmo de descenso que tratan de mejorar el rendimiento, pues el descenso de gradiente básico, tiene una convergencia lenta, en general, y costosa numéricamente. Antes de ver algunos ejemplos de los optimizadores más usuales, veamos una de las variantes más importantes al método básico: el método del momento (*momentum*).

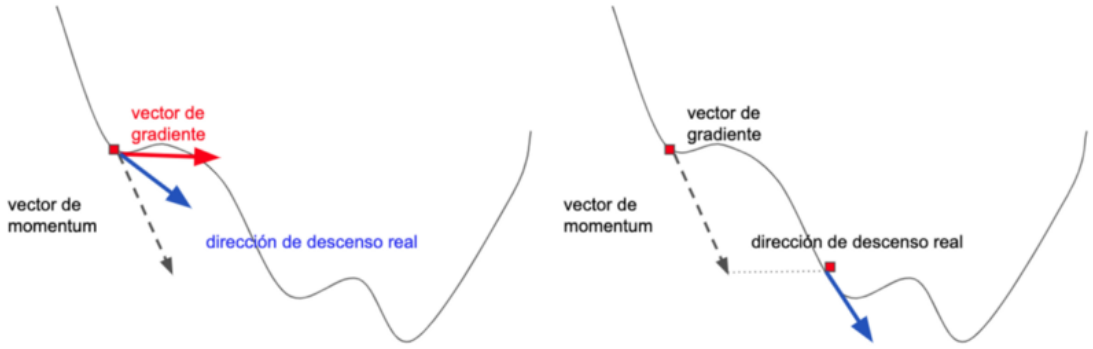


Figura 2.9: Esquema de la mejora del método del gradiente con el uso del *momentum* en el descenso al mínimo global, evitando el mínimo local [14].

El *momentum* o «cantidad de movimiento» hace referencia al momento lineal de la Física, cuanto más velocidad y más masa tenga un cuerpo, más «movimiento» tendrá, es decir, más difícil será frenarlo. En el contexto de la optimización de redes neuronales, el *momentum* consiste en considerar no solo el gradiente obtenido para una observación, sino un promedio de los gradientes anteriores, de forma que se tiene esta noción de ímpetu y de «cantidad de movimiento» [12]. De esta manera, para actualizar uno de los parámetros internos, se tiene en cuenta la «historia» del gradiente, el promedio de los gradientes calculados anteriormente, además del gradiente actual:

$$\omega = \omega_{anterior} - \alpha \frac{\partial C}{\partial \omega} + mom \frac{\overline{\partial C}}{\partial \omega}$$

Donde ω es uno de los pesos, α es el paso del método de descenso (*learning rate*), C hace referencia a la función de coste, $\frac{\overline{\partial C}}{\partial \omega}$ hace referencia al promedio de los gradientes anteriores y $mom \in [0, 1]$ es el coeficiente que mide la «importancia» que se le da al momento (si se toma como 0 sería el gradiente habitual).

Hasta ahora hemos hablado del descenso del gradiente pensando únicamente en un mínimo global, es decir, sin tener en cuenta posibles mínimos locales que «atasquen» la optimización. Los problemas reales son, en general, complejos y existen mínimos locales, puesto que la función de pérdida no es convexa en general. La introducción del *momentum* puede solventarlos, como puede observarse en la Figura 2.9.

Veamos a continuación las características principales de algunos de los optimizadores más usuales (ver [14, 15, 31]).

***Stochastic Gradient Descent* (SGD):**

Dada la gran cantidad de parámetros internos, calcular todas y cada una de las derivadas del error respecto a cada parámetro y evaluarlas para todas las observaciones es inviable. Una primera aproximación consiste en evaluar las derivadas únicamente para una observación en cada «lote» (*batch*), haciendo el gradiente del funcional objetivo reducido a dicho *batch*.

Este método se utiliza en ocasiones implementando también el método del *momentum*.

***Adaptive Gradient Algorithm* (AdaGrad):**

La innovación que introduce este optimizador es considerar que el factor de entrenamiento (*learning rate*), no es constanste para cada parámetro interno, se actualiza de la siguiente forma:

$$\overline{\alpha} = \frac{\alpha}{1 + (epoch - 1)\mu},$$

donde μ se denomina *learning rate decay* (ver Sección 2.2.8), α es el paso del método de descenso constante y *epoch* es la época. Además se multiplica el *learning rate* por un factor que escala según la magnitud del gradiente.

***AdaDelta*:**

Este método se basa en el AdaGrad, pero considerando únicamente los n gradientes anteriores, y no todos.

***Root Mean Square Propagation* (RMSProp):**

Este optimizador es similar a AdaDelta. La diferencia está en la forma de asignar el factor de entrenamiento a cada parámetro interno, de forma que se le da mayor prioridad a los últimos valores del gradiente añadidos al promedio que a los más antiguos.

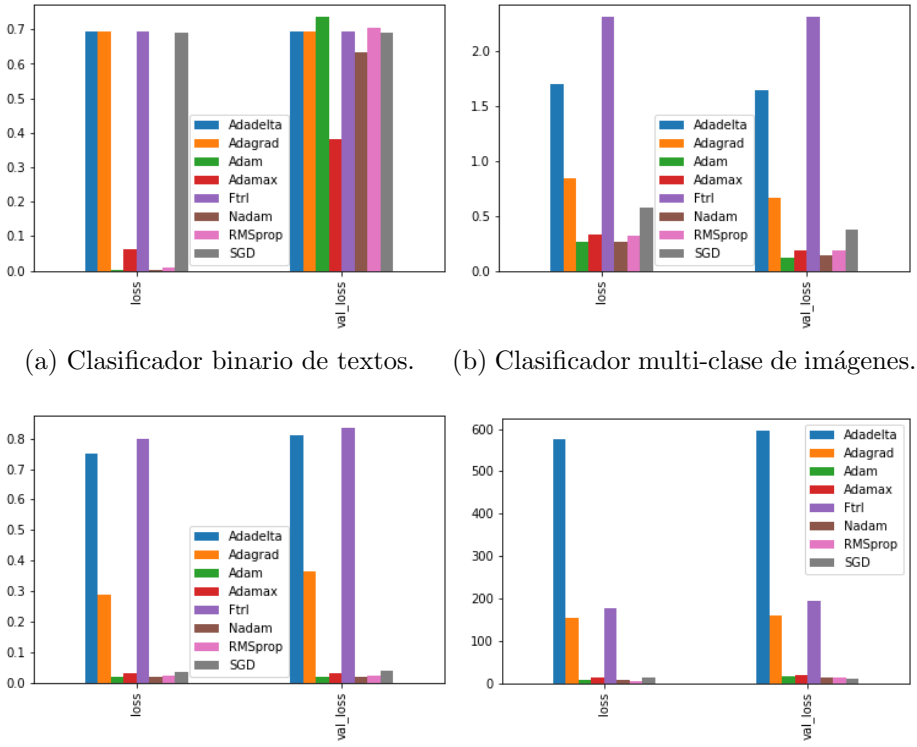
***Adaptive moment estimation* (Adam):**

Este conocido optimizador combina las bondades de AdaGrad y RMSProp y además hace uso del *momentum* explicado anteriormente.

***AdaMax* (Adamax):**

Este optimizador es una variante del método *Adam*, pero en vez de usar la norma 2 se usa la norma infinito en el cálculo del momento.

Como se ha visto, en general, los optimizadores se basan en optimizadores ya creados anteriormente introduciendo algunas variantes, para que su rendimiento sea mayor. En la práctica, el mejor optimizador depende del problema concreto con el que se esté tratando. Además, se hace notar que, en todos los casos, son optimizadores para calcular mínimos sin restricciones.



(a) Clasificador binario de textos.

(b) Clasificador multi-clase de imágenes.

(c) Pronóstico de series temporales (*Forecasting*).

(d) Regresión numérica simple.

Figura 2.10: Error de entrenamiento (*loss*) y error de test (*val_loss*) con varios optimizadores para 4 tipos de problemas clásicos en redes neuronales [14].

En la Figura 2.10 se adjuntan los resultados de los errores (de entrenamiento y de test) obtenidos con diferentes optimizadores (los disponibles en la libreria «Keras» de Python) aplicados a diferentes problemas. Se hace notar que en todos ellos se usa un *learning rate* de 0.001 y 50 épocas.

Se puede observar como, dependiendo del experimento concreto, el optimizador más adecuado varía. Luego, la conclusión general es que la decisión de qué optimizador elegir depende del objetivo de la red. Como un primer intento, se puede utilizar el optimizador Adam, pues se observa que tiene una actuación adecuada en los 4 problemas de la Figura 2.10.

2.2.8. Factor de entrenamiento (*learning rate*)

Según lo visto anteriormente, el *learning rate*, «factor de entrenamiento» o «paso» es el factor por el que se multiplica la magnitud del gradiente para actualizar los parámetros internos del modelo.

El valor adecuado del *learning rate* depende mucho del problema en cuestión. Si los pasos son demasiado pequeños, el entrenamiento puede ser demasiado lento, además de aumentar la probabilidad de «atascarse» en un mínimo local. Por el contrario, si el «factor de entrenamiento» es demasiado grande, al comienzo puede ser positivo, pues la red aprenderá rápidamente, pero en ubicaciones cercanas al mínimo, puede ser que las actualizaciones lleven a los parámetros a ubicaciones aleatorias sin llegar al mínimo global, pudiendo incluso divergir (ver [12]). En la Figura 2.11 se observan los comportamientos descritos.

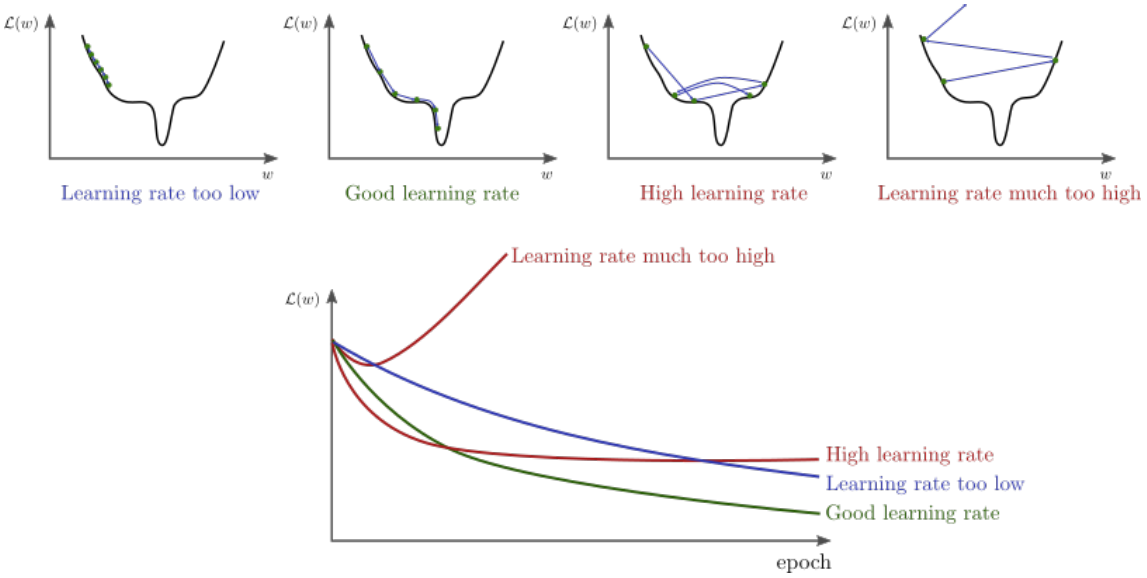


Figura 2.11: Esquema de la importancia del *learning rate* en el proceso de entrenamiento. $\mathcal{L}(\omega)$ hace referencia a la función de pérdida y ω a los parámetros internos. En la parte superior, distintos tipos de descensos según el tamaño del paso. En la gráfica central se observa el comportamiento de convergencia de $\mathcal{L}(\omega)$ en función de las épocas, para cada uno de los 4 tamaños de las gráficas superiores [16].

Se puede ver en la Figura 2.11 (gráfica central inferior) como, un factor de entrenamiento alto, puede ser útil al comienzo, ya que permite acercarse rápido a entornos del mínimo, una vez allí, es mejor un factor más pequeño, pues nos permite alcanzar la convergencia a valores óptimos. Por ello, un buen método es utilizar un factor de entrenamiento variable, de forma que disminuya a medida que aumentan las épocas. Esto es lo que se conoce como «decaimiento del factor de entrenamiento» (*learning rate decay*).

Existen muchos métodos para implementarlo, algunos de los más comunes son los siguientes (ver [17]).

Método básico:

El factor de entrenamiento α varía según la época de la siguiente manera:

$$\alpha(epoch) = \frac{\alpha_0}{1 + decayRate \cdot epoch},$$

donde $\alpha_0 \in \mathbb{R}$ es el factor inicial y *decayRate* $\in \mathbb{R}$ es el *learning rate decay* (suele tomarse entre 0 y 1). De esta forma el factor disminuye a medida que se aumentan las épocas. Por ejemplo, para $\alpha_0 = 0.2$ y tomando $decayRate = 0.2$, se tiene un factor de entrenamiento de 0.001 en la época 1000 y de 0.0003 en la época 3000.

Decaimiento exponencial:

En este caso el decaimiento es exponencial dependiendo de las épocas siguiendo la expresión:

$$\alpha(epoch) = decayRate^{epoch} \cdot \alpha_0.$$

Decaimiento manual:

En este caso se decide de antemano el decaimiento según las épocas. Un ejemplo podría ser:

$$\alpha(epoch) = \begin{cases} 0.01 & \text{si} \quad epoch \leq 1000, \\ 0.001 & \text{si} \quad 1000 < epoch \leq 3000, \\ 0.0005 & \text{si} \quad epoch > 3000, \end{cases}$$

pudiéndose ajustar según las épocas que vayan a realizarse y la velocidad de convergencia del problema concreto.

Capítulo 3

Derivación automática: *Backpropagation*

En este capítulo se trata la «propagación hacia atrás» (*backpropagation*), que es el algoritmo encargado de obtener el gradiente de la función de coste respecto de los parámetros entrenables de la red. Este algoritmo es muy importante, pues su efectividad permite realizar el descenso del gradiente durante el proceso de entrenamiento en un tiempo coherente.

No se debe confundir el algoritmo de optimización utilizado en el proceso de entrenamiento con el algoritmo de derivación automática. *Backpropagation* hace referencia únicamente al proceso de obtención del gradiente de una función en el contexto de las redes neuronales, generalmente esta función suele ser la función de coste, pero también puede aplicarse a otras funciones que se necesiten, eventualmente, en el proceso de entrenamiento de la red [5].

El algoritmo de *backpropagation* hace uso de la «regla de la cadena» (*chain rule*). Para dos funciones diferenciables $f, g : \mathbb{R} \to \mathbb{R}$ la regla de la cadena establece lo siguiente: la derivada de la composición z de f y g es $z'(x) = f'(g(x))g'(x)$. También se puede escribir en la notación de Leibnitz, poniendo:

$$y = g(x), \qquad z = f(y),$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

Esta regla puede generalizarse para funciones de varias variables. Consideremos ahora $f : \mathbb{R}^n \to \mathbb{R}$ y $g : \mathbb{R}^m \to \mathbb{R}^n$ diferenciables, se tiene entonces, para $x \in \mathbb{R}^m$ e $y \in \mathbb{R}^n$:

$$\begin{aligned} &y = g(x), \qquad z = f(y), \\ &\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}, \qquad i = 1, ..., m. \end{aligned} \tag{3.1}$$

Veamos entonces cómo se aplica el algoritmo usando un ejemplo concreto de una

red sencilla. Consideremos una red de 4 capas, 2 de ellas ocultas (L2 y L3), con 2 neuronas en la capa de entrada (L1) y 2 en la de salida (L4), tal y como vemos en la Figura 3.1. Sea L el número total de capas, $\omega_{jk}^{(\ell)}$ el peso correspondiente a la conexión que va de la neurona k de la capa $\ell-1$ a la neurona j de la capa ℓ, $b_j^{(\ell)}$ el sesgo de la neurona j de la capa ℓ, $z_j^{(\ell)}$ la salida de dicha neurona y $a_j^{(\ell)}$ el resultado de aplicar la función de activación $\sigma : \mathbb{R} \to \mathbb{R}$ a dicha salida. Consideremos además que C_x hace referencia a la función de pérdida de la red (el coste) con las entradas $x = (x_1, x_2)$, que se tomará como el error cuadrático medio habitual $C_x = \|a - \hat{a}\|^2$, donde $\hat{a}$ es el valor esperado de la salida para las entradas correspondientes, según los datos de entrenamiento, y a el vector de salida formado por $(a_1^{(L)}, a_2^{(L)})$.

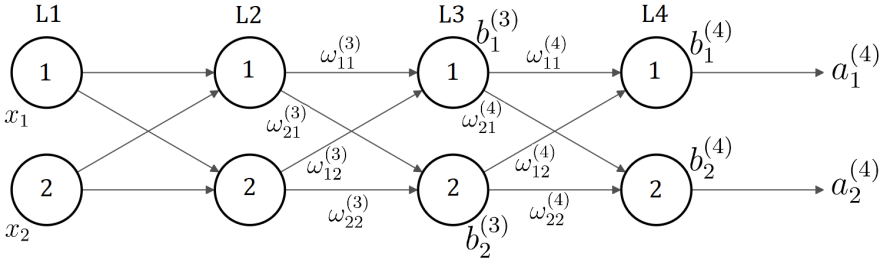


Figura 3.1: Esquema de una red neuronal sencilla para estudiar el algoritmo de *backpropagation*, con 4 capas y 2 neuronas por capa. De las 4 capas hay una de entrada, otra de salida y 2 capas ocultas. Se aplica la función de activación a las capas ocultas y a las salidas de la última capa.

En este contexto, supongamos que estamos interesados en obtener en primer lugar la derivada del coste respecto del parámetro ω_{11}^4. Podemos seguir el recorrido de relaciones entre los parámetros de la red neuronal de la Figura 3.2 (en sentido inverso, de derecha a izquierda). Lo que buscamos, aplicando la regla de la cadena expuesta anteriormente, se puede escribir como sigue:

$$\frac{\partial C_x}{\partial \omega_{11}^{(4)}} = \frac{\partial C_x}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial \omega_{11}^{(4)}} = \frac{\partial (\|a - \hat{a}\|^2)}{\partial a_1^{(4)}} \frac{\partial (\sigma(z_1^{(4)}))}{\partial z_1^{(4)}} \frac{\partial (\omega_{11}^{(4)} a_1^{(3)} + \omega_{12}^{(4)} a_2^{(3)} + b_1^{(4)})}{\partial \omega_{11}^{(4)}}$$

$$= \frac{\partial \left[(a_1^{(4)} - \hat{a}_1)^2 + (a_2^{(4)} - \hat{a}_2)^2 \right]}{\partial a_1^{(4)}} \sigma'(z_1^{(4)}) a_1^{(3)} = 2(\hat{a}_1 - a_1^{(4)}) \sigma'(z_1^{(4)}) a_1^{(3)}.$$

Suponiendo que se utiliza la función de activación sigmoide y recordando la relación con su derivada ($\sigma' = \sigma(1 - \sigma)$) se puede escribir:

$$\frac{\partial C_x}{\partial \omega_{11}^{(4)}} = 2(\hat{a}_1 - a_1^{(4)}) a_1^{(4)} (1 - a_1^{(4)}) a_1^{(3)}.$$

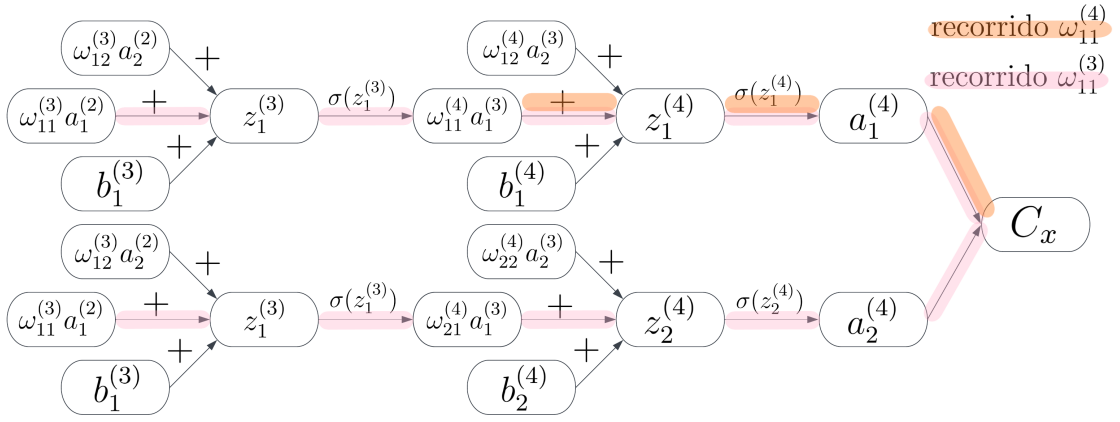


Figura 3.2: Esquema de las relaciones entre los parámetros internos correspondiente a la red de la Figura 3.1. Se pueden ver los recorridos que se siguen para obtener las derivadas de C_x respecto de $\omega_{11}^{(4)}$ y $\omega_{11}^{(3)}$ (se recorre en sentido inverso).

Para ver el algoritmo de forma general, se define el parámetro $\delta_1^{(4)}$ como sigue,

$$\delta_1^{(4)} = \frac{\partial C_x}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}},$$

$$\frac{\partial C_x}{\partial \omega_{11}^{(4)}} = \delta_1^{(4)} a_1^{(3)},$$

obteniendo así una expresión más compacta.

Obtengamos ahora, por ejemplo, la derivada respecto a $\omega_{11}^{(3)}$ y veamos que la podremos escribir en función de los δ_i de la capa 4. Siguiendo el esquema de la Figura 3.2, tenemos

$$\frac{\partial C_x}{\partial \omega_{11}^{(3)}} = \frac{\partial C_x}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial a_1^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial \omega_{11}^{(3)}} + \frac{\partial C_x}{\partial a_2^{(4)}} \frac{\partial a_2^{(4)}}{\partial z_2^{(4)}} \frac{\partial z_2^{(4)}}{\partial a_1^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial \omega_{11}^{(3)}}.$$

Utilizando los δ_i, calculando las derivadas y sacando factor común al último término, obtenemos:

$$\frac{\partial C_x}{\partial \omega_{11}^{(3)}} = (\delta_1^{(4)} \omega_{11}^{(4)} \sigma'(z_1^{(3)}) + \delta_2^{(4)} \omega_{21}^{(4)} \sigma'(z_1^{(3)})) a_1^{(3)}.$$

Generalizando esto para una neurona genérica, se puede escribir la derivada del error respecto de cualquier parámetro:

$$\frac{\partial C_x}{\partial \omega_{jk}^{(\ell)}} = \delta_j^{(\ell)} a_k^{(\ell-1)}, \qquad \frac{\partial C_x}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)}, \tag{3.2}$$

$$\delta_j^{(\ell)} = \sum_{k=1}^{N_\ell} \delta_k^{(\ell+1)} \omega_{kj}^{(\ell+1)} \sigma'(z_j^{(\ell)})), \qquad \ell = 1, \ldots, L-1 \tag{3.3}$$

donde N_ℓ es el número de neuronas de la capa ℓ.

Como hemos dicho anteriormente, se ha aplicado la función de activación a la capa de salida, pero esto no tiene por qué hacerse siempre. En el caso de se aplique, se definiría el $\delta_j^{(L)}$ como sigue:

$$\delta_j^{(L)} = \frac{\partial C_x}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}), \tag{3.4}$$

mientras que si no se aplica la función de activación sería:

$$\delta_j^{(L)} = \frac{\partial C_x}{\partial a_j^{(L)}}, \tag{3.5}$$

donde $\frac{\partial C_x}{\partial a_j^{(L)}}$ se obtiene directamente y depende de la función de coste elegida.

En resumen, el algoritmo sería el siguiente:

1. Se inicializan los parámetros de la red.
2. Para cada uno de los datos de entrada se realiza lo siguiente:
   a) Se realiza el *feed-forward*, es decir, se realizan los cálculos de todas las conexiones para obtener la salida (o salidas).
   b) Se calcula el $\delta_j^{(L)}$ para la capa de salida usando las ecuaciones (3.4) o (3.5) según se aplique o no la función de activación en la última capa.
   c) Se calculan los $\delta_j^{(l)}$ para todas las capas anteriores, usando la expresión (3.3).
   d) Se calculan las derivadas de la función de coste C_x respecto a cada uno de los parámetros internos de la red, según las ecuaciones (3.2).

Obteniéndose así el gradiente que se buscaba. Una vez en este punto, se podrían actualizar los parámetros según el optimizador elegido. Podemos entender ahora de donde procede el nombre de este algoritmo (*backpropagation*), pues se obtienen los δ de las neuronas de una capa determinada a partir de los de la capa posterior, es decir, se va realizando la propagación hacia atrás, partiendo de la capa de salida (ver [18, 19]).

Una vez hemos visto cómo funciona el algoritmo, podemos preguntarnos en qué sentido es un algoritmo rápido, pues, a priori, involucra muchos cálculos. Una manera alternativa en la que se puede pensar, es en un método de aproximación numérica de la derivada del coste C_x respecto de un parámetro de la red ω_j:

$$\frac{\partial C_x}{\partial \omega_j} \approx \frac{C_x(\omega_j + \varepsilon) - C_x(\omega_j)}{\varepsilon}$$

con $\varepsilon > 0$ número real pequeño. Es decir, aproximamos la derivada respecto de un parámetro usando la definición de derivada: se calcula la variación que sufre la función de coste al variar ligeramente un parámetro determinado. De forma análoga se procedería con los sesgos.

El problema de este procedimiento radica en que debemos hacerlo para cada uno de los parámetros entrenables de la red y, a su vez, para cada uno de los datos de entrada, lo cual supone un coste computacional enorme. La ventaja del algoritmo de *backpropagation* es que se obtiene el gradiente realizando únicamente un paso hacia delante y uno hacia atrás para cada dato de entrada (o datos en el caso de tener varias entradas) y se dispone de fórmulas iteradas que permiten aprovechar cálculos realizados en etapas anteriores del algoritmo.

Este procedimiento se implementó de forma general en 1986, cuando los científicos David Rumelhart, Geoffrey Hinton, y Ronald Williams publican un artículo titulado *Learning representations by back-propagating errors* [20], en el que exponen las bases de dicho algoritmo, permitiendo expandir los problemas que podían afrontar las redes neuronales.

Capítulo 4

Teorema de aproximación universal

El objetivo del trabajo es la utilización de las redes neuronales en la aproximación de funciones, es por ello que se necesita saber de antemano que es posible acercarse tanto como se quiera a una función dada a través de la función de una red neuronal. En este capítulo se dará una prueba de un teorema que lo asegura.

Se presentan algunas definiciones previas (ver [7]). Se denominará I_n al cubo unidad de dimensión n: $[0,1]^n$. El espacio de funciones continuas en I_n se denota por $C(I_n)$. Se usará la norma del máximo y se denotará por $\|\cdot\|$ dicha norma de una función en su dominio. Se denotará $M(I_n)$ al conjunto de medidas Borel finitas.

Definición 4.1. *Se dice que una función σ es discriminadora si para una medida $\mu \in M(I_n)$ se tiene que para todo $y \in \mathbb{R}^n$ y $\theta \in \mathbb{R}$, la igualdad*

$$\int_{I_n} \sigma(y^T x + \theta) d\mu(x) = 0$$

implica que $\mu = 0$.

Definición 4.2. *Se dice que una función $\sigma : \mathbb{R} \to \mathbb{R}$ es sigmoide si cumple*

$$\sigma(t) \to \begin{cases} 1 & si \quad t \to +\infty \\ 0 & si \quad t \to -\infty \end{cases}$$

A continuación, se presentan dos resultados que permitirán demostrar el Teorema de Aproximación Universal.

Teorema 4.1. *Dada una función $\sigma : \mathbb{R} \to \mathbb{R}$ continua y discriminadora, las siguientes funciones*

$$G(x) = \sum_{i=1}^{N} \alpha_i \sigma(y_i^T x + \theta_i) \tag{4.1}$$

son densas en $C(I_n)$ respecto de la norma del máximo (con $N > 1$ natural). En otras palabras, para cualquier $f \in C(I_n)$ y $\varepsilon > 0$ $\exists \alpha, y, \theta \in \mathbb{R}^N$ tales que $G(x) = G(x; \alpha, y, \theta)$ con la forma anterior satisface

$$|G(x) - f(x)| < \varepsilon \qquad \forall x \in I_n.$$

Demostración. Sea $S \subset C(I_n)$ el conjunto de funciones $G(x)$ de la forma (4.1). Se tiene que S es un subespacio vectorial de $C(I_n)$. Por definición, S es denso en $C(I_n)$ si $\overline{S} = C(I_n)$, donde $\overline{S}$ es la clausura de S.

Se hará la demostración por reducción al absurdo. Sea $R = \overline{S}$ tal que $R \neq C(I_n)$. Es claro que, como R es la clausura de S, R es un subespacio cerrado de $C(I_n)$. Por el Teorema de Hahn Banach (ver [8], Corolario 1.8) se deduce la existencia de un funcional lineal acotado $L \in L^1(\mathbb{R}^n)$ no nulo que cumple $L(S) = L(R) = 0$.

Por el Teorema de representación de Riesz se puede escribir el funcional anterior como

$$L(h) = \int_{I_n} h(x) d\mu(x) \qquad \forall h \in C(I_n)$$

para algún $\mu \in M(I_n)$. En particular, como $\sigma(y^T x + \theta) \in R$ $\forall y, \theta$, se tiene

$$\int_{I_n} \sigma(y^T x + \theta) d\mu(x) = 0 \qquad \forall y, \theta.$$

Como σ es discriminadora, de la igualdad anterior se obtiene $\mu = 0$, que implica L idénticamente nulo, luego se llega a una contradicción.

Se tiene entonces $\overline{S} = C(I_n)$ y S denso en $C(I_n)$.

■

A continuación, se presenta un resultado que muestra que cualquier función sigmoide continua es discriminadora. Cabe señalar que, en las aplicaciones de redes neuronales, las funciones de activación sigmoide continuas se toman monótonas crecientes, pero no se requiere la monotonía en los resultados presentados aquí.

Lema 4.1. *Toda función sigmoide σ acotada y medible es discriminadora. En particular, toda función continua sigmoide es discriminadora.*

Demostración. Sea σ función sigmoide. Se observa que $\forall x, y \in \mathbb{R}^N$ y $\theta, \varphi \in \mathbb{R}$ se tiene:

$$\sigma(\lambda(y^T x + \theta) + \varphi) \begin{cases} \to 1 & \text{si} \quad y^T x + \theta > 0 \quad \text{cuando} \quad \lambda \to +\infty, \\ \to 0 & \text{si} \quad y^T x + \theta < 0 \quad \text{cuando} \quad \lambda \to +\infty, \\ = \sigma(\varphi) & \text{si} \quad y^T x + \theta = 0 \quad \text{para todo} \quad \lambda. \end{cases}$$

Entonces, la función $\sigma_\lambda(x) = \sigma(\lambda(y^T x+\theta)+\varphi)$ converge puntualmente a la función γ cuando $\lambda \to +\infty$, donde $\gamma : \mathbb{R} \to \mathbb{R}$ se define como sigue:

$$\gamma(x) = \begin{cases} 1 & \text{si} \quad y^T x + \theta > 0, \\ 0 & \text{si} \quad y^T x + \theta > 0, \\ \sigma(\varphi) & \text{si} \quad y^T x + \theta = 0. \end{cases}$$

Se definen entonces el hiperplano y los semiespacios siguientes:

$$\begin{aligned} \Pi_{y,\theta} &= \left\{ x \in \mathbb{R}^N | y^T x + \theta = 0 \right\}, \\ H'_{y,\theta} &= \left\{ x \in \mathbb{R}^N | y^T x + \theta < 0 \right\}, \\ H_{y,\theta} &= \left\{ x \in \mathbb{R}^N | y^T x + \theta > 0 \right\}. \end{aligned}$$

Se tiene entonces que $\lim_{\lambda\to\infty} \sigma_\lambda(x) = \gamma(x)$ y además la función $|\sigma_\lambda(x)|$ está acotada, pues σ es sigmoide, luego por el Teorema de convergencia dominada de Lebesgue (ver [10], apartado 1.34) se tiene

$$0 \overset{\text{hip}}{=} \lim_{\lambda\to\infty} \int_{I_n} \sigma_\lambda(x) d\mu(x) = \int_{I_n} \gamma(x) d\mu(x)$$
$$= 1 \int_{H_{y,\theta}} d\mu(x) + 0 \int_{H'_{y,\theta}} d\mu(x) + \sigma(\varphi) \int_{\Pi_{y,\theta}} d\mu(x) = \mu(H_{y,\theta}) + \sigma(\varphi)\mu(\Pi_{y,\theta}),$$

para todo φ, θ, y. Se verá ahora que, como la medida de todos los semiplanos es nula, la medida μ deberá ser idénticamente nula.

Fijado y, para una función medible y acotada h se define el funcional lineal F como sigue:

$$F(h) = \int_{I_n} h(y^T x) d\mu(x).$$

Como μ es una medida finita, F es un funcional acotado en $L^\infty(\mathbb{R})$. Tomemos h la función indicatriz del intervalo $[\theta, \infty)$ (esto es $h(u) = 1$ si $u \geq \theta$ y $h(u) = 0$ si $u < \theta$). En ese caso, se tiene:

$$h(y^T x) = \begin{cases} 1 & \text{si} \quad y^T x - \theta \geq 0, \\ 0 & \text{si} \quad y^T x - \theta < 0. \end{cases}$$

En particular h es función sigmoide. Entonces se tiene

$$0 \overset{\text{hip}}{=} F(h) = \int_{I_n} h(y^T x) d\mu(x) = \int_{H_{y,-\theta}} h(y^T x) d\mu + \int_{H'_{y,-\theta}} h(y^T x) d\mu + \int_{\Pi_{y,-\theta}} h(y^T x) d\mu$$
$$= \mu(H_{y,\theta}) + \mu(\Pi_{y,\theta}).$$

Si se hace lo mismo esta vez con h la función indicatriz del intervalo abierto (θ, ∞), se tiene

$$0 \overset{\text{hip}}{=} F(h) = \int_{I_n} h(y^T x) d\mu(x) = \mu(H_{y,\theta}).$$

Luego, se deduce que $F(h) = 0$ para toda función indicatriz de cualquier intervalo. Por linealidad, se deduce que también es cero para cualquier combinación de ellas,

es decir, para cualquier función simple. Como las funciones simples son densas en $L^\infty(\mathbb{R})$ (ver [9], apartado (2.4.13)) pues $L^\infty \subset L^p$, se deduce que F es idénticamente nulo.

En particular, para las funciones acotadas y medibles $s(u) = \operatorname{sen}(m \cdot u)$ y $c(u) = \cos(m \cdot u)$, se puede escribir

$$F(s+ic) = F(s)+iF(c) = \int_{I_n} \cos(m^T x)+i\operatorname{sen}(m^T x)d\mu(x) = \int_{I_n} e^{im^T x}d\mu(x) = 0 \quad \forall m$$

Es decir, la transformada de Fourier de μ es 0 (ver [9], apartado 8.1.1). Luego μ debe ser 0. Luego σ es discriminadora. ■

Se puede entonces probar ahora el teorema principal de esta sección, que afirma que cualquier función continua f puede ser aproximada por una combinación lineal de funciones sigmoide.

Teorema 4.2 (de aproximación universal)**.** *Sea σ una función sigmoide continua. Entonces las siguientes funciones*

$$G(x) = \sum_{i=1}^{N} \alpha_i \sigma(y_i^T x + \theta_i)$$

son densas en $C(I_n)$. Es decir, dada una función $f \in C(I_n)$ y dado $\epsilon > 0$, existe $G(x)$ de la forma anterior tal que,

$$|G(x) - f(x)| < \epsilon \qquad x \in I_n.$$

Demostración. Por el Lema 4.1 se tiene que toda función sigmoide continua es discriminadora y por el Teorema 4.1 se tiene lo que se quiere. ■

Existen muchas generalizaciones de este teorema utilizando otro tipo de funciones de activación, ya que las funciones sigmoides no son las únicas que se implementan en las redes neuronales (ver [7]).

Capítulo 5

Redes Neuronales para la resolución de problemas diferenciales: PINN

Hasta ahora se han introducido los fundamentos de las redes neuronales y su funcionamiento de forma general, en este capítulo veremos cómo se pueden aplicar a la aproximación numérica de ecuaciones diferenciales. Una de las ventajas más importantes de utilizar este método es que es fácilmente adaptable a problemas diversos, lineales y no lineales, ecuaciones y sistemas diferenciales, ecuaciones y sistemas de ecuaciones en derivadas parciales, problemas inversos y problemas de control... Además, veremos ejemplos de aplicación a la resolución de problemas concretos.

5.1. PINN

Las redes neuronales que se usan para resolver este tipo de problemas se denominan «Redes neuronales informadas por la física» o PINN por sus siglas en inglés *Physics-Informed Neural Networks*. Para poder comprender mejor cómo se adaptan los hiperparámetros que vimos en el Capítulo 2 a nuestro objetivo, comenzaremos aplicando las redes neuronales a un ejemplo sencillo de una ecuación diferencial.

Tomaremos como ejemplo la resolución de una ecuación diferencial sencilla de la que se conoce la solución explícita, con una condición inicial dada. Más concretamente, consideramos el problema de Cauchy:

$$\begin{cases} u'(t) = \lambda t, & t \in [t_{min}, t_{max}], \\ u(t_{min}) = u_0, \end{cases} \tag{5.1}$$

donde $\lambda, u_0, t_{min}, t_{max} \in \mathbb{R}$ son parámetros conocidos. La solución analítica es

$$u(t) = \lambda \frac{(t^2 - t_{min}^2)}{2} + u_0, \qquad t \in [t_{min}, t_{max}]. \tag{5.2}$$

La pregunta ahora es: ¿cuáles son la función de pérdida, las entradas y salidas de la red neuronal y los datos de entrenamiento?

Como ya dijimos en el Capítulo 2, la red neuronal es, en realidad, una función que toma la entrada (o entradas), opera con un conjunto de parámetros internos y nos devuelve la salida (o salidas) correspondiente. Por tanto, el número de entradas y salidas de la red, es decir, el número de neuronas en la capa de entrada y salida, debe ser igual al número de variables independientes y dependientes de nuestro problema, respectivamente. En nuestro caso, 1 neurona de entrada (t) y 1 neurona de salida ($u(t)$).

Veamos ahora cuál debería ser la función de coste adecuada. Denominemos $\hat{u}(t)$ la salida de la red, que sería la aproximación de la solución exacta, $u(t)$. El objetivo de la red es hacer que se cumpla la ecuación diferencial y la condición inicial del problema de Cauchy (5.1). Por tanto, el funcional de coste podría ser el siguiente:

$$C = \frac{1}{N_c} \sum_{i=1}^{N_c} \left\| \hat{u}'(t_i) - \lambda t_i \right\|^2 ,$$

donde $\hat{u}'(t)$ es la derivada, obtenida mediante el algoritmo de *backpropagation*, de la función de la red neuronal $\hat{u}(t)$, N_c es el número total de tiempos de entrada, t_i, usados en el entrenamiento y $\|\cdot\|$ hace referencia a la norma 2. Es decir, se toma la función de coste como el error cuadrático medio entre ambos miembros de la ecuación diferencial, que es lo que se denomina «residuo» de la ecuación diferencial.

Además, como debe cumplirse la condición inicial, le sumamos a la función de coste anterior el término correspondiente, obteniendo el funcional de coste asociado a este problema

$$C = \frac{1}{N_c} \sum_{i=1}^{N_c} \left\| \hat{u}'(t_i) - \lambda t_i \right\|^2 + \left\| \hat{u}(t_{min}) - u_0 \right\|^2 . \tag{5.3}$$

Durante el entrenamiento de la red, se minimizará esta función de coste de forma que se satisfaga, «idealmente», la ecuación diferencial y la condición inicial. Más concretamente, tras el proceso de entrenamiento se tendría

$$C \longrightarrow 0 \Rightarrow \begin{cases} \|\hat{u}'(t_i) - \lambda t_i\| \longrightarrow 0 \\ \|\hat{u}(t_{min}) - u_0\| \longrightarrow 0 \end{cases} \Rightarrow \begin{cases} \hat{u}'(t_i) \longrightarrow \lambda t_i \\ \hat{u}(t_{min}) \longrightarrow u_0 \end{cases} \text{ para } \quad i = 1, \ldots, N_c.$$

Que es justo lo que queremos que cumpla la solución. Vemos que, en principio, la red neuronal cumple el problema de Cauchy (5.1), al menos de forma aproximada, en los tiempos de entrenamiento t_i, pero, si los hiperparámetros son adecuados, cabe esperar que se extrapole bien la solución al resto de puntos del dominio.

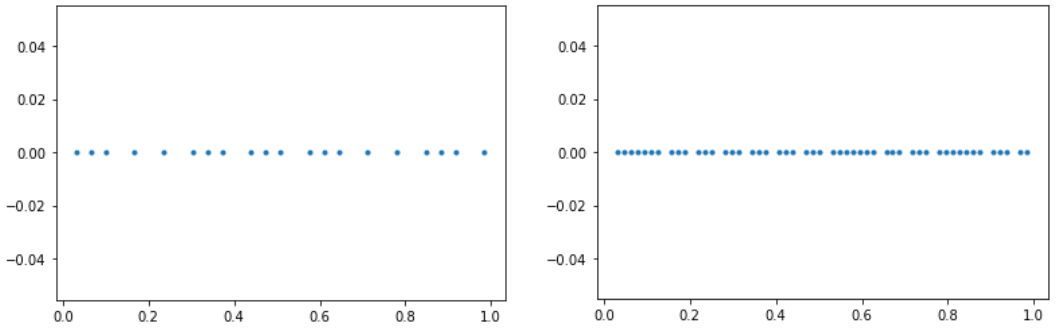
Podemos preguntarnos quiénes son los datos de entrenamiento t_i en este caso. La respuesta es que, a priori, no conocemos la solución exacta, luego no tenemos datos que imponer, solo sabemos que la solución debe cumplir la ecuación diferencial y debe valer u_0 en el tiempo inicial t_{min}. Lo único en lo que debemos pensar es en quiénes van a ser los datos de entrada con los que vamos a realizar el entrenamiento y en los que se aplicará la función de pérdida especificada anteriormente. En general,

tenemos dos opciones principales: la primera es tomar una partición del intervalo $[t_{min}, t_{max}]$, normalmente uniforme; la segunda es tomar puntos aleatorios en todo el intervalo.

Para tomar los puntos de forma aleatoria hay distintos métodos, algunos de los usuales son los que se describen a continuación.

Puntos de Sobol Es una sucesión casi aleatoria de números (aparentemente aleatorios, pero que provienen de un algoritmo no aleatorio). Esta sucesión surge con el objetivo de rellenar $[0, 1]^s$ minimizando los huecos, donde s es la dimensión del hipercubo (ver [21]).

Se puede ver una representación de estos puntos para el caso de una dimensión en la Figura 5.1.



(a) 20 primeros puntos. (b) 50 primeros puntos.

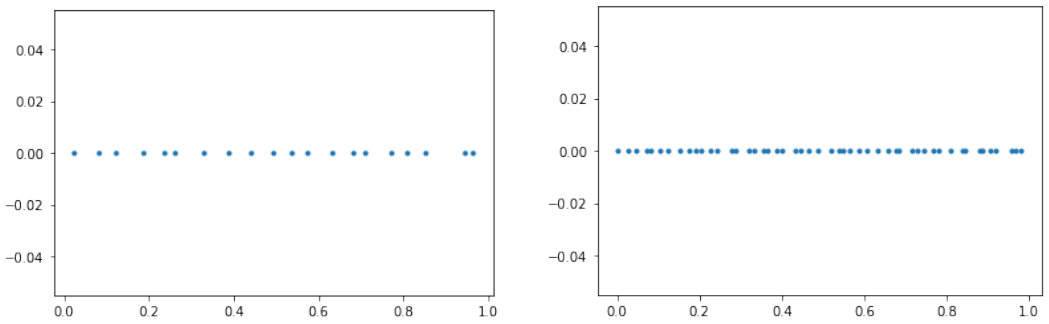
Figura 5.1: Puntos de Sobol en 1 dimensión.

Muestreo de hipercubo latino (*Latin Hypercube sampling*) Es un método estadístico para generar una muestra de números casi aleatorios. Si tenemos una cuadrícula que contiene las posiciones de una muestra dada formada por n símbolos, se dice que es «cuadrado latino» si es una matriz $n \times n$ en la que que, en cada casilla, hay una muestra de los n símbolos que aparece exactamente una vez en cada fila y columna (en el caso de dos dimensiones) (ver [22]). Naturalmente, «hipercubo latino» es la generalización de este concepto a un número arbitrario de dimensiones.

Podemos ver una representación gráfica de un muestreo de hipercubo latino en el caso de una dimensión en la Figura 5.2.

El ejemplo considerado muestra la resolución del problema directo, por tanto no necesitamos datos «concretos» (además de los puntos de entrenamiento t_i) para realizar el entrenamiento. En el caso de que se buscase resolver un problema inverso, sí que necesitaremos datos experimentales para determinar los parámetros desconocidos, como veremos más adelante.

En conclusión, para la aplicación de las redes neuronales a la resolución de ecuaciones diferenciales debemos tener en cuenta los siguientes aspectos:



(a) 20 puntos. (b) 50 puntos.

Figura 5.2: Muestreo de hipercubo latino (LHS) en 1 dimensión.

1. El número de neuronas en la capa de entrada y en la capa de salida deben ser el número de variables independientes y dependientes, respectivamente.

2. La función de coste debe contener el residuo de la ecuación diferencial, generalmente, el error cuadrático medio del mismo. Además, las condiciones adicionales (iniciales y/o de contorno) deben añadirse con un término más a la función de coste, como el error cuadrático medio del residuo de estas condiciones

3. Los datos de entrenamiento son, ahora, los datos de entrada que se usarán en dicho proceso (en el caso del problema directo). Para generarlos, debemos tomar, bien puntos aleatorios en el dominio en el que se busque solucionar el problema, bien una partición uniforme del mismo.

5.2. Problema directo para una EDO

Veamos un ejemplo de utilización de las redes neuronales para el caso de resolución de un problema directo similar al que se explica en la sección precedente. Además, se comparará el impacto que tiene la variación de distintos hiperparámetros que hemos visto anteriormente en la aproximación obtenida.

Consideramos el siguiente problema de Cauchy:

$$\begin{cases} u'(t) = 3t, & t \in [0, 5], \\ u(0) = 5. \end{cases} \tag{5.4}$$

Veamos primero la aproximación obtenida con la elección de parámetros que se ha considerado adecuada. Se tienen los datos y resultados de la simulación en la Tabla 5.1 y en la Figura 5.3.

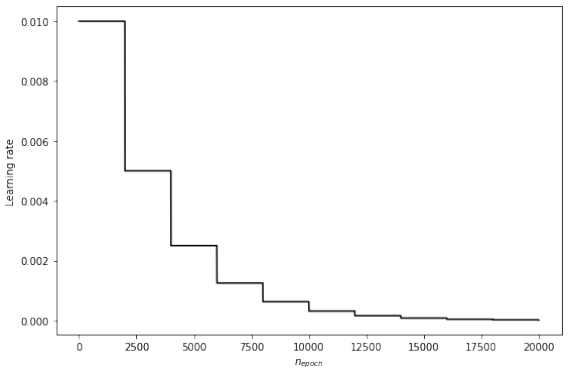
La inicialización de los parámetros internos de la red se realizará en todos los casos con la inicialización «Xavier Normal» (ver la Sección 2.2.2). Además, todas las gráficas en las que se represente la función de pérdida frente a las épocas, se realizarán en escala semi-logarítmica para poder apreciar bien la variación de la función.

Hiperparámetros				**Resultados**	
Activación	Sigmoide	**Estructura**	$[1, 10, 10, 1]$	**F. coste final**	$3 \cdot 10^{-6}$
Optimizador	Adamax	**Épocas**	20000	$\lvert\hat{u}(0) - u_0\rvert$	$7 \cdot 10^{-6}$
Learning rate	$l_r = 0.01$, $\times 0.5/(2000$ épocas$)$	**Nº puntos**	1000	**Tiempo**	$64s$

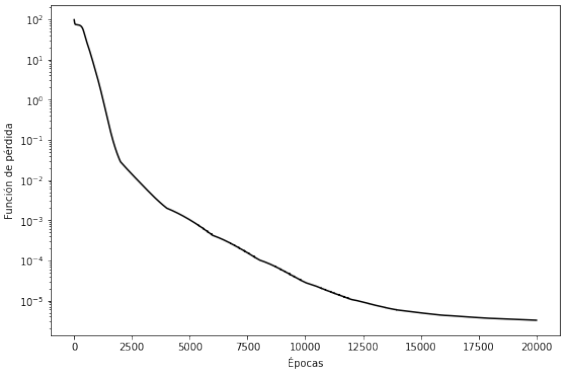
Tabla 5.1: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (5.4).

Los hiperparámetrios recogidos en la Tabla 5.1 fueron explicados en el Capítulo 2, aunque se hacen las siguientes aclaraciones: la **Estructura** hace referencia a la estructura de capas y neuronas de la red, así $[1, 10, 10, 1]$ nos indica que tiene una capa de entrada con 1 neurona, luego dos capas intermedias de 10 neuronas y una última capa de salida de 1 neurona; el **Nº puntos** es el número de datos de entrada t_i que se hacen pasar por la red en cada época del proceso de entrenamiento, sería N_c con la notación de la función de pérdida (5.3); la **F. coste final** es la evaluación de la función de coste tras el proceso de entrenamiento de la red neuronal; $|\hat{u}(0) - u_0|$ es el error absoluto cometido por la red neuronal en la condición inicial; el **Tiempo** es el tiempo total empleado por el ordenador en el proceso de entrenamiento.

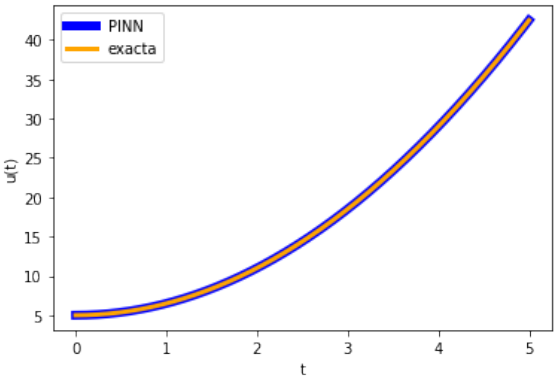
En la Figura 5.3 podemos ver como el *learning rate* variable permite al optimizador ajustar la función de coste con mucha precisión en un tiempo adecuado. En la gráfica c) de dicha figura se observa como la solución aproximada por la red neuronal coincide prácticamente con la solución exacta. Además, se puede comprobar este hecho teniendo en cuenta el valor de la función de coste tras el entrenamiento y el error en la condición inicial que se ve en la Tabla 5.1.



(a) Learning rate frente a las épocas.



(b) Función de pérdida frente a las épocas.



(c) Solución exacta («exacta») y aproximación obtenida del problema («PINN»).

Figura 5.3: Simulación con los valores adecuados e hiperparámetros de la Tabla 5.1. Resolución del problema (5.4).

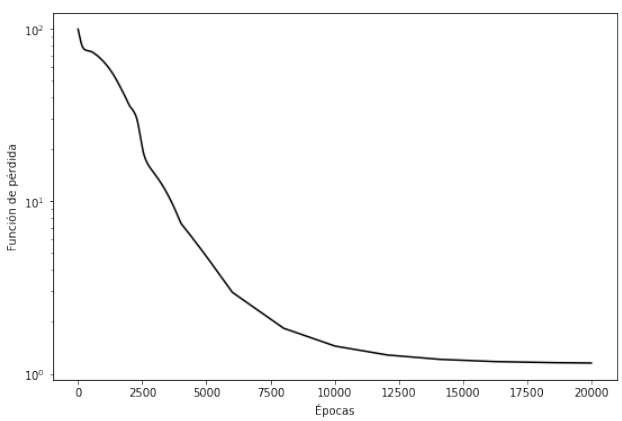
Estudiemos a continuación cómo afecta la variación de algunos hiperparámetros a la aproximación obtenida por la red neuronal.

5.2.1. Estructura de la red

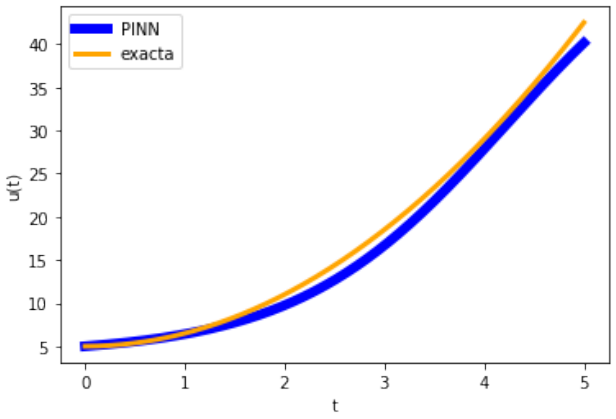
Veamos qué ocurre si se varía el número de parámetros internos de la red neuronal. Más concretamente, si se cambia la estructura de la misma, mientras que el resto de hiperparámetros se dejan sin variaciones.

En primer lugar, se utilizará una estructura con pocas neuronas, para poder apreciar la falta de «capacidad» de la red (no será capaz de ajustar bien con tan pocos parámetros), se usará una muy sencilla: $[1, 1, 1]$, es decir, una neurona de entrada, una neurona en la capa intermedia y una neurona de salida. Vemos la solución obtenida en la Figura 5.4.

Se puede observar que la red ya no ajusta tan bien como lo hacía antes. Ahora la función de la red neuronal puede obtenerse fácilmente: $\omega_2 \tanh(\omega_1 t + b_1) + b_2$, y con esta expresión no podemos ajustar tan bien la solución exacta. Al haber reducido



(a) Función de pérdida frente a épocas.

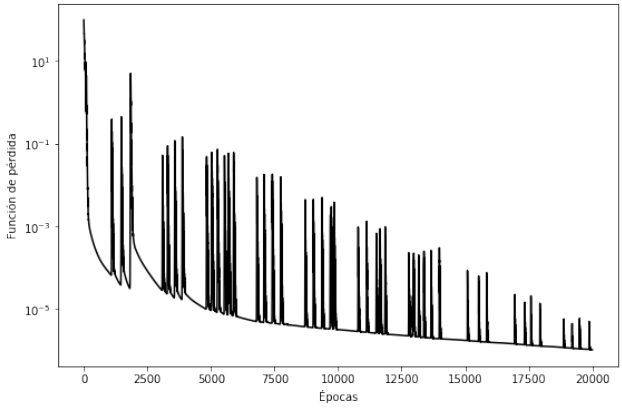


(b) Solución aproximada y solución exacta.

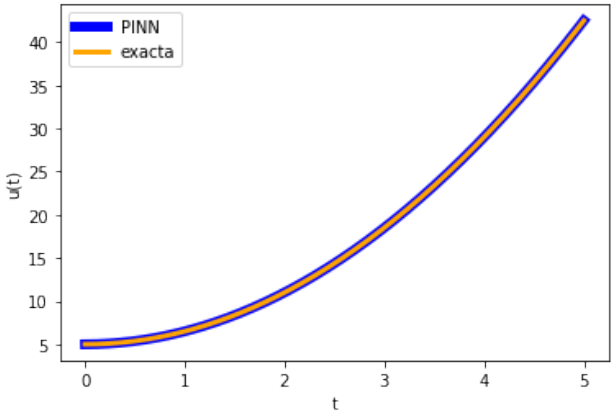
Figura 5.4: Resolución del problema (5.4) con muy pocos parámetros ($[1, 1, 1]$).

mucho el número de parámetros internos, el tiempo de entrenamiento ha pasado a ser de $44s$.

Veamos ahora qué ocurre si en lugar de pocos parámetros, tenemos demasiados. Usaremos, por ejemplo, la estructura $[1, 100, 100, 100, 100, 100, 100, 1]$, es decir, 1 neurona de entrada, otra de salida, y 6 capas intermedias con 100 neuronas cada una.



(a) Función de pérdida frente a épocas.



(b) Solución aproximada y solución exacta.

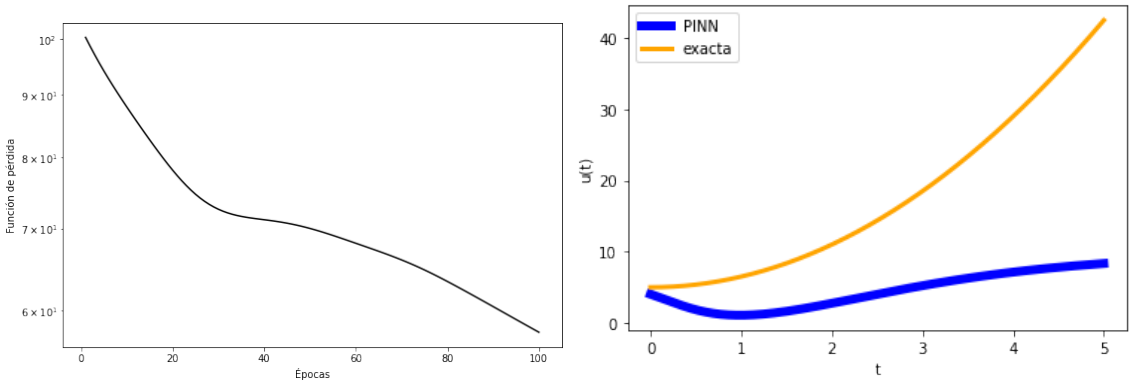
Figura 5.5: Resolución del problema (5.4) con demasiados parámetros $[1, 100, 100, 100, 100, 100, 100, 1]$.

Se puede observar en la Figura 5.5 como se producen muchas oscilaciones en la función de coste y, aunque la solución parece adecuada, pues se obtiene un valor final de la función de coste similar al obtenido con los parámetros adecuados, se debe tener en cuenta que el tiempo de simulación ha ascendido a $450s$, es decir, más de 6 veces el tiempo de simulación con un número adecuado de parámetros.

5.2.2. Número de épocas

Veamos ahora cómo afecta el número de épocas de la simulación en la aproximación obtenida. Este hiperparámetro esta ligado a la potencia del ordenador, pues, en general, un número muy grande implicará mucho tiempo de cálculo, mientras que uno demasiado pequeño limita el potencial de la red.

En primer lugar se realizará la simulación con 100 épocas. Los resultados obtenidos se recogen la Figura 5.6

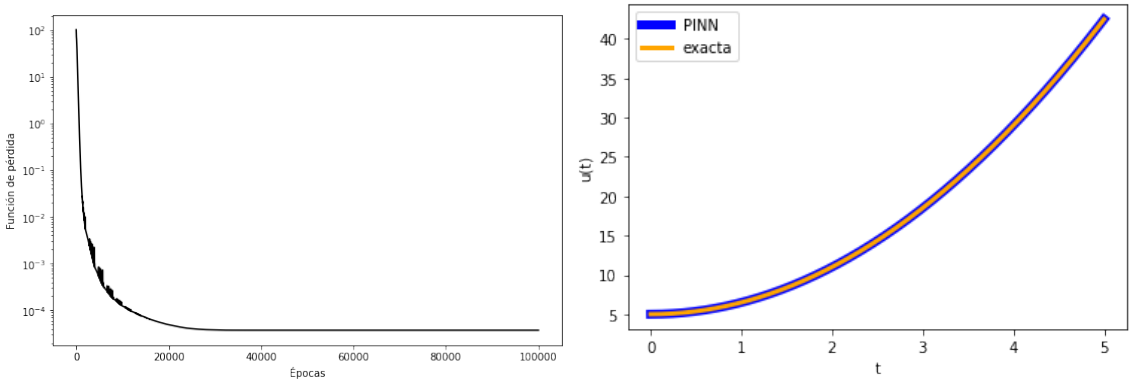


(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.6: Resolución del problema (5.4) con pocas épocas (100).

Es claro que no hemos dado a la red las épocas necesarias para que se aproxime lo suficiente a la solución exacta, se aprecia que el error tiene una clara tendencia a seguir disminuyendo, y se observa que la solución aproximada difiere mucho de la exacta.

Veamos ahora el otro extremo, cuando el número de épocas es muy alto, le damos a la simulación 100000 épocas (Figura 5.7).



(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

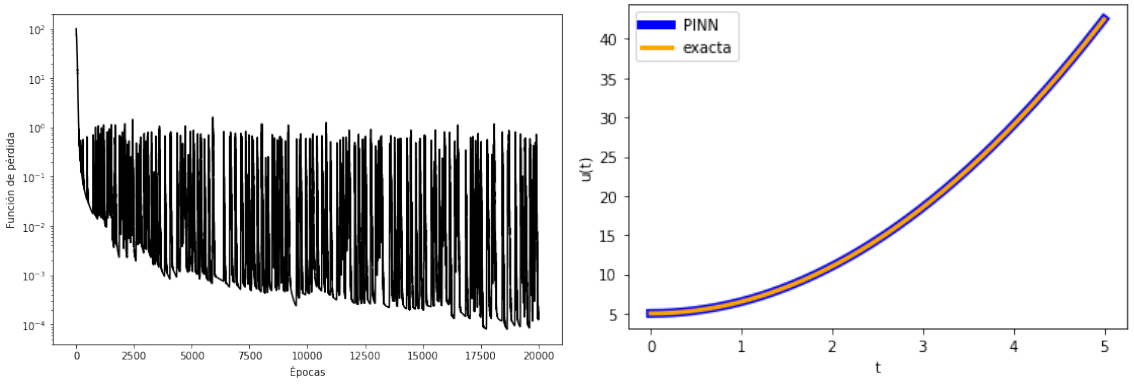
Figura 5.7: Resolución del problema (5.4) con muchas épocas (100000).

Ahora el escenario es muy distinto: el error ha disminuido mucho y la solución aproximada se ajusta mejor a la exacta. Sin embargo, se puede apreciar en la Figura 5.7 como la función de coste se ha «atascado» en un valor. El tiempo de simulación ha ascendido a $283s$, mientras que se observa que, desde la época 30000 aproximadamente, no se han producido cambios notables en la función de coste, luego esto supone una pérdida de recursos en la simulación.

5.2.3. Factor de entrenamiento (*learning rate*)

Estudiemos ahora cómo afecta el tamaño de paso a la simulación. Sabemos del Capítulo 2 que un paso demasiado pequeño supone un avance muy lento en la optimización, mientras que uno muy grande puede dar lugar a oscilaciones e incluso divergencia de la función de coste. Para que se pueda apreciar bien la influencia del paso, tomaremos un paso constante para estas simulaciones.

En primer lugar escojamos un paso grande: $l_r = 0.3$. Se tienen las gráficas obtenidas en la Figura 5.8. Aunque el error tiende a disminuir en un primer momento, ya que un paso grande agiliza la minimización al principio, presenta oscilaciones muy fuertes al llegar a valores más pequeños. No es capaz de realizar un ajuste «fino».

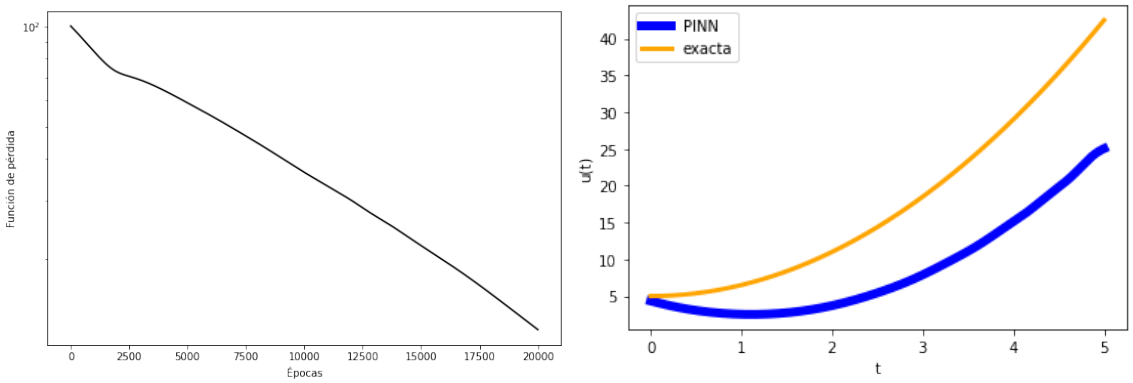


(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.8: Resolución del problema (5.4) con un paso constante grande ($l_r = 0.3$).

Por otra parte, se toma un paso muy pequeño ($l_r = 0.0001$), de forma que la red neuronal no será capaz de hacer cambios lo suficientemente grandes como para realizar un primer ajuste burdo, avanzando muy lentamente (Figura 5.9).

Se concluye que el paso grande puede ser útil al comienzo, mientras que, para realizar un ajuste más fino, es necesario un paso pequeño. Es por ello que el paso variable que se utilizó en la simulación de la Figura 5.3, es mucho más eficiente.

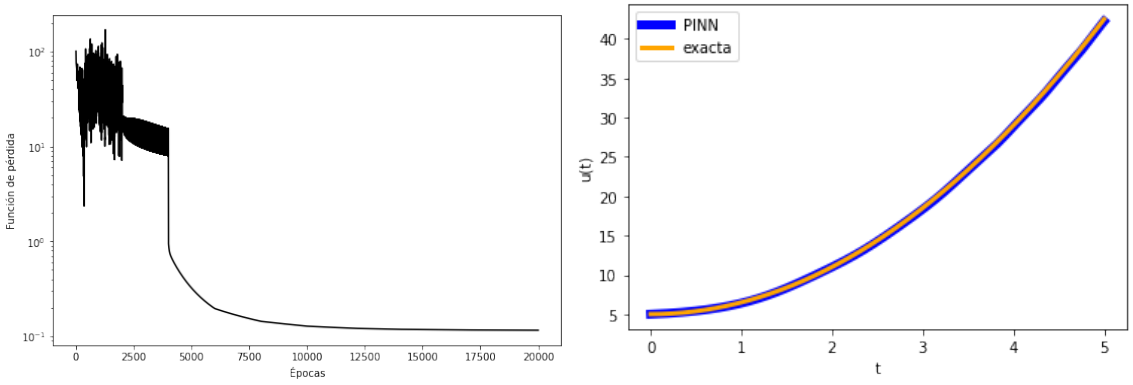


(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.9: Resolución del problema (5.4) con un paso constante muy pequeño ($l_r = 0.0001$).

5.2.4. Optimizador

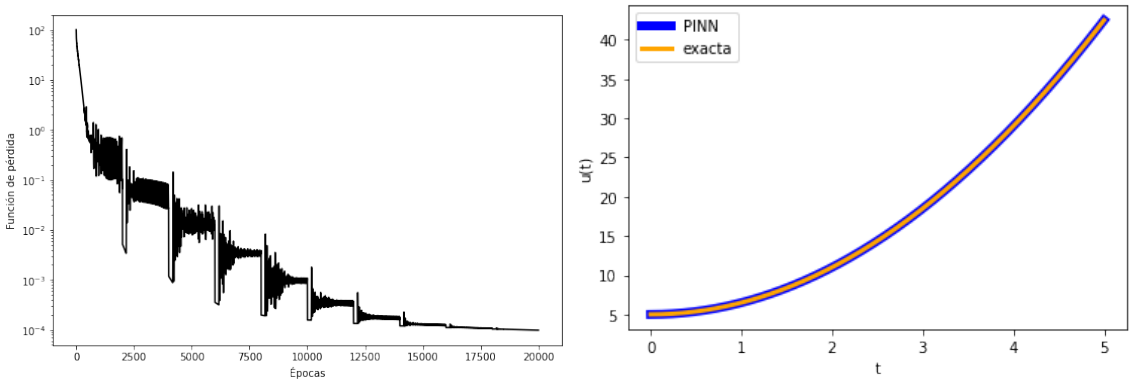
Probaremos ahora cómo actúan diferentes optimizadores (ver la Sección 2.2.7). Veamos en un primer lugar el optimizador SGD (Figura 5.10). Se puede ver que la función de pérdida al principio tiene unas fuertes oscilaciones, además, tiende a un valor del funcional de pérdida mayor que con el optimizador Adamax, luego es menos eficiente para nuestro problema.



(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.10: Resolución del problema (5.4) con el optimizador SGD.

Veamos ahora el optimizador RMSprop, en la Figura 5.11. En este caso, la función de pérdida tiende a un valor más bajo que el que se obtiene con el SGD, aunque sigue siendo algo superior al que se llega con Adamax. Además, se observan oscilaciones en la misma. Luego, podemos concluir que el Adamax parece un optimizador adecuado para nuestro problema.



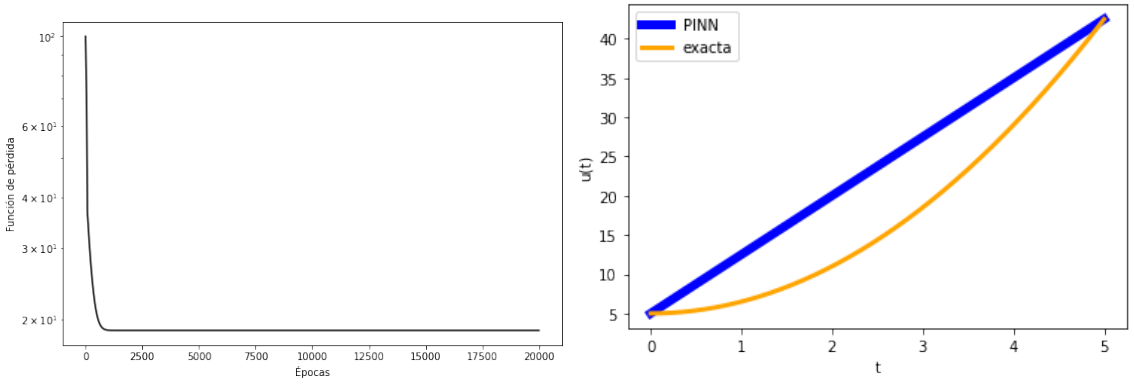
(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.11: Resolución del problema (5.4) con el optimizador RMSprop.

5.2.5. Función de activación

La función de activación es un hiperparámetro importante en la red neuronal (ver Sección 2.2.1), probemos varias opciones para ver cómo afecta a la aproximación obtenida.

En primer lugar, usaremos la función ReLU. Vemos la aproximación obtenida en la Figura 5.12. Para poder entender esta aproximación, debemos tener en cuenta que esta función de activación es lineal en la parte positiva (la negativa la anula), luego la función de la red también será lineal. Es por ello que la aproximación dista mucho de la real. No parece una buena función de activación para nuestro problema.

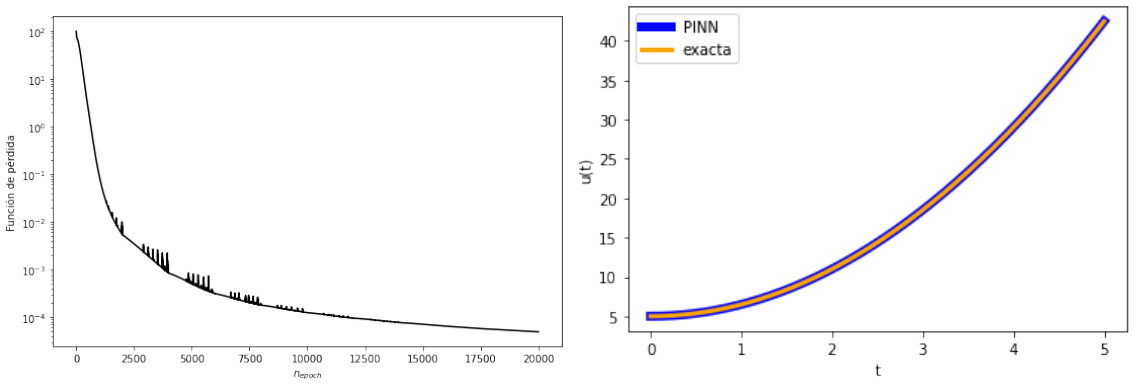


(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.12: Gráficas para la resolución del problema (5.4) con la función de activación ReLU.

Usemos ahora la función Tanh (Figura 5.13). Es una función similar a la sigmoide, en el sentido de que ambas tienen comportamientos asintóticos similares en $\pm\infty$, luego no es de sorprender que el resultado obtenido sea similar. Aunque ambas parecen actuar de forma adecuada, la sigmoide consigue reducir más la función de

coste, de forma que se ha tomado como parámetro adecuado.



(a) Función de pérdida frente a épocas. (b) Solución aproximada y solución exacta.

Figura 5.13: Resolución del problema (5.4) con la función de activación Tanh.

5.3. Problema directo para un sistema diferencial

Lo explicado anteriormente puede extenderse fácilmente a un sistema diferencial, especificando más neuronas en la capa de salida de la red. Además, la función de coste tendrá ahora un residuo por cada una de las ecuaciones del sistema diferencial, y se impondrán las condiciones iniciales correspondientes.

Podemos aproximar, por ejemplo, el modelo SIR para las epidemias (ver [24]), que responde al siguiente sistema:

$$\begin{cases} S'(t) = -\beta S(t)I(t), \\ I'(t) = \beta S(t)I(t) - \gamma I(t), & t \in [t_{min}, t_{max}], \\ R'(t) = \gamma I(t), \\ S(t_{min}) = S_0, I(t_{min}) = I_0, R(t_{min}) = R_0, \end{cases} \tag{5.5}$$

donde $S(t)$ son los individuos susceptibles, aquellos que no han pasado la enfermedad todavía, $I(t)$ los infectados, y $R(t)$ los recuperados en el instante t. Las constantes $\beta, \gamma \geq 0$ son dos parámetros que determinan la evolución de la población. En este modelo de epidemias simple, se supone que los individuos no mueren por la enfermedad, ni tampoco se producen nacimientos, ya que el número total de individuos en la población se mantiene constante $S'(t) + I'(t) + R'(t) = 0$.

El funcional de pérdida es similar al usado para la ecuación diferencial (5.3), pero

adaptado al sistema:

$$C = \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{S}'(t_i)+\beta\hat{S}(t_i)\hat{I}(t_i)\right\|^2 + \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{I}'(t_i)-\beta\hat{S}(t_i)\hat{I}(t_i)+\gamma\hat{I}(t_i)\right\|^2$$
$$+\frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{R}'(t_i)-\gamma\hat{I}(t_i)\right\|^2 + \left\|\hat{S}(t_{min})-S_0\right\|^2 + \left\|\hat{I}(t_{min})-I_0\right\|^2 + \left\|\hat{R}(t_{min})-R_0\right\|^2,$$

donde $\hat{S}(t), \hat{I}(t), \hat{R}(t)$ son las salidas de la red en el tiempo t, soluciones aproximadas del modelo SIR (5.5) y los tiempos t_i son los N_c valores utilizados en el entrenamiento de la red neuronal.

Este sistema diferencial no se puede resolver de manera exacta, por tanto para comparar y comprobar el funcionamiento de la red se utilizará un solucionador externo ya implementado en Python, como puede ser `odeint`, del que hablaremos en el capítulo final dedicado a las herramientas informáticas.

El sistema que vamos a resolver es el siguiente:

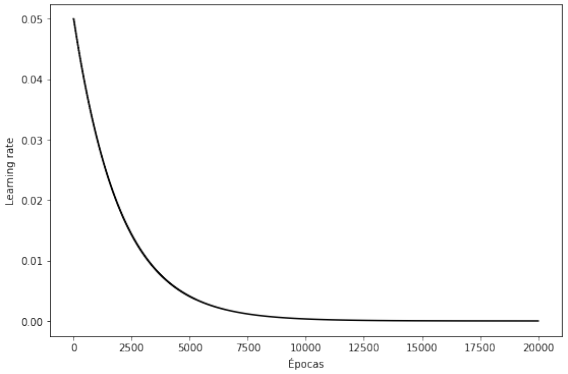
$$\begin{cases} S'(t) = -0.1S(t)I(t), \\ I'(t) = 0.1S(t)I(t) - I(t), \qquad t \in [0,3], \\ R'(t) = I(t), \\ S(0) = 100, I(0) = 50, R(0) = 0. \end{cases} \tag{5.6}$$

Los parámetros de la red escogidos para la simulación se recogen en la Tabla 5.2 y los resultados de la simulación en la Figura 5.14.

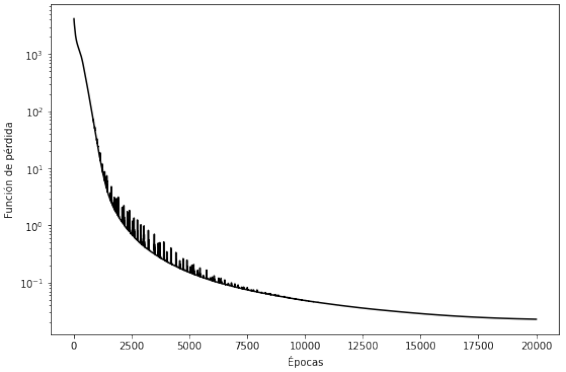
Hiperparámetros				**Resultados**	
Activación	Tanh	**Estructura**	$[1, 10, 10, 3]$	**F. coste final**	$2.3 \cdot 10^{-2}$
Optimizador	Adamax	**Épocas**	20000	$\|\|\hat{u}(0) - u_0\|\|$	$1.5 \cdot 10^{-2}$
Learning rate	$l_r = 0.05$, $\times 0.99/(20$ épocas$)$	**Nº puntos**	1000	**Tiempo**	$138s$

Tabla 5.2: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (5.6).

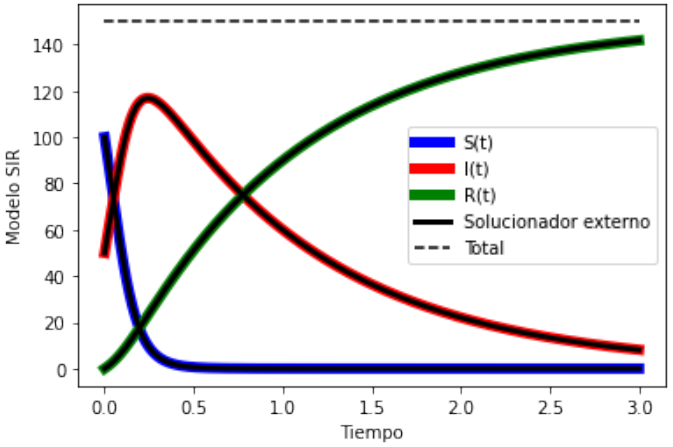
Podemos observar como la aproximación obtenida por la red neuronal se ajusta bien a la esperada, obtenida con `odeint`. La función de pérdida es minimizada de forma adecuada. Es comprensible que el tiempo de computación y el mínimo alcanzado del funcional de pérdida sean mayores, ya que ahora hay 3 residuos (tenemos 3 ecuaciones diferenciales) en la función de pérdida y 3 condiciones iniciales impuestas.



(a) Learning rate frente a las épocas.



(b) Función de pérdida frente a las épocas.



(c) Solución aproximada por la red neuronal (colores), solución aproximada dada por un solucionador externo a la red (lineas continuas negras) y población total (discontinua).

Figura 5.14: Simulación con los valores adecuados e hiperparámetros de la Tabla 5.2. Resolución del problema (5.6).

Capítulo 6

PINN para el ajuste de parámetros

Una vez se ha visto cómo se pueden resolver problemas directos, tanto para ecuaciones como para sistemas diferenciales, estamos en las condiciones de poder resolver problemas de ajuste para la determinación de parámetros desconocidos.

Vamos a diferenciar tres tipos de problemas de ajuste de parámetros:

Tipo 1 En estos problemas el parámetro desconocido puede obtenerse *a posteriori* a partir de la solución aproximada. Por ejemplo, si el parámetro es la condición inicial, se puede obtener tras el entrenamiento evaluando la función obtenida por la red neuronal en el tiempo inicial (ver Sección 6.1).

Tipo 2 El parámetro desconocido es un parámetro discreto (un valor real) que no puede obtenerse *a posteriori*. Por ejemplo, los parámetros β, γ del modelo SIR (5.5) (ver Sección 6.2).

Tipo 3 En este caso el parámetro es una función continua. Por ejemplo, en un problema de control (ver Sección 6.3).

Se verán a continuación cada uno de ellos con más detalles.

6.1. Obtención de parámetros a posteriori

En estos problemas tipo 1, podemos obtener el parámetro en cuestión a partir de la solución aproximada. Desde el punto de vista computacional, es un problema análogo al problema directo, ya que para obtener dicho parámetro sólo hay que evaluar la aproximación obtenida por la red neuronal donde sea necesario.

Por ejemplo, podemos resolver el siguiente problema, donde el parámetro desconocido es la condición inicial e imponemos la condición final, es decir, suponemos conocida la función en el tiempo final:

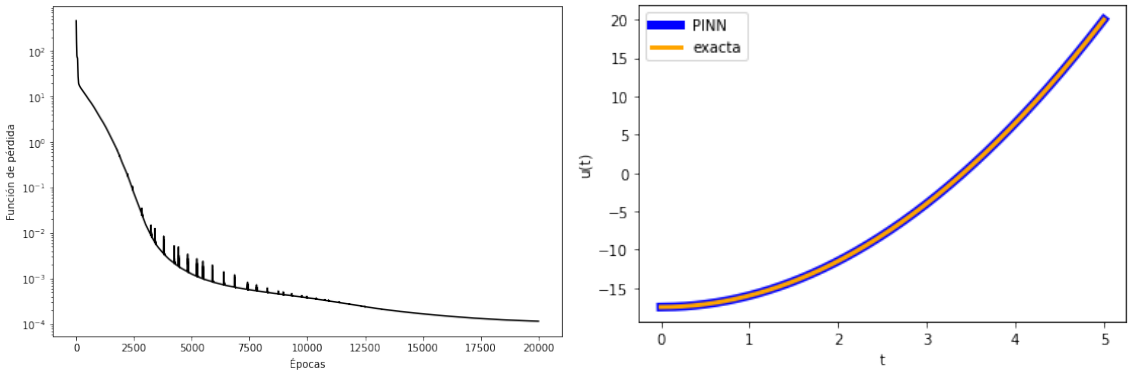
$$\begin{cases} u'(t) = 3t, & t \in [0,5], \\ u(5) = 20, \\ u(0) = \lambda, \end{cases} \tag{6.1}$$

siendo $\lambda \in \mathbb{R}$ desconocido.

De acuerdo con la solución exacta (5.2), el valor exacto del parámetro que buscamos es $\lambda_{exa} = -17.5$. Los hiperparámetros y resultados de la simalación pueden verse en la Tabla 6.1 y en la Figura 6.1.

Hiperparámetros				**Resultados**	
Activación	Sigmoid	**Estructura**	$[1, 10, 10, 1]$	**F. coste final**	$1 \cdot 10^{-4}$
Optimizador	Adamax	**Épocas**	20000	$\lvert\hat{u}(0) - \lambda_{exa}\rvert$	$1.9 \cdot 10^{-3}$
Learning rate	$l_r = 0.1$, $\times 0.8/(500$ épocas$)$	**Nº puntos**	1000	**Tiempo**	$62s$

Tabla 6.1: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (6.1).



(a) Función de pérdida frente a las épocas.

(b) Solución aproximada por la red neuronal y solución exacta.

Figura 6.1: Simulación con los hiperparámetros de la Tabla 6.1. Resolución del problema (6.1).

Se puede observar como la red neuronal aproxima de forma eficaz el problema con la condición final conocida, alcanzando valores muy pequeños para la función de pérdida, con una actuación similar a la del problema directo.

6.2. Ajuste de parámetros discretos

En este tipo de problemas tipo 2 buscamos aproximar uno o varios parámetros desconocidos. Hasta ahora no hemos necesitado observaciones de las variables de la ecuación para obtener una aproximación de la solución, bastaba con imponer a la función de la red neuronal que debe cumplir la ecuación diferencial y las condiciones

iniciales. Ahora el problema cambia, la función de pérdida no sólo debe incluir el residuo de las ecuaciones diferenciales y las condiciones iniciales, para poder determinar parámetros desconocidos, necesitamos darle observaciones adicionales de la solución en ciertos tiempos, de esta forma podremos obtener los parámetros que mejor ajusten la solución a las observaciones.

La función de pérdida para el caso de una ecuación diferencial ordinaria $u'(t) = f(u,t)$, junto con la condición inicial $u(t_{min}) = u_0$ será, por tanto,

$$C = \frac{1}{N_c}\sum_{i=1}^{N_c} \|\hat{u}'(t_i) - f(\hat{u}, t_i)\|^2 + \|\hat{u}(t_{min}) - u_0\|^2 + \frac{1}{N_o}\sum_{k=1}^{N_o} \|\hat{u}(t_k) - u_k\|^2,$$

donde N_c denota el número total de «puntos de colocación» (*collocations points*), que son los valores de entrada utilizados para evaluar el residuo de la ecuación diferencial; N_o es el número total de observaciones u_k en el intervalo de resolución, es decir, los valores experimentales que sabemos (o intuimos) que sigue la solución [23].

Las preguntas ahora son: ¿cómo introducimos los parámetros desconocidos en la red y cómo se ajustarán sus valores? Los parámetros desconocidos se introducirán como parámetros entrenables de la red, de manera que podrán ser ajustados, de acuerdo con el descenso de gradiente, junto con el resto de parámetros internos. Al igual que el resto de parámetros de la red neuronal, debemos darle un valor inicial.

Veamos a continuación la resolución de dos ejemplos para este tipo de ajuste de parámetros, uno aplicándolo a una ecuación diferencial ordinaria y otro a un sistema.

6.2.1. Ejemplo de ajuste de parámetros discretos para una EDO

Resolveremos el problema de Cauchy (5.1), donde λ será un parámetro desconocido esta vez, concretamente se resolverá el siguiente problema: dados u_k, hallar $\lambda \in \mathbb{R}$ tal que $u(t_k) = u_k$, con k desde 1 hasta el número de observaciones que se utilice, siendo u la solución de

$$\begin{cases} u'(t) = \lambda t, & t \in [0,5], \\ u(0) = 2. \end{cases} \tag{6.2}$$

El objetivo ahora es: dado un conjunto de observaciones u_k, que se supone que sigue este modelo, determinar el parámetro λ que hace que la solución los ajuste de la mejor forma posible. En la «vida real», cuando trabajamos con modelos biológicos como el modelo SIR introducido anteriormente en (5.5), las observaciones las podemos obtener a partir de los datos reales que, *a priori*, sigue el modelo. En nuestro caso, no contamos con observaciones reales, pero, como tenemos la solución exacta (5.2), lo que haremos será utilizar como datos los valores de la solución exacta para un parámetro dado, y veremos si la red consigue aproximar dicho valor.

Los datos los obtendremos con $\lambda = -2$, que será el valor que debe aproximar la red neuronal. Además, para los tiempos t_i se toma una partición uniforme del

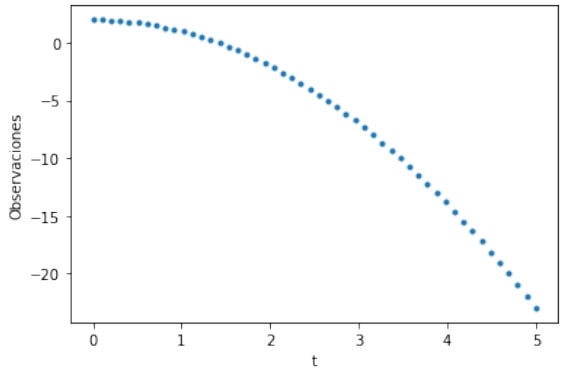
intervalo con un número determinado de puntos (especificados en la Tabla 6.2 de hiperparámetros de la simulación).

Los hiperparámetros escogidos para la simulación se recogen en la Tabla 6.2, y los resultados obtenidos en la Figura 6.2.

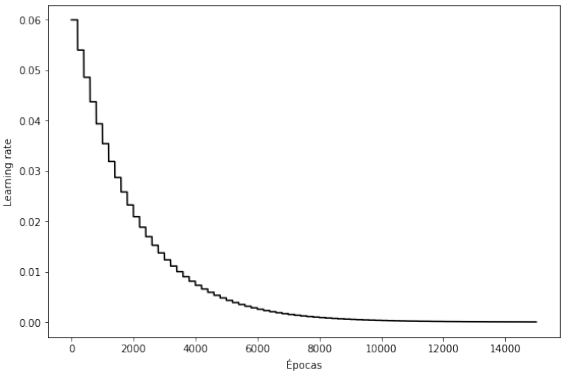
Hiperparámetros				**Resultados**	
Activación	Sigmoid	**Estructura**	$[1, 10, 10, 10, 1]$	**F. coste final**	$4 \cdot 10^{-4}$
Optimizador	Adamax	**Épocas**	15000	$\lvert\hat{\lambda} - \lambda_{exa}\rvert$	$3 \cdot 10^{-4}$
Learning rate	$l_r = 0.06$, $\times 0.9/(200$ épocas)	**Nº puntos**	1000	**Tiempo**	$65s$
Inicialización λ	$\lambda_0 = 3$	**Nº obs.**	50		

Tabla 6.2: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (6.2).

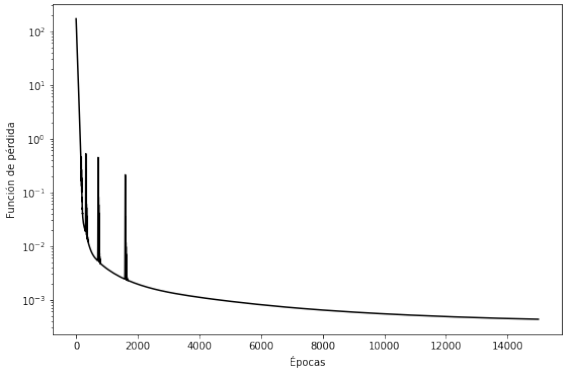
Podemos observar en la Figura 6.2 como el parámetro desconocido λ llega rápidamente al valor real -2, con un error final de menos del $0.02\,\%$ que es bastante aceptable. Como podemos ver, la aproximación de la red neuronal se ajusta satisfactoriamente a la solución exacta.



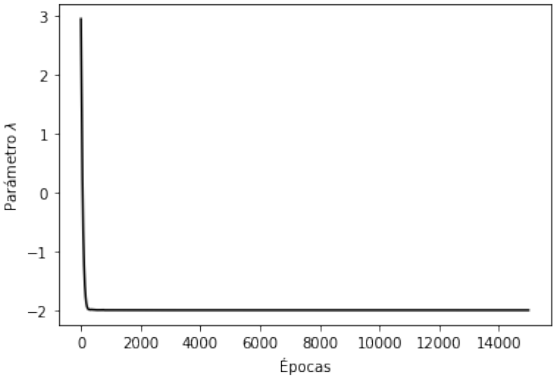
(a) Observaciones usadas para el entrenamiento, a partir de la solución real con $\lambda = -2$.



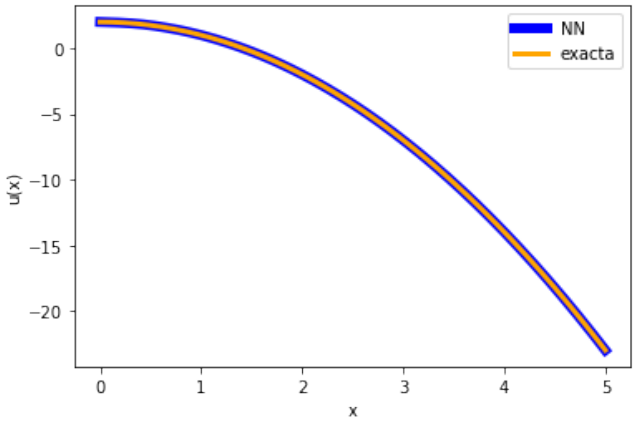
(b) Learning rate frente a las épocas.



(c) Función de pérdida frente a las épocas.



(d) Evolución del parámetro desconocido λ en función de las épocas.



(e) Solución aproximada por la red neuronal y solución exacta.

Figura 6.2: Simulación con los hiperparámetros de la Tabla 6.2. Resolución del problema (6.2).

6.2.2. Ejemplo de ajuste de parámetros discretos para un sistema de EDO's

Veamos en esta sección un problema inverso de tipo 2 aplicado a un sistema diferencial. Además, en este caso tendremos 2 parámetros desconocidos, en lugar de 1 y sólo contaremos con información de algunas variables. Se realizarán también algunas simulaciones para ver el funcionamiento de la red neuronal en el caso de que las observaciones introducidas tengan un porcentaje de ruido determinado, y en el caso de que solo se tenga información en una parte del dominio de resolución.

El caso de sistema que vamos a resolver es el modelo SIR que ya ha sido introducido en (5.5), pero ahora tanto β, γ, como S_0 serán desconocidos. Concretamente:

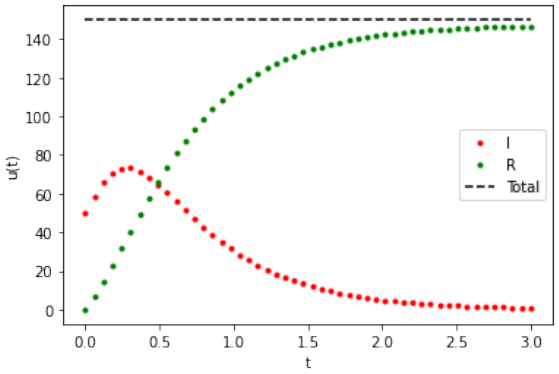
$$\begin{cases} S'(t) = -\beta S(t)I(t), \\ I'(t) = \beta S(t)I(t) - \gamma I(t), \qquad t \in [0,3], \\ R'(t) = \gamma I(t), \\ S(0) = S_0, I(0) = 50, R(0) = 0. \end{cases} \tag{6.3}$$

Además, no tendremos observaciones de todas las variables S, I, R, sólo dispondremos de observaciones de los infectados I y de los recuperados R. Esta situación puede darse en la vida real, ya que podríamos no poder medir fácilmente todas las variables de nuestro modelo. ¿De dónde obtenemos las observaciones? El modelo SIR no tiene una solución analítica y, como no tenemos acceso a datos reales, usaremos un solucionador externo a la red para obtener los datos, como puede ser la función `odeint` de Python, que veremos en el Capítulo 8 de útiles informáticos. Estas observaciones se obtendrán para unos valores de β,γ y S_0 concretos, que son los valores reales que la red neuronal deberá aproximar. Además, como no tenemos datos de $S(t)$, el valor de la condición inicial $S(0)$ también es desconocido, aunque, como ya vimos en los problemas de ajuste tipo 1, no es necesario que forme parte de los parámetros entrenables de la red neuronal, pues lo podremos obtener *a posteriori*, una vez la red neuronal esté entrenada.

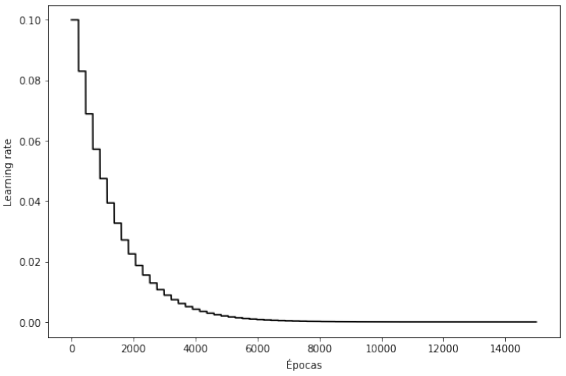
Los hiperparámetros de la simulación están recogidos en la Tabla 6.3 y los resultados de la simulación y las observaciones utilizadas se encuentran en la Figura 6.3.

Hiperparámetros				**Resultados**	
Activación	Sigmoid	**Estructura**	$[1, 15, 15, 15, 3]$	**F. coste final**	$4.4 \cdot 10^{-3}$
Optimizador	Adamax	**Épocas**	15000	$\|\|\hat{\lambda} - \lambda_{exa}\|\|$	$4 \cdot 10^{-5}$
Learning rate	$l_r = 0.1$, $\times 0.83/(230\text{épocas})$	**Nº puntos**	1000	$\|\hat{S}(0) - S_0\|$	$3 \cdot 10^{-2}$
Inicialización λ	$\beta = 2$ $\gamma = -3$	**Nº obs.**	50	**Tiempo**	$114s$

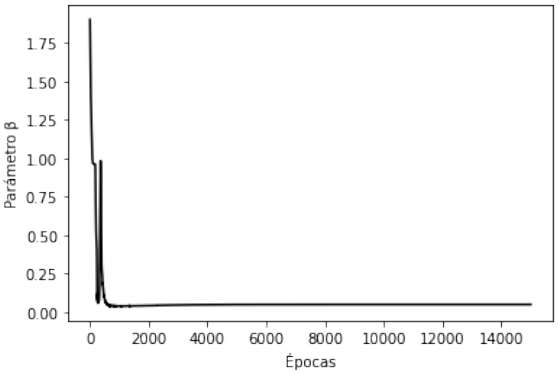
Tabla 6.3: Hiperparámetros para la simulación con los valores adecuados ($\lambda = (\beta, \gamma)$). Resolución del problema (6.3).



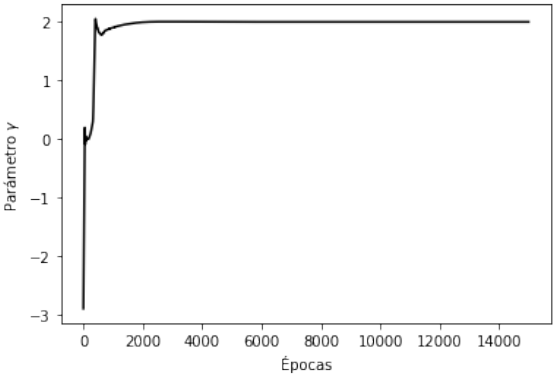
(a) Observaciones obtenidas con solucionador externo para $\beta = 0.05, \gamma = 2, S_0 = 100$.



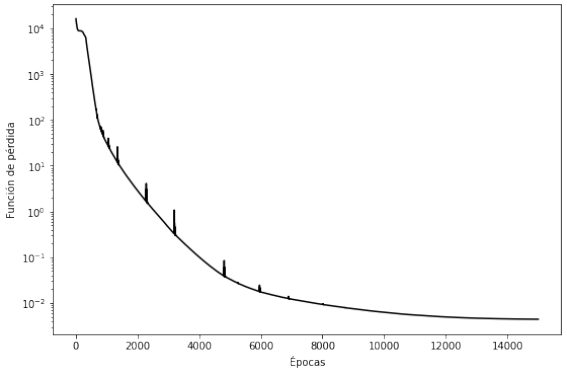
(b) Learning rate frente a las épocas.



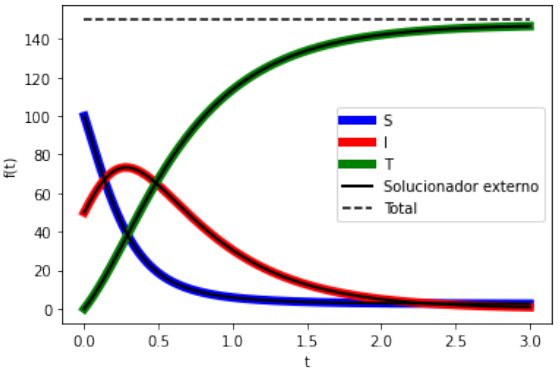
(c) Parámetro desconocido β frente a épocas.



(d) Parámetro desconocido γ frente a épocas.



(e) Función de pérdida frente a épocas.



(f) Solución aproximada por la red neuronal y solución obtenida con una función externa a la red.

Figura 6.3: Simulación con los hiperparámetros de la Tabla 6.3. Resolución del problema (6.3).

Se observa, al igual que en el caso de una EDO, que ambos parámetros son ajustados de forma adecuada y en pocas épocas, ya que podemos apreciar que en torno a la época 2000 se estabilizan en el valor exacto ($\beta = 0.05, \gamma = 2$). Además, la red neuronal es capaz de aproximar de forma satisfactoria la población de susceptibles

$S(t)$ aunque no hayamos introducido ninguna observación sobre ella. La función de pérdida llega a valores muy pequeños, lo que nos indica que se minimiza correctamente.

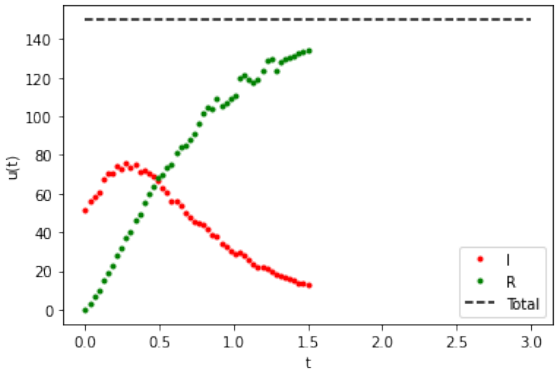
Otro de los aspectos en los que podemos pensar, es en que en la vida real, las observaciones que tengamos de medidas no seguirán de forma tan precisa las ecuaciones diferenciales de nuestro problema, además no tenemos por qué tener observaciones de las variables en todo el dominio. Por esa razón, se va a realizar otra simulación introduciendo ruido en los datos obtenidos con el solucionador externo, y, además, se van a tomar estas observaciones solo en la mitad del intervalo de resolución, esto es, en $[0, 1.5]$.

Los hiperparámetros escogidos para la simulación se recogen en la Tabla 6.4, y las gráficas obtenidas en la Figura 6.4.

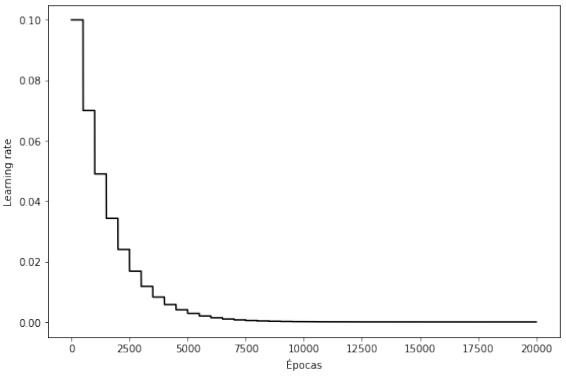
Hiperparámetros				**Resultados**	
Activación	Sigmoid	**Estructura**	$[1, 15, 15, 15, 3]$	**F. coste final**	5.22
Optimizador	Adamax	**Épocas**	20000	$\|\|\hat{\lambda} - \lambda_{exa}\|\|$	$2 \cdot 10^{-2}$
Learning rate	$l_r = 0.1$, $\times 0.7$/(500 épocas)	**Nº puntos**	1000	$\|\hat{S}(0) - S_0\|$	1.6
Inicialización λ	$\beta = 2$, $\gamma = -3$	**Nº obs.**	50, en $[0, 1.5]$	**Tiempo**	$160s$
Ruido	3 %				

Tabla 6.4: Hiperparámetros para la simulación con los valores adecuados ($\lambda = (\beta, \gamma)$). Resolución del problema (6.3).

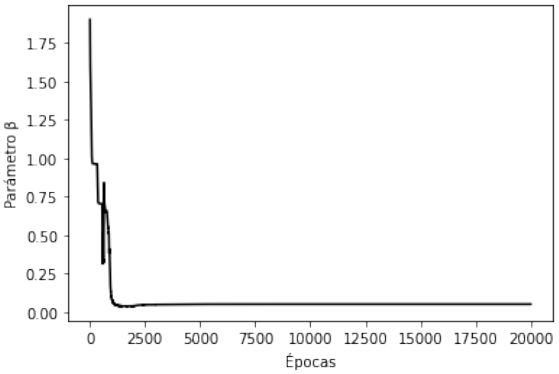
En este caso, no es de extrañar que el valor final de la función de pérdida sea mayor, ya que el error que se comete al aproximar los puntos con ruido es mayor que sin él. Además, al tener ruido, puede haber varias soluciones muy próximas que «disten» poco de la solución concreta que buscamos, luego se entiende que el error sea mayor. Sin embargo, obtenemos los parámetros buscados (β, γ y S_0) con un error relativo menor al 3 % en los 3 casos. Se puede observar como la función de pérdida y los parámetros alcanzan valores «estables» en pocas épocas, aun así, se han dado más épocas para intentar que la red nos de valores más cercanos al exacto, ya que este problema es, necesariamente, más complejo.



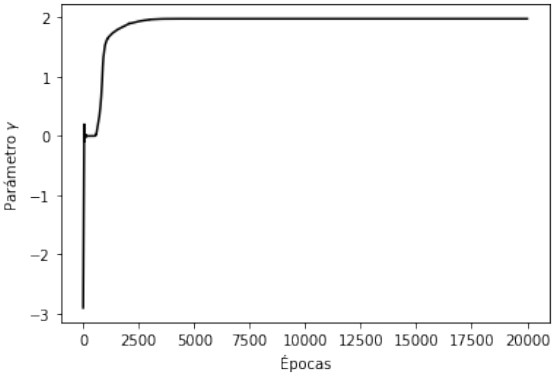
(a) Observaciones a partir de la solución con solucionador externo con $\beta = 0.05, \gamma = 2, S_0 = 100$. Tiene un 3 % de ruido y, además, no están en todo el dominio.



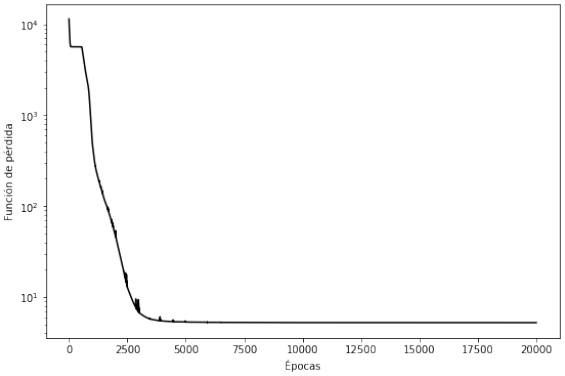
(b) Learning rate frente a las épocas.



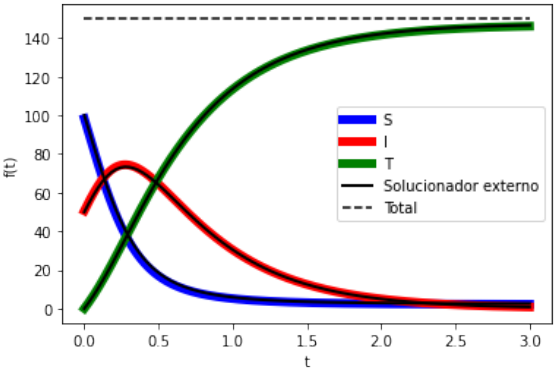
(c) Parámetro desconocido γ frente a épocas.



(d) Parámetro desconocido γ frente a épocas.



(e) Función de pérdida frente a épocas.



(f) Solución aproximada por la red neuronal y solución obtenida con una función externa a la red.

Figura 6.4: Gráficas para la simulación con los hiperparámetros de la Tabla 6.4. Resolución del problema (6.3).

6.3. Ajuste de parámetros continuos

En estos problemas tipo 3, el objetivo es obtener una función, ya no es un parámetro discreto como antes, que minimice una condición buscada siguiendo un problema diferencial. Es un problema muy distinto, pues ahora no podemos hacer lo que se hizo en la sección anterior: añadir un parámetro extra a los parámetros internos da la red, de forma que se ajustará en el proceso de entrenamiento al igual que el resto (ver Sección 6.2).

Hay dos formas principales de abordar un problema de este tipo: añadiendo una salida extra a la red neuronal, que será la función buscada, o creando una red neuronal auxiliar para la función desconocida, que se entrenaría de forma simultánea. En este trabajo usaremos la primera opción, por ser la más asequible.

Hasta ahora teníamos tantas neuronas de salida como variables dependientes tuviese nuestro problema diferencial, para este problema se añadirá una nueva neurona en la capa de salida de la red neuronal, que será la función a determinar. Por ejemplo, en el caso del modelo SIR (5.5) teníamos 3 variables que dependían del tiempo: $S(t), I(t), R(t)$, por tanto, la red tenía 3 neuronas de salida (S, I, R) y una neurona de entrada (el tiempo); añadimos ahora una nueva salida que será el control que queremos obtener, que es una función del tiempo.

Como buscamos que se cumpla una condición concreta, debemos añadir un nuevo sumando a nuestro funcional de coste, para que sea minimizado en el entrenamiento, junto al resto de términos. Todo esto puede entenderse mucho mejor con un ejemplo concreto que pasamos a exponer a continuación.

6.3.1. Ejemplo «educativo» de ajuste de parámetros continuos

Como primer ejemplo utilizaremos el sistema SIR 5.5 que ya hemos visto anteriormente . Supongamos que el parámetro β ya no es un valor real, sino que ahora es una función, es decir, tenemos el sistema diferencial siguiente:

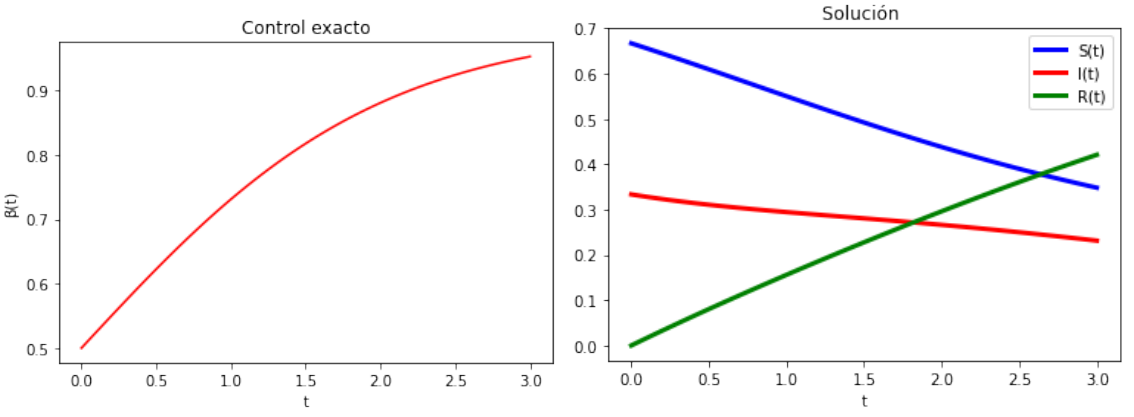
$$\begin{cases} S'(t) = -\beta(t)S(t)I(t), \\ I'(t) = \beta(t)S(t)I(t) - 0.5I(t), & t \in [0,3], \\ R'(t) = 0.5I(t), \\ [S(0), I(0), R(0)] = {}^{[S_0,I_0,R_0]}/_{(S_0+I_0+R_0)}. \end{cases} \tag{6.4}$$

Buscamos determinar $\beta(t)$ imponiendo una condición extra: que los infectados decrezcan de una manera determinada. Este problema puede ser mirado como un problema de control en el que la función β actúa como un control sobre el sistema, teniendo como objetivo que la solución de 6.4 cumpla una determinada condición (ver más adelante). Además, se hace notar que el sistema se ha normalizado, es decir, la suma de los susceptibles, infectados y recuperados se ha hecho igual a 1. Esto se

hace para que durante el entrenamiento, la magnitud de los números no suponga un problema. Hasta ahora no ha sido necesario, pues se ha trabajado con problemas «sencillos», pero cuando el problema es más complejo, es recomendable.

Otro aspecto a tener en cuenta son las restricciones en la función $\beta(t)$, en este caso, no tendría sentido un $\beta(t)$ que tome valores negativos, ya que esto supondría que la población total no se mantiene constante, fenómeno que no contemplamos en el modelo. Esta restricción puede imponerse de varias formas: introduciendo un término que penalice los valores negativos de $\beta(t)$, de forma que a la red neuronal no le interese tomarlos ya que aumentaría el error; o bien imponiéndolo desde la propia estructura de la red neuronal, por ejemplo, aplicando la función sigmoide a las neuronas de la capa de salida, de esta forma la salida sería positiva. Esta última opción es más óptima a la hora del entrenamiento, pues es una condición más que se deba cumplir, sino que ya viene impuesta por la propia estructura, será lo que se lleve a cabo.

Vamos a generar un ejemplo para saber de antemano lo que debemos obtener: resolveremos el problema utilizando un algoritmo externo a la red con una función $\beta(t)$ concreta, e impondremos que los infectados de nuestra red neuronal sigan la solución dada por este algoritmo. La función $\beta(t)$ que utilizaremos será la sigmoide, que podemos ver en la Figura 6.5 (a). Si utilizamos la función externa `odeint` para resolver el sistema (6.4) con esta función $\beta(t)$, obtenemos las aproximaciones de la Figura 6.5 (b).



(a) Control exacto $\beta(t)$ para (6.4).

(b) Solución para el sistema (6.4) con el control asociado (a), obtenida con `odeint`.

Figura 6.5: Control y solución para el problema (6.4).

Luego la condición que impondremos sobre la solución de 6.4 será que los infectados sigan esa curva concreta. Es decir, nuestro estado deseado será aquel en el que los infectados sigan la curva de la Figura 6.6.

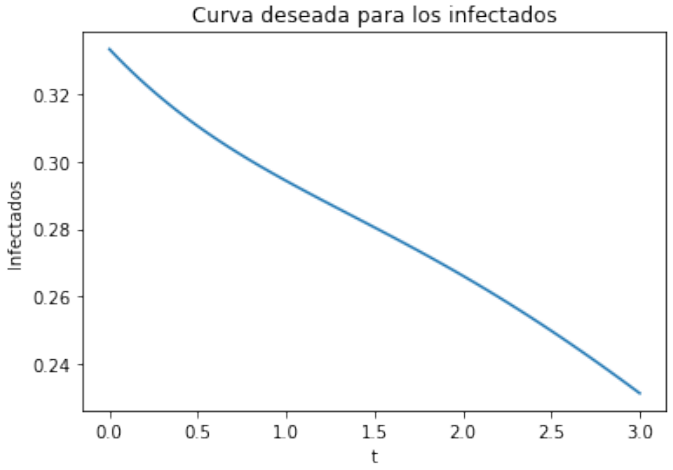


Figura 6.6: Curva deseada para los infectados.

Nuestra función de pérdida para este problema es la siguiente

$$C = \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{S}'(t_i) + \hat{\beta}(t)\hat{S}(t_i)\hat{I}(t_i)\right\|^2 + \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{I}'(t_i) - \hat{\beta}(t)\hat{S}(t_i)\hat{I}(t_i) + \gamma\hat{I}(t_i)\right\|^2$$
$$+\frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{R}'(t_i) - \gamma\hat{I}(t_i)\right\|^2 + \left\|\hat{S}(t_{min}) - S_0\right\|^2 + \left\|\hat{I}(t_{min}) - I_0\right\|^2 + \left\|\hat{R}(t_{min}) - R_0\right\|^2 +$$
$$W\left\|\hat{I}(t) - I_{deseado}(t)\right\|^2.$$

donde $I_{deseado}(t)$ es la curva de la Figura 6.6, obtenida como solución de 6.4 usando el control de la Figura 6.5.

Se ha incluido un factor W al término asociado al estado deseado que queremos obtener [25]. Este factor W nos permite adaptar y equilibrar la importancia que debe dar la red neuronal al último término. Es muy importante a la hora del entrenamiento, pues puede ocurrir que los errores no tengan la misma magnitud, y se le de mucha más importancia a uno que a otro, dando lugar a que obtengamos una mala aproximación. Para determinar este W podemos llevar a cabo un estudio previo, realizando la simulación para distintos valores. Este estudio se tiene en la Figura 6.7. El error asociado al problema directo sería el residuo y la condición de contorno, y el asociado al estado deseado es el término que está multiplicado por W, es decir, que la aproximación para los infectados se ajuste la curva deseada. De este estudio se concluye que el factor óptimo es $W = 0.1$.

Los hiperparámetros para la simulación que se han considerado óptimos son los recogidos en la Tabla 6.5. Con estos hiperparámetros la simulación obtenida se encuentra recogida en las Figuras 6.8.

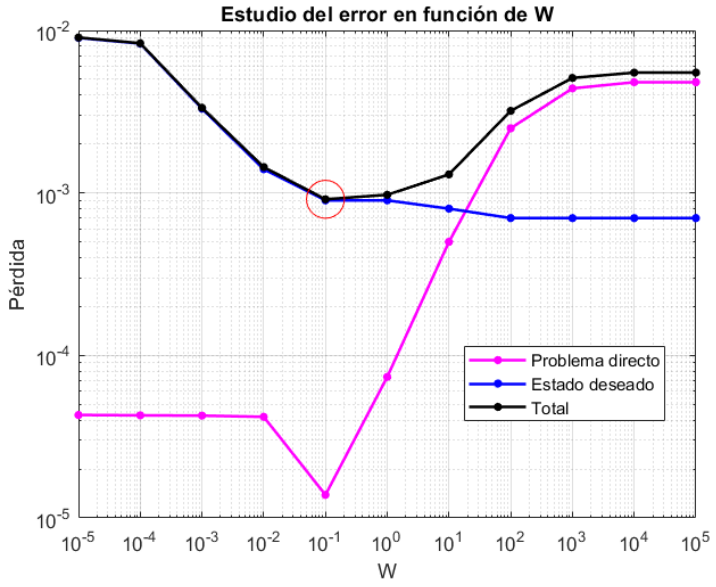
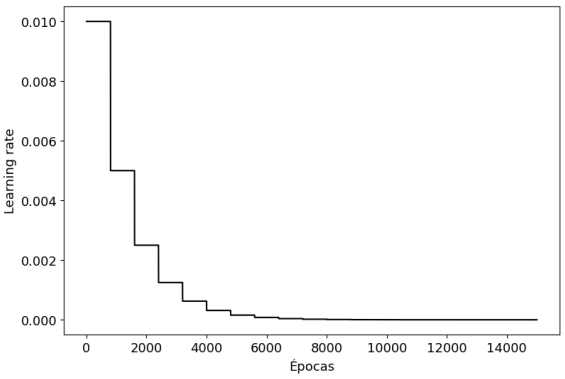


Figura 6.7: Variación del error en función del factor W.

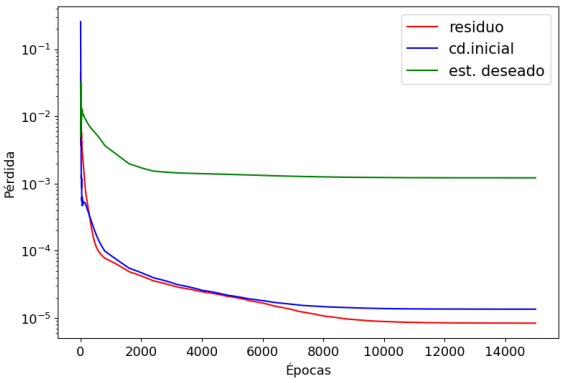
Hiperparámetros				Resultados	
Activación	Tanh	**Estructura**	$[1, 25, 25, 25, 4]$	**F. coste final**	$1 \cdot 10^{-4}$
Optimizador	Adamax	**Épocas**	15000	**Error Pb. directo**	$2.8 \cdot 10^{-5}$
Learning rate	$l_r = 0.01$, $\times 0.5/(800\text{épocas})$	**Nº puntos dom**	1000	**Error Est. deseado**	$1 \cdot 10^{-3}$
W	0.1	**Nº puntos cd. deseada**	1000	**Tiempo**	$389s$

Tabla 6.5: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (6.4).

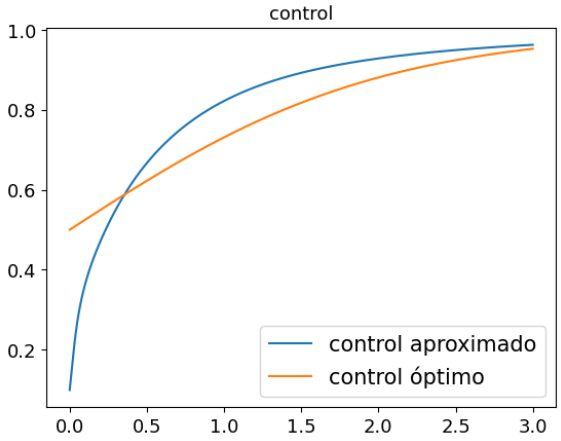
En la Figura 6.8 (d) podemos ver cómo la aproximación obtenida por la red neuronal tras el proceso de entrenamiento (en colores azul, rojo y verde) satisface la ecuación diferencial, pues se ajusta a la solución obtenida con un solucionador externo para el control obtenido por la red, Figura 6.8 (c). El control obtenido, en cambio, no se ajusta de forma tan obvia al control exacto que debería tener. Aun así, debemos tener en cuenta que el problema inverso es un problema complejo que podría tener numerosos mínimos locales, podría suceder que esta aproximación correspondiese a alguno de ellos, ya que vemos que también se ajusta al estado deseado.



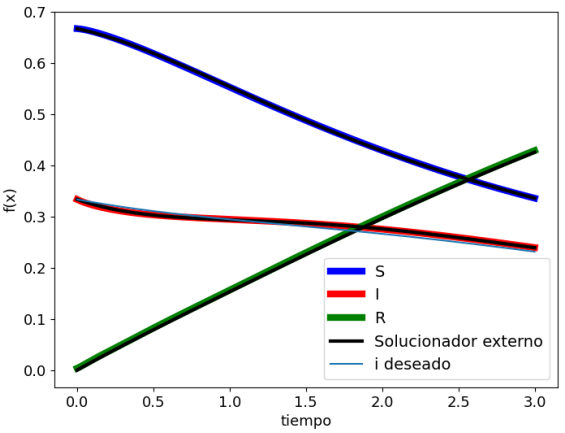
(a) «Learning rate» frente a las épocas.



(b) Evolución del error en el residuo, condición inicial y estado deseado en función de las épocas.



(c) Control exacto y aproximado.



(d) Solución aproximada por la red neuronal, solución con aproximador externo (para el control aproximado) y estado deseado.

Figura 6.8: Gráficas para la simulación con los hiperparámetros de la Tabla 6.5. Resolución del problema (6.4).

6.3.2. Ejemplo «real» de ajuste de parámetros continuos

Veamos ahora un caso que se puede ajustar más a lo que tendríamos en la vida real. En el ejemplo anterior hemos partido de una forma concreta de control, y hemos utilizado la curva de infectados, obtenida de la resolución, como curva deseada, de manera que sabíamos a priori el control óptimo que se debía obtener. En este caso, partiremos de la curva de los infectados y no conoceremos de antemano el control que debemos obtener.

En una situación de epidemia, buscamos que los infectados desaparezcan, esto podemos modelarlo, por ejemplo, con una caída lineal partiendo de la población de infectados inicial hasta llegar a cero en el tiempo final (Figura 6.9). Esta curva será la que tomaremos como curva deseada para los infectados.

Por otra parte, el parámetro $\beta(t)$ determina la tasa de crecimiento o decrecimiento

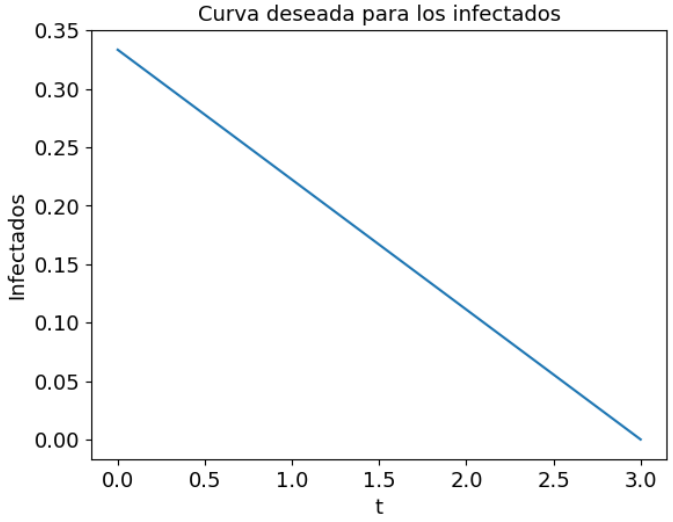


Figura 6.9: Curva deseada para los infectados.

de los susceptibles (que pasan a ser infectados), cuanto mayor sea β mayor será el decrecimiento de los susceptibles, es decir, más infecciones se producen en (6.4). Luego, podemos asociar un β más grande, a un menor número de restricciones y viceversa. Podemos añadir, por tanto, a la función de pérdida un término donde reflejemos el coste que nos supone un beta mayor. La función de pérdida es entonces:

$$C = \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{S}'(t_i) + \hat{\beta}(t)\hat{S}(t_i)\hat{I}(t_i)\right\|^2 + \frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{I}'(t_i) + \hat{\beta}(t)\hat{S}(t_i)\hat{I}(t_i) - \gamma\hat{I}(t_i)\right\|^2$$
$$+\frac{1}{N_c}\sum_{i=1}^{N_c}\left\|\hat{R}'(t_i) - \gamma\hat{I}(t_i)\right\|^2 + \left\|\hat{S}(t_{min}) - S_0\right\|^2 + \left\|\hat{I}(t_{min}) - I_0\right\|^2 + \left\|\hat{R}(t_{min}) - R_0\right\|^2 +$$
$$W\left(\frac{1}{N_d}\sum_{j=1}^{N_d}\left\|\hat{I}(t_j) - I_{deseado}(t_j)\right\|^2 - \left\|\hat{\beta}(t)\right\|^2\right).$$

Podemos realizar un estudio similar al realizado en la sección anterior para encontrar un valor óptimo del factor W. Realizando simulaciones con distintos valores con 10000 épocas de simulación, obtenemos la Figura 6.10. El valor óptimo es $W = 10^{-5}$.

Las simulaciones se realizan con los hiperparámetros de la Tabla 6.6, que se han considerado adecuados. Las gráficas obtenidas están recogidas en la Figura 6.11.

En la Figura 6.11 podemos apreciar que el control calculado difiere bastante del obtenido en la sección anterior. Para ver si la aproximación obtenida satisface el sistema diferencial, podemos resolver el sistema con el control aproximado que se obtiene, utilizando un algoritmo externo (línea negra). Gracias a estas curvas, observamos que la aproximación obtenida satisface de forma más o menos adecuada el sistema SIR. Por otra parte, los infectados evolucionan de forma que se ajustan lo máximo posible al estado final deseado (en azul claro), como tiene que cumplir más condiciones, puede no ser posible un mejor ajuste. En cuanto al control, vemos que

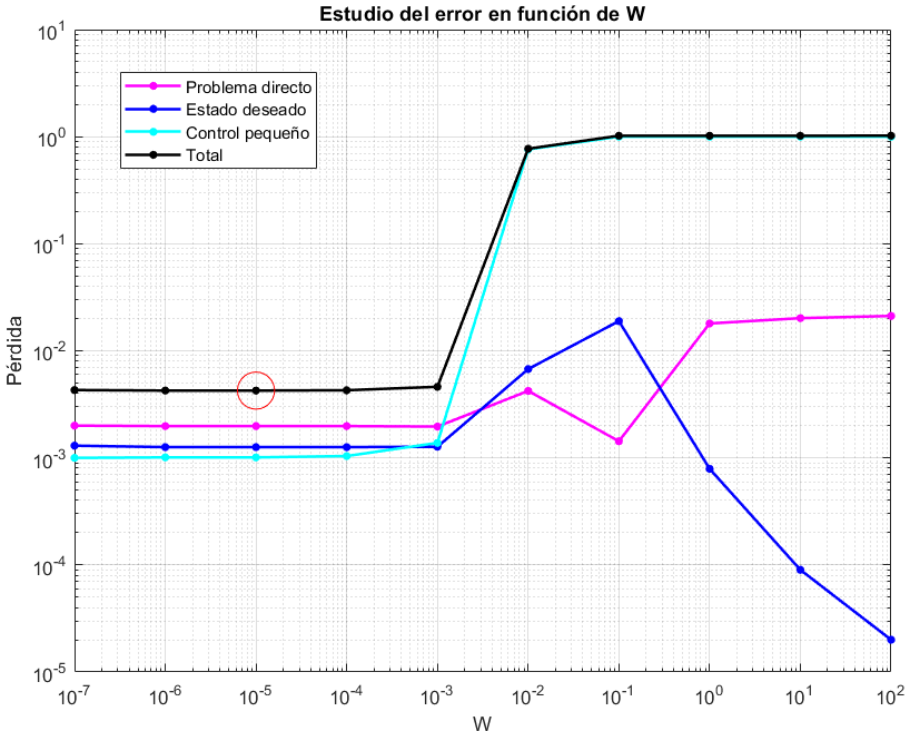
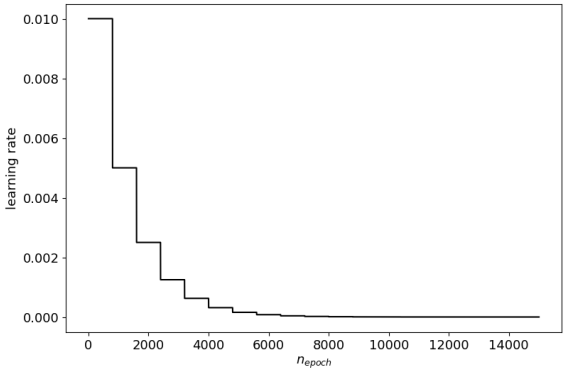


Figura 6.10: Variación del error en función del factor W.

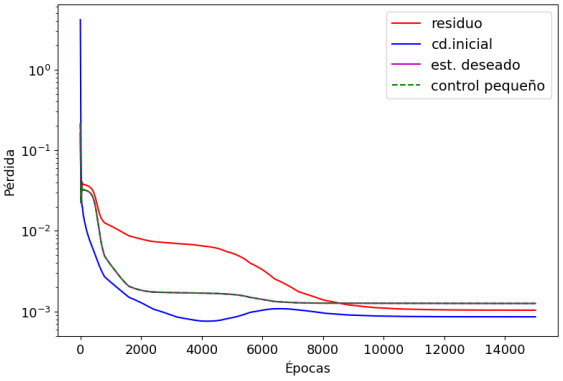
Hiperparámetros				Resultados	
Activación	Sigmoide	**Estructura**	$[1, 20, 20, 20, 4]$	**F. coste final**	$1.9 \cdot 10^{-3}$
Optimizador	Adamax	**Épocas**	15000	**Error Pb. directo**	$1.9 \cdot 10^{-3}$
Learning rate	$l_r = 0.01$, $\times 0.5/(800\text{épocas})$	**Nº puntos dom**	1000	**Error Pb. inverso**	$2.4 \cdot 10^{-3}$
W	10^{-5}	**Nº puntos cd. deseada**	1000	**Tiempo**	$222s$

Tabla 6.6: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (6.4).

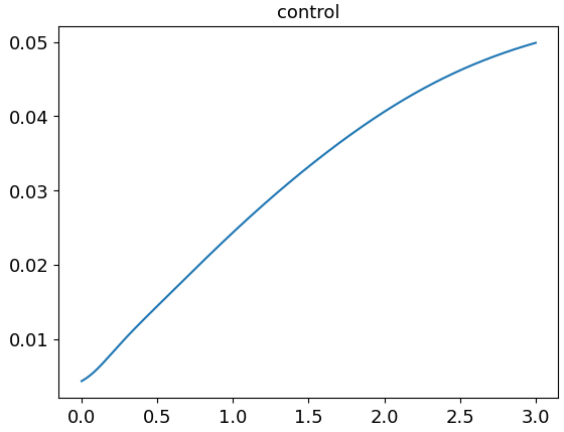
aumenta con el tiempo, lo que supondría mayor número de restricciones al comienzo de la epidemia, y menos restricciones al final, lo cual también es coherente.



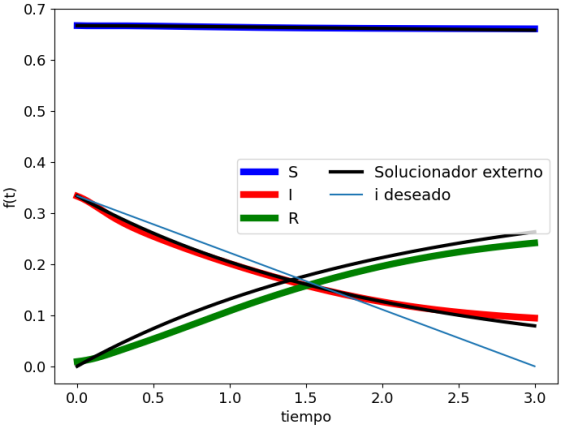
(a) Evolución del «learning rate» frente a las épocas.



(b) Evolución del error frente a las épocas.



(c) Control aproximado.



(d) Solución aproximada por la red neuronal, solución con aproximador externo (para el control aproximado) y estado deseado.

Figura 6.11: Gráficas para la simulación con los hiperparámetros de la Tabla 6.6. Resolución del problema (6.4).

Capítulo 7

Ecuaciones en derivadas parciales

Como ya se ha dicho anteriormente, las «PINN» son fácilmente adaptables a las ecuaciones en derivadas parciales. Basta adaptar la función de pérdida a la ecuación correspondiente. De forma práctica, la red neuronal no distingue entre ecuaciones diferenciales ordinarias y ecuaciones en derivadas parciales, trata ambos problemas de la misma forma, a diferencia de los métodos clásicos de resolución y aproximación de problemas diferenciales.

En este capítulo se va a resolver un problema directo relativo a una ecuación en derivadas parciales y, posteriormente, se utilizará dicha aproximación para obtener observaciones para un problema de ajuste de parámetros.

7.1. Problema directo

Consideramos una aproximación usando «PINN» de la solución de la ecuación en derivadas parciales siguiente:

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} + \operatorname{sen}(x+y)u = 0, & x, y \in [0,1], \\ u(x,1) = \operatorname{sen}(\pi x), \\ u(x,0) = u(0,y) = u(1,y) = 0. \end{cases} \tag{7.1}$$

Para poder obtener una aproximación externa con la que poder comparar la solución obtenida con la red neuronal, se utiliza el método de las diferencias finitas (MDF).

La función de pérdida de este problema sería:

$$C = \frac{1}{N_c} \sum_{i=1}^{N_c} \left\| -\hat{u}_{xx}(x_i, y_i) - \hat{u}_{yy}(x_i, y_i) + \operatorname{sen}(x_i + y_i)\hat{u}(x_i, y_i) \right\|^2 +$$

$$W \cdot \frac{1}{N_o} \sum_{k=1}^{N_o} \left(\left\| \hat{u}(x_k, 1) - \operatorname{sen}(\pi x_k) \right\|^2 + \left\| \hat{u}(x_k, 0) \right\|^2 + \left\| \hat{u}(0, y_k) \right\|^2 + \left\| \hat{u}(1, y_k) \right\|^2 \right),$$

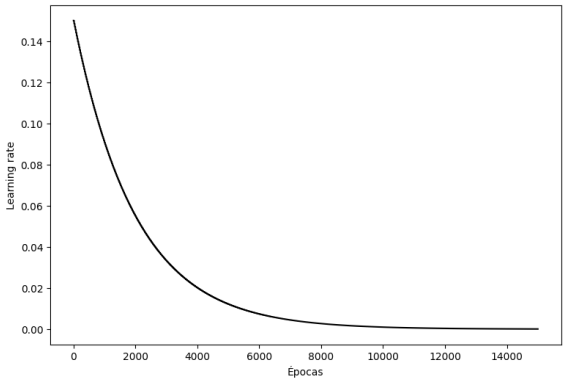
donde W es el factor que ya se introdujo en el capítulo anterior.

Podemos ver en la Tabla 7.1 los hiperparámetros que se utilizarán para realizar la simulación.

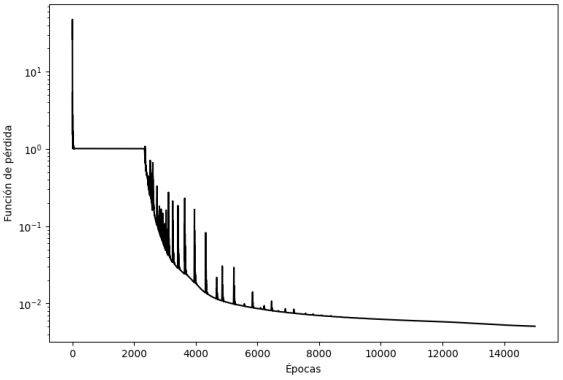
Hiperparámetros				**Resultados**	
Activación	Sigmoide	**Estructura**	$[2, 25, 25, 25, 1]$	**F. coste final**	0.0051
Optimizador	Adamax	**Épocas**	15000	**Error Residuo**	0.0046
Learning rate	$l_r = 0.15$, $\times 0.99/(20$ épocas$)$	**Nº puntos dom**	1000	**Error cd inicial**	$5 \cdot 10^{-}5$
W	10	**Nº cd inicial**	1000	**Tiempo**	$488s$

Tabla 7.1: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (7.1).

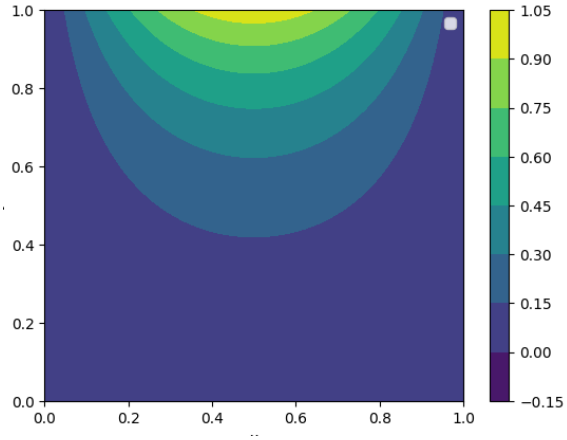
En las Figuras 7.1 y 7.2 se recogen los resultados de la simulación realizada.



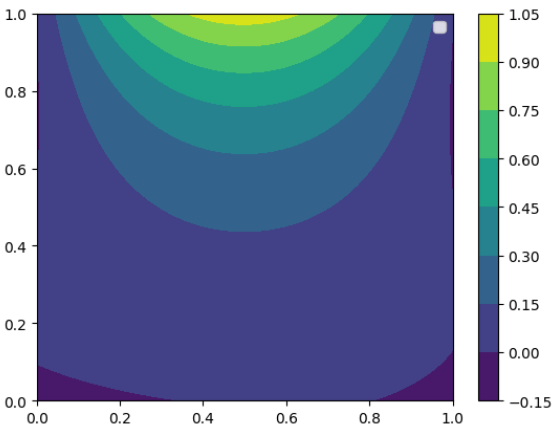
(a) Evolución del «learning rate» por épocas.



(b) Evolución del error por épocas.



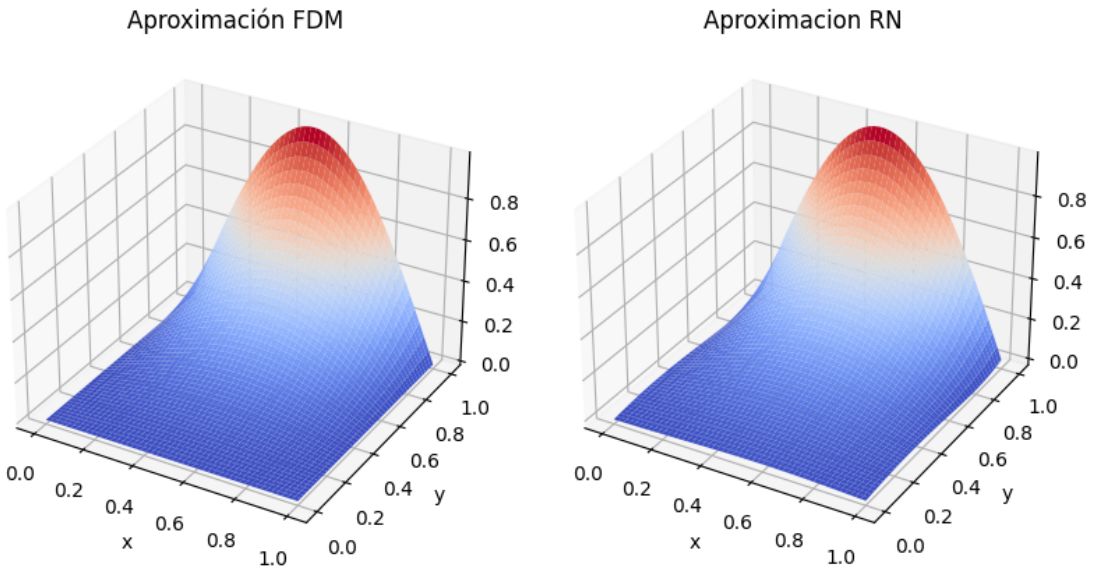
(c) Solución obtenida con el MDF.



(d) Solución obtenida con la red neuronal.

Figura 7.1: Gráficas para la simulación con los hiperparámetros de la Tabla 7.1. Resolución del problema (7.1).

Podemos observar una ligera variación en la solución obtenida, principalmente en la condición de contorno, en la Figura 7.1 (d).



(a) Solución obtenida con el MDF.

(b) Solución obtenida con la red neuronal.

Figura 7.2: Gráficas para la simulación con los hiperparámetros de la Tabla 7.1. Resolución del problema (7.1).

7.2. Ajuste de parámetros

En esta sección vamos a ajustar un parámetro continuo (función «control») en una ecuación en derivadas parciales. El parámetro que ajustaremos será la función $a(x,t)$ del siguiente problema:

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} + a(x,y)u = 0, & x, y \in [0,1], \\ u(x,1) = \operatorname{sen}(\pi x), & \\ u(x,0) = u(0,y) = u(1,y) = 0. & \end{cases} \tag{7.2}$$

Se hace notar que el problema (7.1) que se resolvió en la sección anterior, es un caso particular para $a(x,y) = \operatorname{sen}(x+y)$.

Al igual que hicimos en el capítulo anterior en la Sección 6.3, introduciremos una nueva neurona de salida a la red neuronal, de forma que una salida se corresponderá con la propia aproximación a la red $u(x,y)$ y la segunda será la aproximación de la función $a(x,y)$. Como se trata de un problema inverso, daremos a la red neuronal observaciones (u_l) de la solución del problema para una función $a(x,y)$ concreta, y entrenaremos la red con el objetivo de que el mejor ajuste sea el que buscamos. Los valores de observaciones tomados serán los de la aproximación obtenida con la red neuronal obtenida en la Sección 7.1.

La función de coste pasa a ser ahora:

$$C = \frac{1}{N_c}\sum_{i=1}^{N_c} \|-\hat{u}_{xx}(x_i,y_i) - \hat{u}_{yy}(x_i,y_i) + \hat{a}(x_i,y_i)\hat{u}(x_i,y_i)\|^2 +$$

$$W \cdot \frac{1}{N_o}\sum_{k=1}^{N_o} \left(\|\hat{u}(x_k,1) - \operatorname{sen}(\pi x_k)\|^2 \|\hat{u}(x_k,0)\|^2 + \|\hat{u}(0,y_k)\|^2 + \|\hat{u}(1,y_k)\|^2\right)$$

$$+\frac{1}{N_d}\sum_{l=1}^{N_d} \|\hat{u}(x_l,y_l) - u_l\|^2 .$$

La diferencia respecto al problema directo es el último término que se ha añadido para imponer que la red neuronal se ajuste a las N_d observaciones que se van a usar en el entrenamiento.

Los hiperparámetros con los que se realiza la simulación se han recogido en la Tabla 7.2.

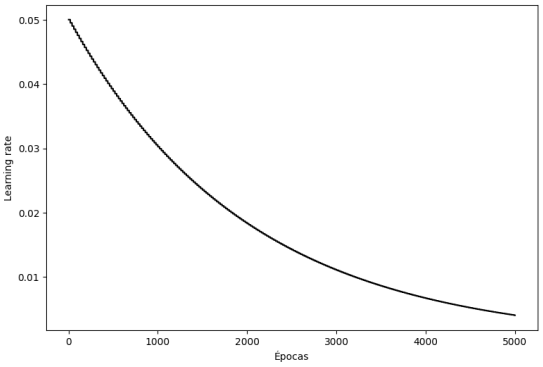
Hiperparámetros				**Resultados**	
Activación	Sigmoide	**Estructura**	$[2, 30, 30, 30, 2]$	**F. coste final**	0.11
Optimizador	Adamax	**Épocas**	5000	**Error Obs**	0.10
Learning rate	$l_r = 0.05$, $\times 0.99/(20$ épocas$)$	**Nº puntos dom**	1000	**Tiempo**	$133s$
W	10	**Nº puntos cd inicial**	1000	**Nº puntos observaciones**	50×50

Tabla 7.2: Hiperparámetros para la simulación con los valores adecuados. Resolución del problema (7.2).

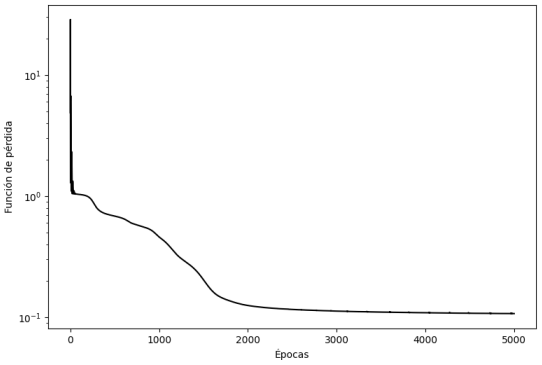
Los puntos tomados en las observaciones, son el resultado de hacer un mallado para 50 puntos en cada uno de los ejes, es decir, un total de 2500 observaciones.

Los resultados de la simulación se pueden observar en la Figura 7.3.

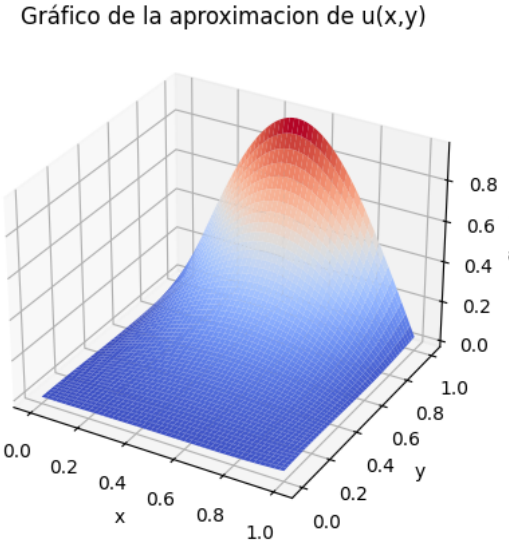
Se puede apreciar que la aproximación obtenida para la función $a(x, y)$ no es igual que la función $\operatorname{sen}(x + y)$ que esperábamos. Esto puede deberse a varios factores. En primer lugar, no podemos descartar que la red neuronal se haya «atascado» en un mínimo local de la función de coste, aunque teniendo en cuenta el error tan pequeño que se obtiene, el mínimo local debería estar próximo a uno global. Otro factor, quizás el más importante, es que el problema no tiene por qué tener solución única, ya que vemos que aunque la función no es la misma, la solución obtenida se ajuste mucho a la esperada.



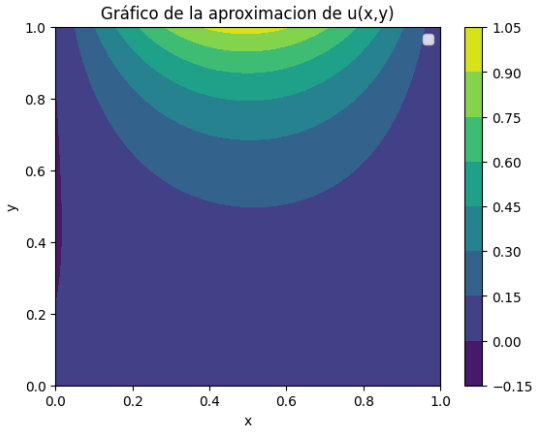
(a) Evolución del «learning rate» por épocas.



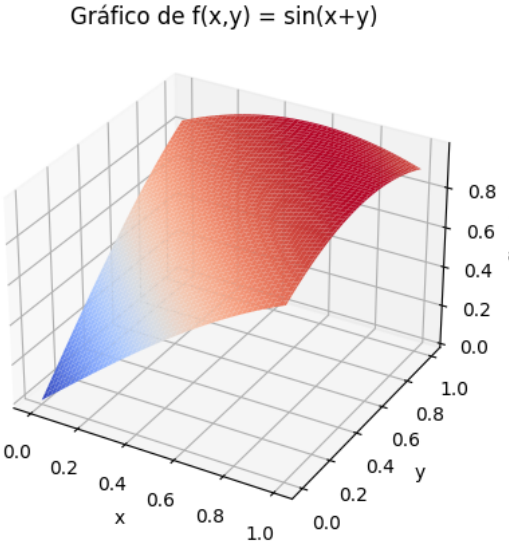
(b) Evolución del error por épocas.



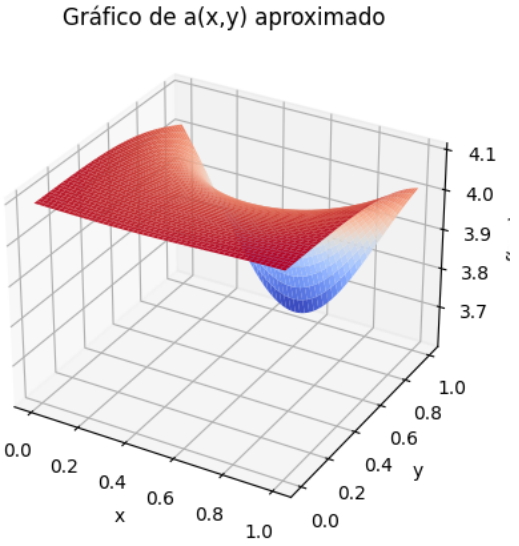
(c) Solución obtenida con la RN.



(d) Solución obtenida con la RN.



(e) Ajuste exacto sen $(x + y)$.



(f) Ajuste obtenido con la red neuronal.

Figura 7.3: Gráficas para la simulación con los hiperparámetros de la Tabla 7.2. Resolución del problema (7.2).

Capítulo 8

Herramientas informáticas y códigos

En esta sección se van a detallar los códigos utilizados para obtener las aproximaciones con redes neuronales.

Existen muchos lenguajes de programación usados en el «machine learning», uno de los más potentes y populares hoy en día es «Python» (ver [26, 27]). Python es un lenguaje de programación interpretado, de tipo dinámico y multiparadigma, con una sintaxis que favorece la legibilidad del código. Además, dispone de gran cantidad de librerías y módulos que añaden al lenguaje multitud de funciones, algoritmos y propiedades.

Dentro de Python existen 2 grandes librerías para la programación de redes neuronales: TensorFlow y Pytorch (ver [28]). Como la inteligencia artificial está en el punto de mira en la actualidad, los grandes desarrollos en este área van de la mano de grandes empresas: TensorFlow fue desarrollado en 2015 por Google y Pytorch en 2017 por Facebook.

Tanto TensorFlow como Pytorch son librerías muy potentes para la implementación de redes neuronales. Para la realización de este trabajo, empecé a programar con la primera de ellas, que resultaba más intuitiva, pero a la hora de realizar códigos más complejos, como problemas inversos, en los que se tiene que añadir parámetros a la estructura de la red neuronal, me resultó más complicado. Mientras que Pytorch, aunque al comienzo puede parecer más difícil de entender, a la hora de tratar con problemas más avanzados me parece más conveniente.

La función utilizada para obtener una aproximación con la que poder comparar la obtenida con la red neuronal es `odeint`, de la librería «SciPy» del módulo «integrate», que resuelve un sistema diferencial, dados los valores iniciales y una partición en la variable independiente.

Por otra parte, en la Sección 6.2.2 se introdujo «ruido» en las observaciones utilizadas para entrenar la red neuronal. Esto se hizo sumando a cada observación un porcentaje de sí misma con signo aleatorio, veremos el código en la Sección 8.2.

Se van a añadir a continuación ejemplos sencillos y códigos básicos, para resolver un problema directo y un problema inverso con redes neuronales. El resto de simulaciones que se han llevado a cabo a lo largo del proyecto se pueden obtener introduciendo pequeñas modificaciones en estos códigos. Por ejemplo, el paso de una ecuación diferencial a sistema de ecuaciones diferenciales resultaría natural.

8.1. Problema directo

Primero veamos el código para un problema directo, el utilizado para resolver el problema (5.4) [29, 30]:

$$\begin{cases} u'(t) = 3t, & t \in [0,5], \\ u(0) = 5. \end{cases}$$

Cargamos todas las librerías necesarias:

```
import torch
import torch.autograd as autograd
from torch import Tensor
import torch.nn as nn                       # red neuronal
import torch.optim as optim                 # optimizadores

import matplotlib.pyplot as plt
import numpy as np
import time
from pyDOE import lhs         #Latin Hypercube Sampling
import scipy.io

from scipy import integrate # para resolver ode

torch.set_default_dtype(torch.float)

#Fijamos la raíz de los números aleatorios
torch.manual_seed(1234)
np.random.seed(1234)
```

En el primer grupo de comandos cargamos los diferentes paquetes de «Pytorch» que vamos a utilizar, dándoles un nombre específico para poder llamarlos de forma fácil (por ejemplo, `torch.autograd` como `autograd`, etc). El siguiente párrafo de comandos sirve para cargar las librerías que usaremos para realizar las gráficas o para resolver de forma aproximada (solucionador externo) la ecuación diferencial. Por último, se fija la raíz de los números aleatorios para que cuando se repita alguna simulación se parta de las mismas condiciones.

Definimos ahora, algunos parámetros concretos del problema:

```
steps=1000 #épocas de la simulación

A=3 # y'(t)=A*t

y0=5 # valor inicial y(0)

lr=0.01 # learning rate inicial

layers = np.array([1,10,10,1]) #estructura de la red

min=0 # intervalo de resolución [min,max]
max=5

Nf=1000 #puntos para evaluar el residuo
```

Se han definido los parámetros relacionados con la ecuación diferencial y el dominio de resolución. Además, se especifica la estructura de la red en la notación que se ha utilizado durante el trabajo ($[1, 10, 10, 1]$). Se decide también el número de puntos en los que se evaluará el residuo durante el entrenamiento.

El siguiente paso es definir la propia red neuronal junto con la función de pérdida.

```
class FCN(nn.Module):
    def __init__(self,layers):
        super().__init__() #call __init__ from parent class
        'activation function'
        self.activation = nn.Sigmoid()
        'loss function'
        self.loss_function = nn.MSELoss(reduction ='mean') #
        ↪  seleccionamos el tipo de función de pérdida
        'Initialise neural network as a list using nn.Modulelist'
        self.linears = nn.ModuleList([nn.Linear(layers[i],
        ↪  layers[i+1]) for i in range(len(layers)-1)])
        self.iter = 0
        'Xavier Normal Initialization'
        for i in range(len(layers)-1):
            nn.init.xavier_normal_(self.linears[i].weight.data,
            ↪  gain=1.0)
            nn.init.zeros_(self.linears[i].bias.data)

    'foward pass' # programamos el paso hacia delante
    def forward(self,x):
        if torch.is_tensor(x) != True:
            x = torch.from_numpy(x)
        a = x.float()
        for i in range(len(layers)-2):
            z = self.linears[i](a)
            a = self.activation(z)
        a = self.linears[-1](a)
```

```
25          return a
26
27      'Loss Functions' # se programa la función de pérdida
28      def lossBC(self): # pérdida en la cd. inicial
29        loss_BC= self.loss_function(self.forward(min*torch.ones(1,1)),\
30        y0*torch.ones(1,1))
31        return loss_BC
32
33      def lossPDE(self,x_PDE): # pérdida en la ecuacion diferencial
34        g=x_PDE.clone() # variable independiente
35        g.requires_grad=True #para poder derivarla
36        f=self.forward(g) # f es la salida de la red
37        # f_x es la derivada de f respecto de x
38        f_x=autograd.grad(f,g,\
39        torch.ones([x_PDE.shape[0],1]).to(device),retain_graph=True,
          ↪  create_graph=True)[0]
40        loss_PDE=self.loss_function(f_x,A*g)
41        return loss_PDE
42
43      def loss(self,x_PDE): # la función de pérdida es la suma de ambas
44        loss_bc=self.lossBC()
45        loss_pde=self.lossPDE(x_PDE)
46        return loss_bc+loss_pde
```

En la primera parte `def __init__` define cual va a ser la función de activación, en este caso se ha elegido la sigmoide. A continuación, se determina el tipo de función de pérdida, es decir, si queremos dividir por el número de elementos, o simplemente sumarlos. Generalmente se trabaja con «mean», es decir, dividiendo por el número total de valores. Se definen entonces las capas con formato lista y se determina la inicialización de los parámetros internos.

Lo siguiente es programar cómo se va a realizar el paso hacia delante («forward propagation»), en este apartado es donde se especifica si se va a aplicar la función de activación a la última capa, o únicamente a las capas intermedias, en este caso no queremos imponer ninguna condición extra a la salida, luego se aplica a las neuronas de las capas intermedias.

Por último, debemos especificar la función de pérdida, se hará en dos partes, una primera para la condición inicial (`lossBC`), y una segunda para el residuo de la ecuación diferencial (`lossPDE`). Utilizando la función `forward` podemos evaluar la red neuronal en valores concretos. Para la condición inicial, imponemos que la red en el valor `min` sea igual a y_0. Para ello se usa la función `loss_function`, con la que llamamos al tipo de función que se ha establecido anteriormente, esta función tiene dos argumentos de entrada, lo que hace es la media al cuadrado de la diferencia entre ambos. Por ejemplo `loss_function(`$\hat{y}$`(min),y0)` donde $\hat{y}$ sería la función de la red, impondría que la red en el valor inicial sea igual al valor que queremos imponerle, eso es precisamente lo que se ha hecho.

En el caso del residuo de la red es algo más complejo, ya que debemos eva-

luar la derivada de la función de la red neuronal, para ello debemos llamar a `autograd.grad(f,g,...)` que hace la derivada (mediante el algoritmo de «back-propagation» de la función f evaluada en los puntos g, que son los puntos en los que se va a evaluar el residuo, que especificaremos posteriormente. La derivada se realiza respecto de todos los parámetros de los que dependa, si tenemos $f(x, y)$ nos devolverá dos columnas, en cada una de ellas la derivada respecto a cada uno de los parámetros, en este caso solo tenemos una neurona de entrada, luego devuelve un único vector correspondiente a la derivada. Utilizamos de nuevo la función `loss_function`, esta vez para imponer que $dy/dx = Ax$. La función de perdida total (`loss`), será la suma de ambas partes.

Una vez tenemos definida la estructura de la red neuronal, vamos a definir los puntos en los que vamos a evaluar el residuo (`x_PDE`) y el resto de hiperparámetros relacionados con el proceso de entrenamiento.

```
x_PDE = torch.Tensor([min]) +
↪  (torch.Tensor([max])-torch.Tensor([min]))*lhs(1,Nf)
torch.manual_seed(123)
x_PDE=x_PDE.float().to(device)
#Creamos el modelo
model = FCN(layers)
model.to(device)
params = list(model.parameters())
optimizer = torch.optim.Adamax(model.parameters(),lr=lr) #elegimos
↪  optimizador y paso inicial
start_time = time.time()
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2000,
↪  gamma=0.5) #para modificar el learning rate
```

Los puntos que se escogen son de un muestreo de hipercubo latino (ver Sección 5.1) en el intervalo de resolución. Seguidamente se carga el modelo. Además, es aquí donde se especifica el optimizador y el paso variable. Como optimizador se ha elegido el «Adamax», especificando como paso el `lr`, que será el inicial. Con la opción `scheduler` se impone el paso variable, en este caso se reduce a la mitad cada 2000 épocas.

Por último, comenzamos con el entrenamiento.

```
# ENTRENAMIENTO
from time import time

t0=time()
hist=torch.Tensor([]) #para almacenar el error en cada etapa
lrate=torch.Tensor([]) #para el learning rate

for i in range(steps):

    loss = model.loss(x_PDE)
```

```
    optimizer.zero_grad()
    loss.backward()

    optimizer.step() #avanzamos en el optimizador
    scheduler.step() #actualizamos el learning rate
    hist=torch.cat((hist,torch.Tensor([loss])),0)
    lratenew=optimizer.param_groups[0]['lr']
    lrate=torch.cat((lrate,torch.Tensor([lratenew])),0)

    if i%(steps/10)==0: #imprimimos 10 actualizaciones
      print('It {:05d}: loss = {:10.8e}'.format(i,loss))

print('\nComputation time: {} seconds'.format(time()-t0)) #vemos el
↪  tiempo total
```

Creamos los vectores donde almacenaremos la evolución de los diferentes datos en el proceso de entrenamiento: el error y el «learning rate». Acto seguido se programa el bucle, que se realizará tantas veces como épocas hayamos especificado. En cada etapa del bucle se da un paso del optimizador, se actualizan los parámetros y se almacenan los que nos interesen.

Tras este proceso, ya tendríamos entrenada la red neuronal, solo queda representar la aproximación obtenida y estudiar la variación de los distintos hiperparámetros a lo largo del entrenamiento.

```
#Gráficas para el learning rate.
fig = plt.figure(figsize=(9,6))
plt.plot(torch.range(1,steps), lrate,'k-')
plt.xlabel('Épocas')
plt.ylabel('Learning rate');

# Gráficas para el error.
fig = plt.figure(figsize=(9,6))
ax = fig.add_subplot(111)
ax.semilogy(torch.range(1,steps), hist,'k-')
ax.set_xlabel('Épocas')
ax.set_ylabel('Función de pérdida');

# Evaluamos la aproximación obtenida
x = torch.linspace(min,max,Nf).view(-1,1)
yh=model(x.to(device)) #cargamos la aproximación
yh_plot= yh.detach().cpu().numpy()

#  Usamos un aproximador externo para comparar (en este caso también
↪  se puede usar la exacta, la conocemos).
tspace = np.linspace(min, max,10000)
def aux(u, x):
    return A*x
```

```
23
24 U = integrate.odeint(aux, y0, tspace)
25 # Graficamos la aproximación y la solución exacta
26 fig, ax1 = plt.subplots()
27 ax1.plot(x,yh_plot,color='blue',label='PINN',linewidth=6.0)
28 ax1.plot (tspace,U, color='orange', label='exacta',linewidth=3.0)
29 ax1.set_xlabel('t',color='black')
30 ax1.set_ylabel('u(t)',color='black')
31 ax1.legend(loc = 'upper left')
```

Y con estos códigos obtendríamos gráficas para la evolución del *learning rate* y de la función de pérdida en función de la época. Además, también tenemos la representación de la aproximación obtenida por la red neuronal, el aproximador externo en este caso es simplemente integrar la función, dada la simplicidad de la ecuación diferencial.

8.2. Ajuste de parámetros discretos

Para estos problemas necesitaremos añadir de forma manual un parámetro nuevo a los ya existentes en la red neuronal, que se ajustará junto con el resto. La mayor parte del código es análoga al de la sección anterior, se puntualizarán, por tanto, las diferencias.

```
1  # Definimos los parámetros necesarios para la resolución
2
3  steps=5000 # número de iteraciones
4  ruido=0 # % de ruido para las observaciones
5
6  A=-2.0 # valor exacto para el parametro lambda
7  y0=2 #valor de la condicion inicial
8  min=0 # Dominio de resolucion
9  max=5
10
11 lambda1= 3.0  # inicializacion del parámetro
12
13 lr=0.06 # learning rate inicial
14 layers = np.array([1,10,10,10,1]) # estructura de la red
15
16 Ndom= 1000 # puntos para evaluar las pérdidas
17 Nobs=50 # puntos de observaciones
```

Se han definido los valores concretos del problema. Se define además el valor inicial para el parámetro que se quiere ajustar `lambda1`.

Debemos ahora definir el funcionamiento de la red neuronal, al igual que en la sección anterior. En este momento es cuando añadimos de forma manual un parámetro extra a la red.

```
class FCN(nn.Module):
    ##Neural Network
    def __init__(self,layers):
        super().__init__() #call __init__ from parent class

        'Función de activación'
        self.activation = nn.Sigmoid()

        'loss function'# tipo de función de pérdida
        self.loss_function = nn.MSELoss(reduction ='mean')

        'Initialise neural network as a list using nn.Modulelist'
        self.linears = nn.ModuleList([nn.Linear(layers[i],
        ↪  layers[i+1]) for i in range(len(layers)-1)])
        self.iter = 0

        'Inicializamos el parámetro (Inverse problem)'
        self.lambda1 = torch.tensor([lambda1],
        ↪  requires_grad=True).float().to(device)

        'Registramos para el entrenamiento'
        self.lambda1 = nn.Parameter(self.lambda1)

        'Xavier Normal Initialization'
        for i in range(len(layers)-1):
            nn.init.xavier_normal_(self.linears[i].weight.data,
            ↪  gain=1.0)
            nn.init.zeros_(self.linears[i].bias.data)

    'foward pass' # programamos el paso hacia delante
    def forward(self,x):
        if torch.is_tensor(x) != True:
            x = torch.from_numpy(x)
        a = x.float()
        for i in range(len(layers)-2):
            z = self.linears[i](a)
            a = self.activation(z)
        a = self.linears[-1](a)
        return a

    'Loss Functions' # se programa la función de pérdida
    def loss_BC(self): # pérdida en la cd inicial
      loss_i= self.loss_function(self.forward(min*torch.ones(1,1)),\
      y0*torch.ones(1,1))
      return loss_i

    def loss_PDE(self,x_PDE): # pérdida en la ecuacion diferencial
      lambda1=self.lambda1 # nuestro parametro a optimizar
```

```
      g=x_PDE.clone() # la g es la var. independiente (x o t)
      g.requires_grad=True #Enable differentiation
      f=self.forward(g) # f es la salida de la red
      f_x=autograd.grad(f,g,torch.ones([x_PDE.shape[0],1]).\
      to(device),retain_graph=True, create_graph=True)[0]
      loss_f=self.loss_function(f_x,(lambda1)*g)
      return loss_f

    def loss_data(self,x_obs,y_obs): # pérdida en las observaciones
        loss_obs = self.loss_function(self.forward(x_obs), y_obs)
        return loss_obs

    def loss(self,x_PDE,x_obs,y_obs): # pérdida total
      loss_i=self.loss_BC()
      loss_f=self.loss_PDE(x_PDE)
      loss_obs=self.loss_data(x_obs,y_obs)
      return  loss_i + loss_f + loss_obs
```

La diferencia con el problema directo es el parámetro extra que hemos añadido, el resto de partes, como la definición del paso hacia delante y la función de pérdida, se hacen de la misma forma. Como para el problema de ajuste de parámetros se necesitan observaciones del problema, también debemos introducirlas: `x_obs` e `y_obs`.

Solo queda definir las observaciones y tiempos de entrenamiento que introduciremos en la red.

```
# Generamos los datos
x_obs=torch.linspace(min,max,Nobs).view(-1,1)
y_obs= (A/2)*(x_obs**2)+y0  #observaciones

# Introducimos ruido
for i in range(Nobs):
  y_obs[i]=y_obs[i]+random.randint(-1,1)*ruido*y_obs[i]
y_obs=y_obs.view(-1,1)

# Cargamos los datos
x_PDE=torch.linspace(min,max,Ndom).view(-1,1)
x_PDE=x_PDE.float().to(device)
x_obs=x_obs.float().to(device)
y_obs=y_obs.float().to(device)

# Cargamos el modelo y elegimos optimizador
model = FCN(layers)
model.to(device)
params = list(model.parameters())
optimizer = torch.optim.Adam(model.parameters(),lr=lr,amsgrad=False)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=200,
↪  gamma=0.9)
```

Las observaciones introducidas se han generado con el valor real del parámetro que buscamos, ya que en este ejemplo «educativo» sabemos con anterioridad lo que debemos obtener, en una situación real estos serán datos experimentales. Además, se programa la introducción de ruido en los datos.

Al igual que en el apartado anterior, programamos el entrenamiento y almacenamos las variables para las que queramos ver la evolución.

```
# Programamos el entrenamiento
from time import time
start_time = time()
t0=time()
hist=torch.Tensor([])
par=torch.Tensor([])
lrate=torch.Tensor([])

for i in range(steps):
    loss      = model.loss(x_PDE,x_obs,y_obs)
    nuevovalor=model.lambda1
    optimizer.zero_grad()

    loss.backward()
    optimizer.step()
    scheduler.step()

    par=torch.cat((par,torch.Tensor([nuevovalor])),0)
    hist=torch.cat((hist,torch.Tensor([loss])),0)
    lrate=torch.cat((lrate,torch.\
    Tensor([optimizer.param_groups[0]['lr']])),0)
    if i%(steps/10)==0:
      print('It {:05d}: loss = {:10.8e}'.format(i,loss))
print('\nComputation time: {} seconds'.format(time()-t0))
```

En este caso se va a almacenar también la evolución del parámetro a ajustar.

A partir de este punto, se deben representar la aproximación obtenida con la red neuronal y la evolución de los parámetros, de igual forma que en la sección anterior.

8.3. Ajuste de parámetros continuos

Estos problemas se pueden programar partiendo del problema directo, añadiendo simplemente una nueva neurona en la capa de salida. Esta nueva salida sería la función de «control» que buscamos obtener. Las condiciones extra que van a determinar esta función se añaden en la función de pérdida.

Bibliografía

[1] OpenAI, *ChatGPT*, 2022, https://openai.com/blog/chatgpt.

[2] Claudio Gutiérrez, Andrés Abeliuk, *Historia y evolución de la inteligencia artificial*, *Universidad de Chile, Revista BITS*, 2021, https://revistasdex.uchile.cl/index.php/bits/article/download/2767/2700/10150.

[3] Varol Akman, Patrick Blackburn, *Editorial: Alan Turing and Artificial Intelligence*, *Journal of Logic, Language, and Information*, 2000, https://www.researchgate.net/figure/Turings-1950-paper-is-one-of-the-most-cited-in-philosophical-AI-literature-Reproduced_fig2_28762974.

[4] Joseph Weizenbaum, *ELIZA-A Computer program for the study of natural lenguage communication between man and machine*, *Communications of the ACM*, 1966, https://dl.acm.org/doi/10.1145/365153.365168

[5] Ian Goodfellow and Yoshua Bengio and Aaron Courville, *Deep Learning*, *MIT Press*, 2016, http://www.deeplearningbook.org.

[6] Anthony L. Caterini and Dong Eui Chang, *Deep Neural Networks in a Mathematical Framework*, *Springer International Publishing*, 2018, http://link.springer.com/10.1007/978-3-319-75304-1.

[7] G Cybenkot, *Approximation by Superpositions of a Sigmoidal Function*, *Math. Control Signals Systems*, 1989.

[8] Haim Brézis, *Análisis funcional. Teoría y aplicaciones*, 84-206-8088-5, *Alianza Editorial*, 1983.

[9] Robert B Ash, *Real Analysis and Probability*.

[10] Walter Rudin, *Real and complex analysis*, 0-07-100276-6, *McGRAW-HILL internacional editions*, 1987.

[11] José Luis Sarmiento-Ramos, *Aplicaciones de las redes neuronales y el deep learning a la ingeniería biomédica*, 10.18273/revuin.v19n4-2020001, 16574583, *Revista UIS Ingenierías*, 2020.

[12] Jordi Torres, *PYTHON DEEP LEARNING. Introducción práctica con Keras y TensorLow 2*, 978-607-538-614-0, *Alfaomega*, 2020.

[13] James Dellinger, *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*, *Towards Data Science*, 2019, https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79.

[14] Luis Velasco, *Optimizadores en redes neuronales profundas: un enfoque práctico*, 2020, https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-enfoque-pr%C3%A1ctico-819b39a3eb5.

[15] Jesús Martinez, *¿Qué es un optimizador y para qué se usa en Deep Learning?*, *DATASMARTS*, 2020, https://datasmarts.net/es/que-es-un-optimizador-y-para-que-se-usa-en-deep-learning/.

[16] B.B. Hammel, *What learning rate should I use?*, 2019, http://www.bdhammel.com/learning-rates/.

[17] Vaibhav Haswani, *Learning Rate Decay and methods in Deep Learning*, 2020, https://medium.com/analytics-vidhya/learning-rate-decay-and-methods-in-deep-learning-2cee564f910b#:~:text=Learning%20rate%20decay%20is%20a,help%20both%20optimization%20and%20generalization..

[18] Shree Nayar, *Lecture series 'First Principles of Computer Vision',Computer Science Department, School of Engineering and Applied Sciences, Columbia University*, 2021, https://www.youtube.com/watch?v=sIX_9n-1UbM.

[19] Michael A. Nielsen, *Neural Networks and Deep Learning*, *Determination Press*, 2015, http://neuralnetworksanddeeplearning.com/.

[20] Davis E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, *Learning representations by back-propagating errors Letters to nature*, 1986, https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.

[21] Yu.L. Levitan, N.I. Markovich, S.G. Rozin, I.M. Sobol, *On quasirandom sequences for numerical computations*, *Pergamon Press*, 1988.

[22] Wei-Liem Loh, *ON LATIN HYPERCUBE SAMPLING The Annals of Statistics*, 1996.

[23] M. Raissi, P. Perdikaris, G.E. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, ISSN:0021-9991, *ELSEVIER: Journal of Computational Physics*, 2091, https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[24] Howard Weiss, *The SIR model and the Foundations of Public Health*, ISSN: 1887-1097, *Materials Matemàtics*, 2013, https://mat.uab.cat/web/matmat/es/v2013n03/.

[25] Saviz Mowlavi,Saleh Nabi, *Optimal control of PDEs using physics-informed neural networks*, ISSN:0021-9991, *Journal of Computational Physics*, 2023, https://www.sciencedirect.com/science/article/pii/S002199912200794X.

[26] Python Software Foundation, *PYTHON*, https://www.python.org/.

[27] Eleonora Fanouraki, *State of the Developer Nation 23rd edition: the fall of web frameworks, coding languages, blockchain, and more!*, *Towards Data Science*, 2022, https://www.slashdata.co/blog/state-of-the-developer-nation-23rd-edition-the-fall-of-web-frameworks-coding-languages-blockchain-and-more?.

[28] John Terra, *Keras vs Tensorflow vs Pytorch: Key Differences Among Deep Learning*, 2023, https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article#what_is_pytorch.

[29] Juan Diego Toscano, *Repositorio GitHub*, 2023, https://github.com/jdtoscano94/Learning-Python-Physics-Informed-Machine-Learning-PINNs-DeepONets.

[30] Raissi, M., Perdikaris, P., Karniadakis, G. E., *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, ISSN=0021-9991, *Journal of Computational Physics*, 2019, https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[31] PyTorch Contributors, *TORCH.OPTIM*, https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer.