Redes neuronales y aprendizaje por transferencia

Universidad Complutense de Madrid

Facultad de Ciencias Matemáticas

Trabajo de Fin de Grado

Curso: 2022 - 2023

Grado en Matemáticas

Autor

Francisco Javier Abollado Múgica

Tutor

Antonio López Montes

Resumen

Este trabajo se estructura en dos secciones principales que abordan los fundamentos y aplicaciones contemporáneas de las redes neuronales. La primera parte, **Introducción a las redes neuronales**, proporciona un panorama exhaustivo sobre el desarrollo inicial y las estructuras de las redes neuronales, incluyendo el perceptrón, el Perceptrón Multicapa (MLP), las Redes Neuronales Convolucionales (CNN) y las Redes Neuronales Recurrentes (RNN). Se detallan además los componentes clave como las funciones de activación, las funciones de pérdida y los optimizadores. Esta sección culmina con ejemplos prácticos de implementación de redes neuronales en Python, consolidando así la base teórica con la aplicación práctica.

La segunda parte del trabajo se enfoca en **Aprendizaje por transferencia**, una metodología avanzada y eficiente para la aplicación de redes neuronales en problemas de la vida real. Aquí, se discuten los beneficios significativos del aprendizaje por transferencia, explicando cómo se puede mejorar el rendimiento de un modelo mediante la reutilización de conocimientos preaprendidos de conjuntos de datos extensos como ImageNet. Se presentan ejemplos prácticos en python de cómo esta técnica se aplica en modelos avanzados, como Distilbert, para ilustrar su efectividad y versatilidad en diversas aplicaciones. Este segmento destaca la transición de la teoría a la práctica, enfatizando la relevancia y la potencialidad del aprendizaje por transferencia en el ámbito de la inteligencia artificial.

Palabras clave

Redes neuronales, aprendizaje automático, clasificación, aprendizaje por transferencia, ImageNet, DistilBERT, python

Abstract

This pdf is structured into two main sections that explore the foundations and contemporary applications of neural networks. The first part, **Introduction to Neural Networks**, provides a comprehensive overview of the initial development and structures of neural networks, including the perceptron, Multi-Layer Perceptron (MLP), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). It also details key components such as activation functions, loss functions, and optimizers. This section concludes with practical examples of implementing neural networks in Python, thereby solidifying the theoretical foundation with practical application.

The second part of the thesis focuses on **Transfer Learning**, an advanced and efficient methodology for applying neural networks to real-world problems. Here, the significant benefits of transfer learning are discussed, explaining how model performance can be enhanced by reusing knowledge pre-learned from extensive datasets like ImageNet. Practical examples in python of how this technique is applied in advanced models, such as Distilbert, are presented to illustrate its effectiveness and versatility across different applications. This segment highlights the transition from theory to practice, emphasizing the relevance and potential of transfer learning in the field of artificial intelligence.

Keywords

Neural networks, machine learning, classification, transfer learning, ImageNet, DistilBERT, python

Índice

Índice de figuras

Parte I

PARTE 1: Introducción a las redes neuronales.

Capítulo 1

Red neuronal básica. El perceptrón

1.1. Neurona artificial

Una red neuronal es un modelo matemático que emula el modo en que el cerebro humano procesa la información. La estructura de la red neuronal está formada por pequeños procesadores de información, llamados neuronas artificiales, basándose en el modelo de neuronas del cerebro humano desarrollado en sus inicios por Ramon y Cajal. De forma breve las similitudes entre la neurona humana y la articial son las siguientes.

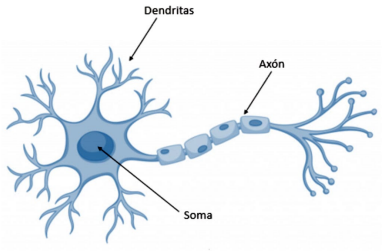


Figura 1.1.1: Neurona Humana

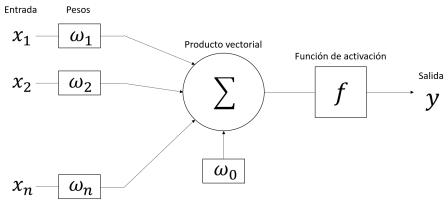


Figura 1.1.2: Neurona Artificial

Aspecto	**Neurona Humana**	**Neurona Artificial**
Canal de entrada de información	Las dendritas	Pesos de la neurona, denotados como $\vec{w} = (w_0, w_1, \ldots, w_n) \in \mathbb{R}^{n+1}$. La información de entrada es un vector $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$.

Órgano de cómputo	El soma	Producto vectorial entre $\vec{w}$ y $\vec{x}$. Este resultado es procesado por una función de activación $f : \mathbb{R} \to \mathbb{R}$.
Canal de salida	El axón	El resultado del cómputo es guardado en la variable y.

Cuadro 1.1: Comparación entre neuronas humanas y artificiales

La estructura de la red neuronal se divide en tres capas principales: la capa de entrada (input layer), una o más capas intermedias (hidden layers) y la capa de salida (output layer). En esta estrutura, las funciones de activación son una parte importante y se explicarán en [sección 3.1]. Su elección depende principalmente de dos factores: su derivavilidad y su coste computacional.

1.2. Arquitecturas previas al perceptrón

Previo al desarrollo de arquitecturas matemáticas más avanzadas como el perceptrón, encontramos el concepto de *clustering* [E], una técnica que parece compartir similitudes en términos de estructura y objetivo de resolución. El clustering es un enfoque de clasificación simple basado en la agrupación de datos en función de sus distancias, utilizando, por ejemplo, la distancia euclidiana.

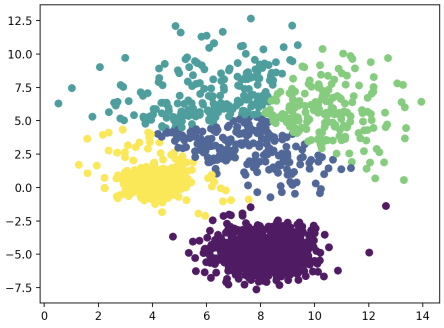


Figura 1.2.1: Clustering en 5 grupos

Clustering y la distancia euclidiana

El clustering se basa en la idea fundamental de *distancia*, un concepto matemático que tiene sus orígenes en la geometría euclidiana. Mediante la minimización de estas distancias, el clustering agrupa datos que están cercanos entre sí, proporcionando una forma básica de clasificación. Sin embargo, su utilidad está limitada a problemas lineales y de bajas dimensiones debido a su dependencia exclusiva del concepto de distancia.

Limitaciones del clustering y la necesidad de estructuras complejas

Aunque el clustering es eficaz para tareas simples de clasificación, su capacidad se ve restringida cuando se enfrenta a problemas más complejos y de alta dimensionalidad. La falta de una estructura subyacente y de mecanismos de adaptación impide que el

clustering aborde adecuadamente estos desafíos. Esto resalta la necesidad de modelos más sofisticados que puedan manejar la no linealidad y la mayor complejidad de los datos.

1.3. Historia del perceptrón

El perceptrón, una pieza clave en la historia de las redes neuronales, fue creado por Frank Rosenblatt en enero de 1957 en el Cornell Aeronautical Laboratory, ubicado en Buffalo, Nueva York. Rosenblatt y su equipo se propusieron crear una máquina capaz de aprender y reconocer patrones complejos, una iniciativa pionera que buscaba imitar funciones humanas avanzadas como la percepción visual, la formación de conceptos y la capacidad de generalizar a partir de experiencias previas.

El modelo básico del perceptrón es un clasificador lineal, lo que significa que solo puede trazar una línea de decisión para separar dos clases de datos. Funciona bien cuando los conjuntos de datos son linealmente separables, pero sus limitaciones se hacen evidentes con patrones más complejos que no se pueden separar simplemente con una línea.

Esta limitación fue destacada por Marvin Minsky y Seymour Papert en 1969 en [13], donde analizaron en profundidad las capacidades y las limitaciones del perceptrón, demostrando matemáticamente que no podía resolver problemas no lineales. Esto llevó a una reducción en el interés y la financiación de la investigación en redes neuronales hasta que en los 80 se introdujo las arquitecturas multicapa y el algoritmo de retropropagación. Estos avances revitalizaron el campo, permitiendo que las redes neuronales abordaran una gama más amplia y compleja de problemas de clasificación y regresión.

1.4. Concepto de entrenamiento de un red neuronal

Cuando se entrena una red neuronal, el objetivo principal es reducir al mínimo el error. Este error [sección 3.2] es la diferencia entre lo que la red predice y lo que realmente debería predecir según los datos de entrenamiento. Para entenderlo mejor, pensemos en la red neuronal como una herramienta que intenta aprender de ejemplos específicos.

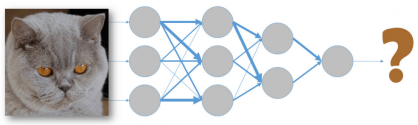


Figura 1.4.1: Entrenamiento de una red

Imagina que cada ejemplo de entrenamiento es un par de datos, representado como $(\vec{x}, y)$. Aquí, $\vec{x}$ es un conjunto de características de entrada (como números o valores que describen algo específico), y y es el resultado correcto o esperado para esas características. Por ejemplo, en una tarea de clasificación, y podría ser la categoría a la que pertenece $\vec{x}$, y en una tarea de regresión, podría ser un valor numérico específico asociado a $\vec{x}$.

Cuando se introduce el conjunto de características $\vec{x}$ en la red neuronal, la red da como resultado una predicción, que llamaremos z. Esta predicción es el intento de la red de adivinar el valor de y basándose en lo que ha aprendido hasta el momento.

El proceso de entrenamiento ajusta constantemente lo que la red ha aprendido con el objetivo de hacer que z (la predicción de la red) se acerque lo más posible a y (el resultado correcto). Esto se hace modificando los "pesos"de la red, que son parámetros que influyen en cómo la red procesa las entradas. Al ajustar estos pesos, la red se vuelve gradualmente mejor en hacer predicciones precisas.

Capítulo 2

Tipos de arquitecturas en las redes.

2.1. Perceptrón simple

El perceptrón simple, un modelo fundamental en la evolución de las redes neuronales artificiales, se distingue por su arquitectura minimalista capaz de efectuar tareas de clasificación en espacios linealmente separables.

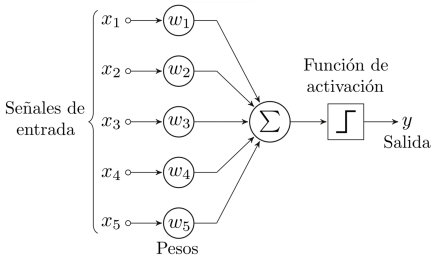


Figura 2.1.1: Perceptrón de una capa

La arquitectura del perceptrón simple se compone de los siguientes elementos principales:

- **Vector de Pesos:** Un conjunto de parámetros numéricos, denotado por $\vec{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$, que representa la importancia o influencia de cada entrada en la decisión final del perceptrón.
- **Bias (Sesgo):** Un término adicional, $w_0 \in \mathbb{R}$, que ajusta el umbral a partir del cual el perceptrón activa una salida en lugar de otra, permitiendo una mayor flexibilidad en la clasificación.
- **Función de Activación:** Una función matemática, $f : \mathbb{R} \to \mathbb{R}$, encargada de transformar la suma ponderada de las entradas y el bias en una salida específica del modelo. Para el perceptrón simple, se utiliza comúnmente una función de activación binaria o escalonada, que divide el espacio de entrada en dos categorías distintas.

Dado un vector de entrada $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, el perceptrón simple genera una salida $y \in \mathbb{R}$, que se calcula mediante la siguiente expresión:

$$y = P(\vec{x}) = f\left(w_0 + \sum_{i=1}^{n} w_i x_i\right). \tag{2.1.1}$$

La función de activación escalonada empleada en el perceptrón se define formalmente como:

$$f_k(x) = \begin{cases} 0 & \text{si } x \leq k, \\ 1 & \text{si } x > k, \end{cases} \tag{2.1.2}$$

Esta función mapea la suma ponderada de las entradas a uno de dos posibles estados, 0 o 1, basado en un umbral k. Este mecanismo permite al perceptrón actuar como un clasificador binario, donde la condición de separación entre las dos categorías es determinada por la ecuación del hiperplano $w_0 + \sum_{i=1}^{n} w_i x_i = k$ en el espacio de entrada $\mathbb{R}^n$.

Geométricamente, el perceptrón actúa como un hiperplano en el espacio multidimensional de las entradas, que divide este espacio en dos semiespacios. Cada semiespacio corresponde a uno de los estados posibles (0 o 1). La eficacia del perceptrón depende de que los datos sean linealmente separables, es decir, que exista un hiperplano que pueda dividir sin errores las categorías de datos que intenta clasificar.

2.2. Perceptrón multicapa

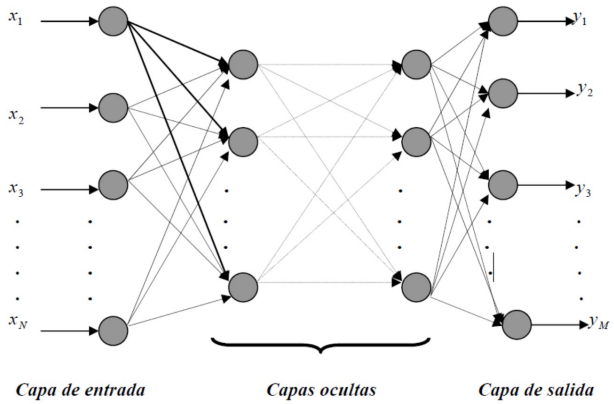


Figura 2.2.1: Perceptrón multicapa

El modelo de perceptrón multicapa (MLP) extiende la arquitectura del perceptrón simple mediante la incorporación de múltiples capas ocultas, lo cual introduce una mayor complejidad tanto en su estructura como en su proceso de entrenamiento. A diferencia del perceptrón simple, cuyo entrenamiento y estructura son relativamente directos al no emplear algoritmos avanzados de optimización, el entrenamiento del MLP se realiza mediante el algoritmo de retropropagación [C].

El proceso de operación del MLP se descompone en dos fases principales: la fase de propagación hacia adelante y la fase de retropropagación del error. En la fase de propagación hacia adelante, la señal de entrada se procesa secuencialmente a través de las capas de la red, empleando funciones de activación específicas en cada nodo, hasta generar la salida en la capa final. Posteriormente, en la fase de retropropagación, el error calculado

como la discrepancia entre la salida obtenida y el valor objetivo se propaga de manera inversa a través de la red, permitiendo la ajuste de los pesos sinápticos.

La estructura del MLP se define formalmente por dos conjuntos principales:

1. Un conjunto de pesos sinápticos,

$$\Theta = \{w_{ij}^l \mid i \in \{1, \ldots, n_l\}, j \in \{1, \ldots, n_{l-1}\}, l \in \{1, \ldots, m\}\},$$

   donde w_{ij}^l representa el peso entre el j-ésimo nodo en la capa $l-1$ y el i-ésimo nodo en la capa l.

2. Un conjunto de funciones de activación,

$$F = \{f_i^l : \mathbb{R} \to \mathbb{R} \mid i \in \{1, \ldots, n_l\}, l \in \{1, \ldots, m\}\},$$

   donde f_i^l designa la función de activación aplicada en el nodo i-ésimo de la capa l.

Aquí, n_l denota el número de nodos en la capa l y m representa el total de capas en la red.

Al igual que en el perceptrón simple, para ejecutar la red es necesario suministrar un conjunto de entradas, $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, a partir del cual el MLP produce una salida $y \in \mathbb{R}$. No obstante, la mayor complejidad estructural del MLP implica que su proceso de entrenamiento, fundamentado en el algoritmo de retropropagación, es sustancialmente más elaborado que el correspondiente al perceptrón de una sola capa, destacando la necesidad de una explicación detallada de este procedimiento en la siguiente sección.

2.3. Red convolucional

Las redes neuronales convolucionales [10], son una clase especializada de redes neuronales profundas diseñadas específicamente para procesar datos con una estructura matricial, como imágenes. Han demostrado ser altamente efectivas en una amplia gama de tareas, desde reconocimiento de objetos hasta segmentación de imágenes y más.

2.3.1. Capa convolucional

El corazón de las CNN es la operación de convolución. La convolución implica deslizar un pequeño filtro (o kernel) sobre la imagen de entrada y calcular el producto punto entre los valores de los píxeles de la imagen y los valores del filtro en cada posición. Matemáticamente, la convolución 2D de una imagen I con un filtro K se define como:

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n) \cdot K(m, n)$$

Esta operación produce un mapa de características que resalta ciertas características de la imagen, como bordes, texturas o patrones específicos.

Filtros

Los filtros en una CNN son matrices pequeñas que se aplican a la imagen de entrada. Cada filtro se especializa en detectar ciertas características, como bordes horizontales, verticales o diagonales, colores específicos, texturas, etc. Durante el entrenamiento, la red aprende automáticamente los filtros que son más útiles para cada tarea.

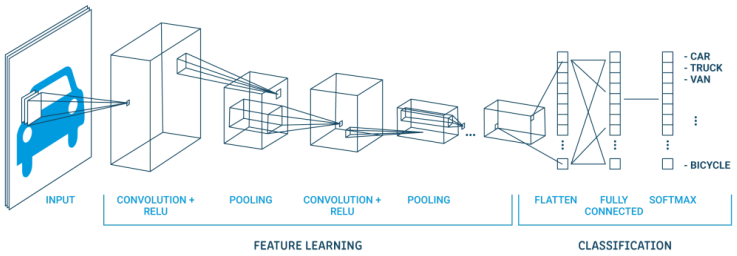


Figura 2.3.1: Estructura de red convolucional

2.3.2. Capas de Pooling

Después de la convolución, es común aplicar una capa de pooling para reducir la dimensionalidad de los datos y hacer que la red sea más robusta ante pequeñas variaciones en la posición de las características detectadas.

Pooling Máximo (Max Pooling)

En el max pooling, se divide la imagen en regiones y se toma el valor máximo de cada región, reduciendo así el tamaño de la representación. Esto se puede expresar como:

$$\text{MaxPooling}(I)(i,j) = \max_{m,n} I(i+m, j+n)$$

Pooling Promedio (Average Pooling)

En el average pooling se toma el valor promedio de cada región, lo que también reduce la dimensionalidad de los datos. Matemáticamente, se define como:

$$\text{AvgPooling}(I)(i,j) = \frac{1}{mn} \sum_{m} \sum_{n} I(i+m, j+n)$$

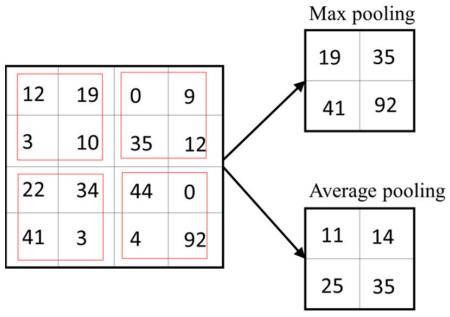


Figura 2.3.2: Pooling

2.4. Red recurrente

Las Redes Neuronales Recurrentes (RNN) son un tipo especializado de redes neuronales diseñadas para modelar secuencias de datos, donde la salida en cada paso de tiempo depende de la entrada actual y de la salida previa. Son ampliamente utilizadas en tareas que involucran datos secuenciales, como el procesamiento del lenguaje natural, la traducción automática y la predicción de series temporales.

La arquitectura básica de una RNN consiste en una capa de neuronas que se conectan consigo mismas a lo largo de una secuencia temporal. Cada neurona en una RNN recibe entradas tanto de la entrada actual como de su estado oculto anterior. Esto permite a la red mantener una memoria interna y capturar dependencias a largo plazo en los datos de secuencia.

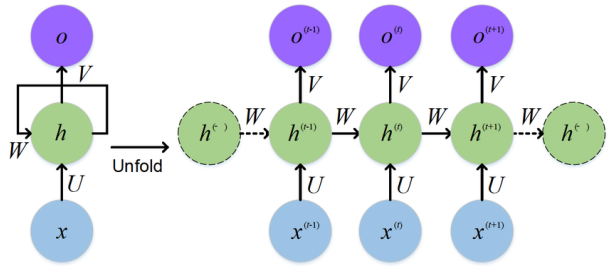


Figura 2.4.1: Estructura de un red neuronal recurrente

2.4.1. Celdas de Memoria en RNN

Celda de Memoria Simple (Simple RNN Cell)

La celda de memoria simple es la forma más básica de una unidad recurrente. En cada paso de tiempo, toma la entrada actual x_t y el estado oculto anterior h_{t-1}, y produce una nueva salida h_t. Matemáticamente, esto se puede expresar como:

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

donde σ es una función de activación no lineal, W_{hx} y W_{hh} son matrices de pesos y b_h es el sesgo.

Celdas de Memoria de Tipo LSTM y GRU

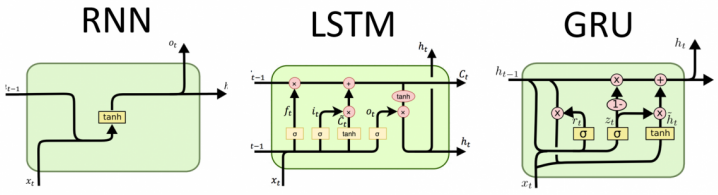


Figura 2.4.2: Distintas estructuras de redes recurrentes

Para abordar el problema del desvanecimiento del gradiente y la dificultad para capturar dependencias a largo plazo, se han propuesto varias variantes de RNN, como Long

Short-Term Memory (LSTM) [16] y Gated Recurrent Unit (GRU) [17]. Estas celdas de memoria están diseñadas para permitir un flujo de información más largo y son capaces de aprender a retener y olvidar información según sea necesario.

Capítulo 3

Algunos elementos fundamentales.

3.1. Funciones de activación

Las funciones de activación son responsables de introducir no linealidades en el modelo, por lo que juegan un papel fundamental en las redes neuronales permitiendo que la red aprenda y realice predicciones complejas. La elección de la función de activación adecuada es esencial, dado que impacta directamente en la capacidad de la red para converger durante el entrenamiento y en la precisión de las predicciones que genera.

3.1.1. Unidad Lineal Rectificada (ReLU)

La función ReLU es una función de cálculo rápido que ayuda a mitigar el problema de la saturación de gradiente cuando los valores de la combinación lineal de la entrada son mayores a 0 permitiendo una propagación hacia atrás efectiva incluso en redes profundas.

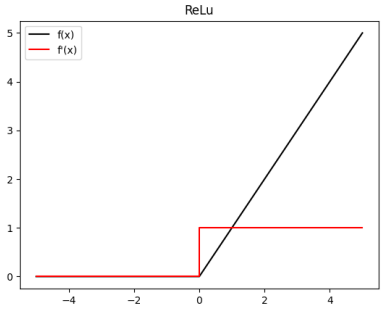


Figura 3.1.1: Relu

- Función:

$$f(x) = \begin{cases} 0 \text{ si } x < 0, \\ x \text{ si } x \geq 0 \end{cases}$$

- Derivada:

$$f'(x) = \begin{cases} 0 \text{ si } x < 0, \\ 1 \text{ si } x \geq 0 \end{cases}$$

3.1.2. SoftMax

Softmax no es una función de activación propiamente dicha, en el sentido que no recibe como entrada un único elemento, sino un vector. Su resultado es un vector cuya suma de valores es igual a 1, simulando así una distribución de probabilidad. La función

y su derivada son más complejas que Relu, necesitando así un mayor coste computacional.

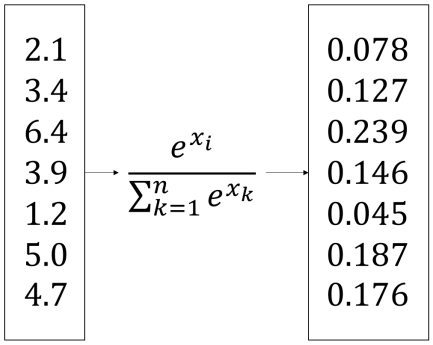


Figura 3.1.2: Softmax

- Función:

$$\alpha(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{n} e^{x_k}}$$

para $j = 1, \ldots, n$.

- Derivada:

$$\alpha'(x_j) = \frac{e^{x_j}(\sum_{k=1}^{n} e^{x_k}) - e^{2x_j}}{(\sum_{k=1}^{n} e^{x_k})^2}$$

para $j = 1, \ldots, n$.

3.1.3. Sigmoide

Posee la propiedad de mapear el dominio de los reales al intervalo (0,1), con lo cual se asemeja a una distribución de probabilidad. La función sigmoide sufre de un problema conocido como *covariate shift*, debido a que la salida de la función se encuentra centrada en 0.5 existe un desplazamiento de la salida de las neuronas que ocasiona que los pesos tengan problemas de convergencia.

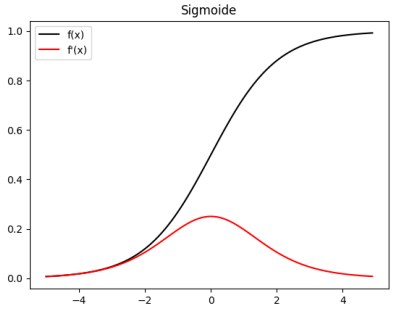


Figura 3.1.3: Sigmoide

- Función:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Derivada:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

3.1.4. Tangente hiperbólica

La tangente hiperbólica se puede ver como una versión escalada y desplazada de la sigmoide. Esta función mapea el dominio de los reales al intervalo abierto (-1, 1). Debido a lo anterior se encuentra centrada en 0 y elimina el problema del *covariate shift*. De igual forma su derivada es más 'activa' que la sigmoide, lo cual permite reducir tiempos de entrenamiento y convergencia, sin embargo, también sufre de saturación de gradiente a los extremos de la función, $\lim_{x \to \pm\infty} f'(x) = 0$.

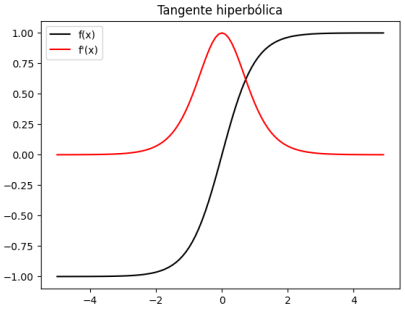


Figura 3.1.4: Tangente Hiperbólica

- Función:

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Derivada:

$$f'(x) = \frac{4e^{-2x}}{(1 + e^{-2x})^2}$$

3.2. Función de Pérdida

En el aprendizaje automático, especialmente en las redes neuronales, las funciones de pérdida (también conocidas como funciones de costo) juegan un papel crucial. Estas funciones cuantifican el error entre las predicciones realizadas por el modelo y los valores reales observados. El objetivo principal del entrenamiento de una red neuronal es minimizar esta función de pérdida, proceso que guía la optimización de los parámetros del modelo (pesos y sesgos).

La función de pérdida C es una función que toma como entradas el valor predicho por el modelo $\hat{y}$, y el valor real y, y devuelve un número que representa el coste asociado a la diferencia entre ambos. Matemáticamente, se expresa como:

$$C : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$$

donde $\mathcal{Y}$ es el conjunto de posibles etiquetas o valores reales, y $\mathbb{R}^+$ denota los números reales no negativos, indicando que el coste no puede ser negativo.

3.2.1. Ejemplos comunes

Error Cuadrático Medio (MSE)

El Error Cuadrático Medio (MSE, por sus siglas en inglés) es ampliamente utilizado en problemas de regresión. Se define como el promedio de los cuadrados de las diferencias entre los valores reales y los predichos:

$$C(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

donde y_i es el valor real y $\hat{y}_i$ es la predicción del modelo para la i-ésima instancia.

Entropía Cruzada

La Entropía Cruzada es una medida de la diferencia entre dos distribuciones de probabilidad, y se utiliza principalmente en tareas de clasificación. Para la clasificación binaria, se define como:

$$C(y, \hat{y}) = -\sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

y para la clasificación multiclase, se extiende a:

$$C(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \log(\hat{y}_{ij})$$

donde y_{ij} indica si la clase j es la clase correcta para la observación i y $\hat{y}_{ij}$ es la probabilidad predicha de que la observación i pertenezca a la clase j.

3.3. Optimizadores

La optimización es una piedra angular en el campo del aprendizaje automático y, en particular, es esencial para el entrenamiento eficaz de redes neuronales. Este proceso matemático se enfoca en encontrar los valores óptimos de los parámetros de un modelo que minimizan o maximizan una función objetivo. En el contexto de las redes neuronales, la función objetivo suele ser la función de pérdida o costo, $L(\Theta)$ [D], que mide el desempeño del modelo al comparar las salidas predichas con los valores reales esperados.

Por lo tanto, el proceso de optimización busca ajustar los parámetros de la red, denotados por Θ, de tal manera que se minimice la función de pérdida. Este enfoque no solo mejora la precisión de las predicciones realizadas por la red, sino que también contribuye a la generalización del modelo, permitiéndole realizar predicciones acertadas sobre datos no vistos anteriormente.

Matemáticamente, el objetivo de la optimización en redes neuronales se puede expresar como la búsqueda del conjunto de valores para Θ que resulte en el mínimo valor de $L(\Theta)$:

$$\min_{\Theta} L(\Theta)$$

donde Θ representa el conjunto de todos los parámetros de la red, que incluyen los pesos y sesgos distribuidos a lo largo de sus diversas capas.

3.3.1. Descenso del Gradiente Estocástico (SGD)

El Descenso del Gradiente Estocástico (SGD) es una variante del método de descenso del gradiente tradicional, optimizado para tratar con grandes volúmenes de datos. A diferencia del descenso del gradiente que requiere el cálculo del gradiente de la función de pérdida sobre el conjunto de datos completo, el SGD actualiza los parámetros utilizando el gradiente basado solo en un subconjunto aleatorio de datos, conocido como mini-lote. Esta metodología reduce significativamente la carga computacional y permite actualizaciones de parámetros más frecuentes, lo que puede acelerar la convergencia en prácticas de entrenamiento de grandes redes neuronales.

Regla de Actualización de Parámetros

$$\Theta_{t+1} = \Theta_t - \eta \nabla_{\Theta} L(\Theta_t, X_{\text{mini-lote}})$$

donde Θ_t representa los parámetros de la red en la iteración t, η es la tasa de aprendizaje, y $\nabla_\Theta L(\Theta_t, X_{\text{mini-lote}})$ es el gradiente de la función de pérdida evaluado en un mini-lote $X_{\text{mini-lote}}$.

Cálculo del Gradiente para el mini-lote

Cuando se tiene una función de pérdida compuesta por la suma de varias funciones de pérdida individuales, como en la siguiente expresión:

$$L(\Theta) = \sum_{i=1}^{n} L(x_i, y_i; \Theta)$$

donde $L(x_i, y_i; \Theta)$ es la función de pérdida asociada a la entrada x_i y la salida deseada y_i, y Θ representa los parámetros (pesos y sesgos) de la red, el gradiente de $L(\Theta)$ con respecto a los parámetros Θ se calcula utilizando la propiedad de linealidad del operador de derivada.

La derivada (o gradiente) de una suma de funciones es igual a la suma de las derivadas (o gradientes) de cada función. Esto se expresa matemáticamente como:

$$\nabla_\Theta L(\Theta) = \nabla_\Theta \left(\sum_{i=1}^{n} L(x_i, y_i; \Theta) \right) = \sum_{i=1}^{n} \nabla_\Theta L(x_i, y_i; \Theta)$$

Durante el proceso de backpropagation en el entrenamiento de una red neuronal, se siguen los siguientes pasos:

1. **Propagación hacia adelante:** Cada entrada x_i es procesada por la red para obtener la predicción $\hat{y}_i$, utilizando los parámetros actuales Θ.

2. **Cálculo del Error:** Se calcula la pérdida $L(x_i, y_i; \Theta)$ para cada par de entrada-salida usando una función de pérdida predefinida.

3. **Propagación hacia atrás (Backpropagation):** Se calcula el gradiente de la pérdida respecto a cada parámetro de la red para cada ejemplo individual, $\nabla_\Theta L(x_i, y_i; \Theta)$.

4. **Acumulación de gradientes:** Los gradientes individuales se suman para obtener el gradiente total de la pérdida acumulada:

$$\nabla_\Theta L(\Theta) = \sum_{i=1}^{n} \nabla_\Theta L(x_i, y_i; \Theta)$$

5. **Actualización de Parámetros:** Los parámetros de la red se actualizan en dirección opuesta al gradiente acumulado, típicamente utilizando una tasa de aprendizaje η:

$$\Theta \leftarrow \Theta - \eta \nabla_\Theta L(\Theta)$$

Si se desea utilizar el promedio de las pérdidas para regularizar el impacto del tamaño del conjunto de datos, simplemente se divide el gradiente acumulado por el número de ejemplos n, y la actualización de los parámetros se realiza igualmente pero con este gradiente promedio.

3.3.2. Estimación adaptativa del momento (ADAM)

ADAM combina elementos de otros dos optimizadores, RMSprop y Momentum, ajustando las tasas de aprendizaje de los parámetros a través de estimaciones de los primeros y segundos momentos de los gradientes.

Regla de Actualización de Parámetros

$$\Theta_{t+1} = \Theta_t - \frac{\eta_t}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Donde:

- Θ_t: Vector de parámetros del modelo en la iteración t.
- η_t: Tasa de aprendizaje en el tiempo t, que puede ser ajustada dinámicamente.
- $\hat{m}_t$: Estimación corregida del primer momento (la media del gradiente) en el tiempo t. Se calcula como:
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
- $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L(\Theta_t)$
- $\hat{v}_t$: Estimación corregida del segundo momento (la varianza no centrada del gradiente) en el tiempo t. Se calcula como:
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
- $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla L(\Theta_t))^2$
- β_1, β_2: Factores de decaimiento para los estimados de los momentos. Típicamente, $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$ son valores recomendados.
- ϵ: Un pequeño número (por ejemplo, 10^{-8}) para evitar divisiones por cero.

Explicación de los Parámetros en ADAM

ADAM ajusta adaptativamente las tasas de aprendizaje de los parámetros a través de estimaciones de los primeros y segundos momentos de los gradientes. Los parámetros clave en ADAM incluyen:

- $\hat{m}_t$ **(Primer momento corregido)** y $\hat{v}_t$ **(Segundo momento corregido)**: Estos son utilizados para obtener tasas de aprendizaje adaptativas para cada parámetro. El primer momento $\hat{m}_t$ es una estimación sesgada hacia cero del primer momento (la media del gradiente), y el segundo momento $\hat{v}_t$ es una estimación sesgada hacia cero de los cuadrados del gradiente (varianza no centrada).
- β_1 **y** β_2: Son los factores de decaimiento para los estimados de los momentos. β_1 afecta la estimación del primer momento del gradiente, y β_2 maneja la estimación del segundo momento. Típicamente, se utilizan valores de $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$, proporcionando un buen equilibrio entre estabilidad y reactividad en la actualización de parámetros.
- ϵ: Es un pequeño número añadido al denominador en la actualización de parámetros para evitar la división por cero, típicamente alrededor de 10^{-8}. Este término ayuda a mantener la estabilidad numérica del algoritmo.

Estos componentes de ADAM son críticos para su efectividad y para la adaptación de las tasas de aprendizaje en base al comportamiento de los gradientes, facilitando el entrenamiento eficiente de modelos complejos en aprendizaje profundo.

Capítulo 4

Ejemplos de redes.

En este capítulo presentaremos diversos ejemplos implementados íntegramente en Python. A lo largo de las diferentes secciones mostraremos fragmentos de código ilustrativos para explicar detalles específicos. Para acceder al código completo ir a los códigos del anexo.

4.1. Perceptrón simple - Separación lineal

En este ejemplo [F] vamos a realizar una separación lineal utilizando la arquitectura del perceptrón simple.

```
class Perceptron:

    def __init__(self, n_inputs, lr):
        self.n = n_inputs
        self.lr = lr
        self.w = np.random.randn(n_inputs)
        self.w0 = np.random.randn(1)

    def forward(self, X):
        pred = np.dot(self.w, X) + self.w0
        return pred
```

Listing 4.1: Creación del perceptrón

El objetivo es dividir el espacio bidimensional en dos partes, $A = \{r(x) > 0\}$ y $B = \{r(x) \leq 0\}$. Para este ejemplo, utilizaremos la recta $r(x) = mx + n$ en el espacio, con los parámetros $m = -2$ y $n = +12$.

Una vez dividido el espacio en dos, generaremos puntos aleatorios para entrenar el modelo. El perceptrón recibirá dos entradas por cada ejemplo, las coordenadas x e y, y tendrá una salida z que buscamos que sea igual a la salida de la recta $r(x)$. Para entrenarlo, etiquetaremos los puntos del subespacio A con 1 y los del B con -1, con la intención de que la salida del perceptrón $z = w_1 \cdot x + w_2 \cdot y + w_0$, termine siendo tras el entrenamiento $z = mx - y + n$. Para tener los puntos de A con valores positivos y los de B con valores negativos, utilizaremos una función de activación simple

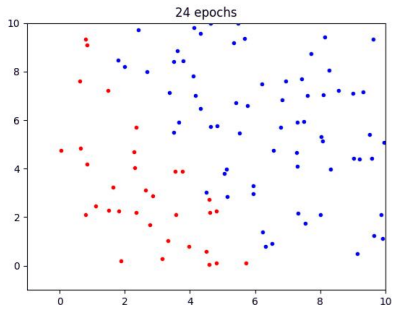
$$f(x) = \begin{cases} -1 & \text{si} \quad x \leq 0 \\ 1 & \text{si} \quad x > 0 \end{cases} \tag{4.1.1}$$

para llevar los resultados a un rango entre -1 y 1.

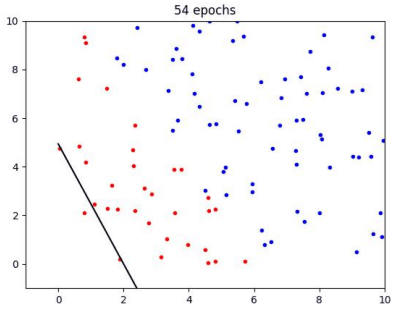
```
# functions
f = lambda x : -2*x + 12
g = lambda x : 1 if x > 0 else -1

# points
px = 10*np.random.rand(n)
py = 10*np.random.rand(n)

points1 = np.array([(x,y) for x,y in zip(px,py) if y > f(x)])
points2 = np.array([(x,y) for x,y in zip(px,py) if y <= f(x)])

# training data
X = np.array([px, py])
Y = np.array(list(map(g, (py-np.array(list(map(f, px)))))))
```

Listing 4.2: Creación de puntos en el plano

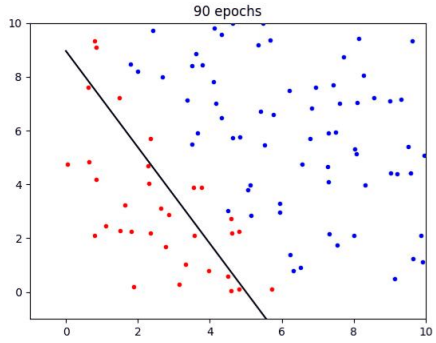
Realizaremos el entrenamiento y comprobaremos cómo el perceptrón separa prácticamente perfectamente el espacio según los puntos de ejemplo que tenemos. Para ello definimos la función *fit* dentro del perceptrón.



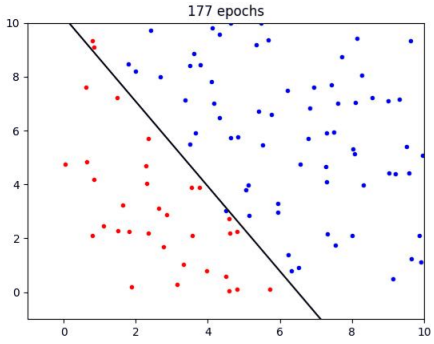
(a) Frame 24



(b) Frame 54



(c) Frame 90



(d) Frame 177

Figura 4.1.1: Evolución del perceptrón durante el entrenamiento

```
def fit(self, X, Y, epochs=1):
```

```
    losses = []
    for epoch in range(epochs):
      for i in range(len(X[0])):
        x = X[:,i]
        y = Y[i]
        pred = self.forward(x)
        error = y-pred
        self.w += self.lr * error * x
        self.w0 += self.lr * error
        losses.append(abs(error))
      print(f"[{epoch+1}] last error: {losses[-1]} & mean error: {sum(
    losses)/len(losses)}")
    return losses
```

Listing 4.3: Función para entrenar el perceptrón con los puntos del plano

Tras 200 etapas observamos cómo el ajuste del perceptrón es casi perfecto teniendo encuenta los datos de entrenamiento conocidos.

4.2. Redes Convolucionales - datos MNIST

En este ejemplo [G] exploraremos el uso de redes convolucionales para la clasificación de dígitos escritos a mano utilizando el conjunto de datos MNIST. El conjunto de datos MNIST [22] es una colección de imágenes en escala de grises de dígitos escritos a mano, que van desde 0 hasta 9. Cada imagen tiene una resolución de 28x28 píxeles. Para ello, en python, utilizaremos el dataset proporcionado por keras.

```
from keras.datasets import mnist
(X, Y), (test_X, test_y) = mnist.load_data()
```

El objetivo principal es familiarizarnos con el uso de redes convolucionales para tareas de clasificación de imágenes. Las redes convolucionales son especialmente efectivas en tareas de visión por computadora debido a su capacidad para aprender y detectar características locales en las imágenes, como bordes, texturas y patrones. Para ello haremos uso de la librería de redes neuronales de pytorch que nos proporcionará la arquitectura necesaria para crear el modelo.

```
from torch import nn
```

Utilizaremos una arquitectura de red convolucional relativamente simple que consta de varias capas convolucionales y capas de normalización. Las capas convolucionales son responsables de extraer características relevantes de las imágenes, mientras que las capas de normalización ayudan a estabilizar y acelerar el proceso de entrenamiento.

La arquitectura es la siguiente:

```
class Model(nn.Module):
    def __init__(self, lr):
        super(Model, self).__init__()
        self.layer1 = nn.Conv2d(1, 32, 3, padding=1)
        self.layer2 = nn.BatchNorm2d(32)
```

```
        self.layer3 = nn.ReLU()
        self.layer4 = nn.Conv2d(32, 32, 3, padding=1)
        self.layer5 = nn.BatchNorm2d(32)
        self.layer6 = nn.ReLU()
        self.layer7 = nn.Conv2d(32, 1, 3, padding=1)
        self.layer8 = nn.BatchNorm2d(1)
        self.layer9 = nn.ReLU()
        self.layer10 = nn.Flatten()
        self.layer11 = nn.Linear(28*28, 10)
```

Esta red consta de varias capas convolucionales intercaladas con capas de normalización y activación ReLU. La capa final es una capa lineal que produce la salida final de la red, que representa las probabilidades de cada clase (dígito del 0 al 9).

Tras el entrenamiento obtenemos una precisión del 96.54 %, muy efectiva teniendo en cuenta el tamaño del modelo.

Explorando los Filtros Convolucionales

Una parte interesante de las redes convolucionales es su capacidad para aprender y visualizar los filtros convolucionales que se aplican a las imágenes de entrada. Estos filtros actúan como detectores de características y son esenciales para la capacidad de la red para reconocer patrones en las imágenes.

En la siguiente imagen observamos los 32 filtros convolucionales 3x3 aprendidos por la primera capa de la red.

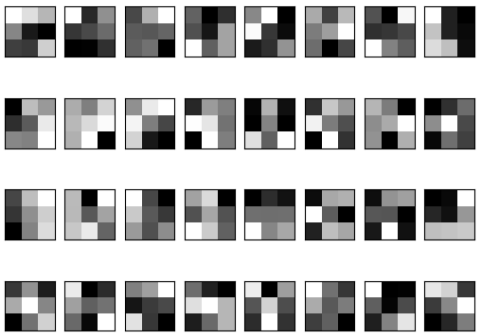


Figura 4.2.1: Filtros de la primera capa de la red

Estos filtros capturan características simples, como bordes, texturas y gradientes, que son fundamentales para el proceso de clasificación. A continuación observamos los resultados de un ejemplo

pasado por los 32 filtros de la red, para observar en qué se está fijando internamente la red. Tenemos dos visualizaciones dependiendo de la escala de color para apreciar mejor cada filtro.

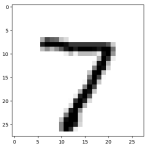


Figura 4.2.2: Ejemplo MNIST

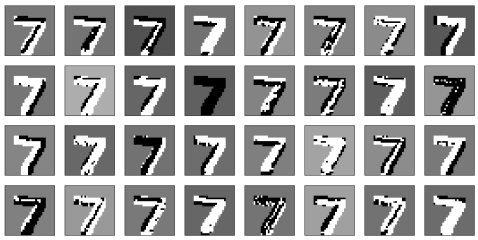


Figura 4.2.3: Ejemplo MNIST tras el primer filtro

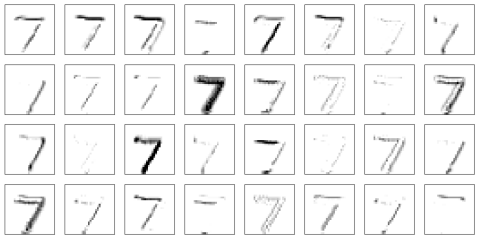


Figura 4.2.4: Ejemplo MNIST tras el primer filtro (V2)

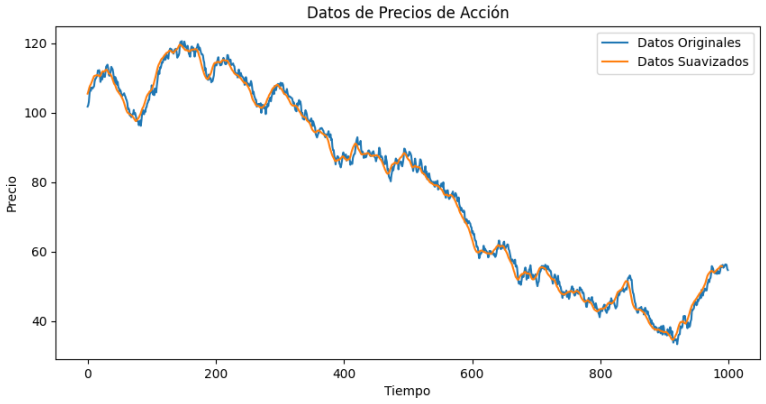
4.3. Redes Recurrentes para Predicción de Series Temporales

Las redes recurrentes, como las LSTM (Long Short-Term Memory), son ampliamente utilizadas en la predicción de series temporales debido a su capacidad para capturar dependencias temporales en los datos. Estas redes son capaces de recordar información de estados anteriores a medida que procesan nuevas entradas, lo que las hace ideales para modelar patrones y tendencias en series temporales, como los precios de acciones.

En este ejemplo [H], utilizaremos una red LSTM para predecir el precio de una acción en la siguiente hora en función de los precios históricos de la acción. Utilizaremos de nuevo la biblioteca Keras, que proporciona una interfaz sencilla para construir y entrenar redes neuronales.

Primero, generaremos datos de ejemplo que simulan los precios históricos de una acción. Utilizaremos la función *numpy.random.randn()* para generar precios aleatorios y luego aplicaremos una función de suavizado para que los datos se asemejen más a los precios reales de una acción.

```
import numpy as np

# Generar datos de precios de ejemplo
np.random.seed(0)
num_datos = 1000
precios = 100 + np.cumsum(np.random.randn(num_datos))

# Funci n de suavizado para simular datos de precios de acciones
def suavizar_datos(datos, ventana=10):
    suavizado = np.convolve(datos, np.ones(ventana)/ventana, mode='valid')
    return suavizado

precios_suavizados = suavizar_datos(precios)

# Visualizar los datos de precios de ejemplo
import matplotlib.pyplot as plt

plt.plot(precios, label='Datos Originales')
plt.plot(range(len(precios_suavizados)), precios_suavizados, label='Datos
    Suavizados')
plt.xlabel('Tiempo')
plt.ylabel('Precio')
plt.legend()
plt.title('Datos de Precios de Acci n')
plt.show()
```



Ahora que tenemos nuestros datos de precios de ejemplo, podemos dividirlos en secuencias de longitud fija para entrenar nuestra red LSTM. Utilizaremos, por ejemplo, las 24 horas anteriores para predecir el precio de la siguiente hora.

```
# Funci n para crear secuencias de datos
def crear_secuencias(datos, longitud_secuencia):
    secuencias = []
    for i in range(len(datos) - longitud_secuencia):
        secuencia = datos[i:i+longitud_secuencia]
        objetivo = datos[i+longitud_secuencia]
        secuencias.append((secuencia, objetivo))
    return secuencias

# Definir la longitud de la secuencia
longitud_secuencia = 24  # Utilizamos 24 horas anteriores para predecir la
    siguiente

```

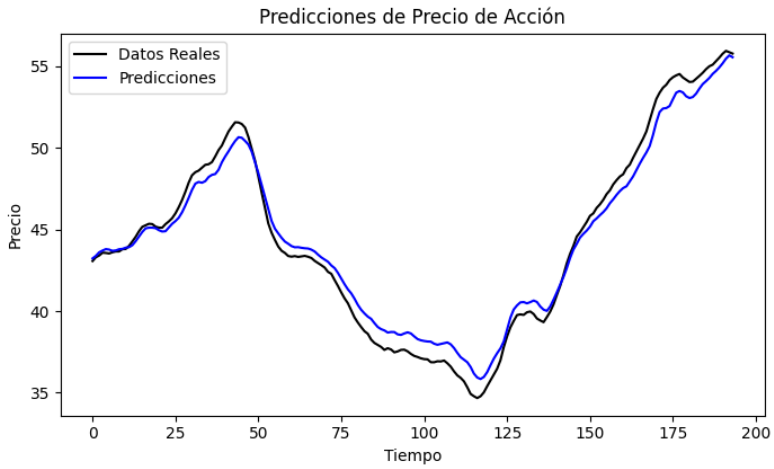
```
# Crear secuencias de datos
secuencias = crear_secuencias(precios_suavizados, longitud_secuencia)

# Dividir los datos en conjuntos de entrenamiento y prueba
porcentaje_entrenamiento = 0.8
num_entrenamiento = int(porcentaje_entrenamiento * len(secuencias))

datos_entrenamiento = secuencias[:num_entrenamiento]
datos_prueba = secuencias[num_entrenamiento:]

# Convertir los datos a arreglos numpy
X_train, y_train = np.array([x for x, _ in datos_entrenamiento]), np.array
    ([y for _, y in datos_entrenamiento])
X_test, y_test = np.array([x for x, _ in datos_prueba]), np.array([y for _,
     y in datos_prueba])

# Ajustar las dimensiones de los datos para LSTM (n mero de muestras,
    longitud de la secuencia, n mero de caracter sticas)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

Con los datos preparados, ahora podemos construir y entrenar nuestra red LSTM utilizando Keras.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense

# Definir el modelo LSTM
modelo = Sequential()
modelo.add(LSTM(50, input_shape=(longitud_secuencia, 1)))
modelo.add(Dense(1))

# Compilar el modelo
modelo.compile(optimizer='adam', loss='mean_squared_error')

# Entrenar el modelo
modelo.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(
    X_test, y_test), verbose=1)
```

Una vez entrenado el modelo, podemos hacer predicciones sobre los datos de prueba y visualizar los resultados.

```
# Hacer predicciones
predicciones = modelo.predict(X_test)

# Visualizar los resultados
plt.plot(y_test, label='Datos Reales')
plt.plot(predicciones, label='Predicciones')
plt.xlabel('Tiempo')
plt.ylabel('Precio')
plt.legend()
plt.title('Predicciones de Precio de Acci n')
plt.show()
```


Predicciones de Precio de Acción
Datos Reales
Predicciones
Precio
Tiempo
55
50
45
40
35
0
25
50
75
100
125
150
175
200

Parte II

PARTE 2: Aprendizaje por transferencia.

Capítulo 5

Introducción al apredizaje por transferencia

5.1. Introducción

El aprendizaje por transferencia [20] es una técnica en el campo del aprendizaje automático, especialmente relevante para las redes neuronales profundas. Esta metodología surgió como solución a dos grandes desafíos: la necesidad de extensos volúmenes de datos y el alto coste computacional asociado al entrenamiento de modelos profundos. La premisa del aprendizaje por transferencia consiste en utilizar el conocimiento adquirido (modelos preentrenados) en una tarea para facilitar el aprendizaje en tareas relacionadas pero distintas.

Este enfoque se basa en la observación de que ciertas características generales aprendidas en tareas como el reconocimiento visual son transferibles a una amplia variedad de problemas dentro del mismo dominio. Por ejemplo, las características que detectan bordes, texturas y formas en imágenes son útiles en múltiples tareas de procesamiento visual.

5.2. Fundamentos

Desde una perspectiva matemática, consideramos dos dominios de aprendizaje: el dominio fuente D_S donde el modelo ha sido entrenado, y el dominio objetivo D_T donde se aplicará el modelo. Un modelo de red neuronal profunda generalmente consta de múltiples capas, donde las capas iniciales funcionan como extractores genéricos de características y las capas profundas se especializan más hacia la tarea original.

El proceso de transferencia típicamente implica:

1. **Selección del Modelo Preentrenado:** Elección de un modelo entrenado previamente en un conjunto de datos amplio.
2. **Reutilización de Características:** Uso de las capas iniciales del modelo preentrenado, que contienen características genéricas útiles para la nueva tarea.
3. **Afinamiento (Fine-Tuning):** Ajuste de las capas profundas que se entrenan nuevamente en un conjunto de datos específico para la nueva tarea, ajustando los pesos a partir de los ya aprendidos.

Para implementar esto técnicamente, es posible "congelar" las capas iniciales del modelo, de modo que sus pesos no se actualicen durante el entrenamiento en el nuevo conjunto de datos. Esto se realiza estableciendo los parámetros de estas capas como no entrenables:

```
for layer in model.layers[:n]:
    layer.trainable = False
```

Capítulo 6

Ejemplos de aprendizaje por transferencia

En este capítulo, exploraremos ejemplos prácticos de aprendizaje por transferencia, con especial atención en su aplicación y efectividad. El segundo ejemplo, enfocado en entrenamiento de procesamiento de lenguaje natural, que es notablemente exigente desde el punto de vista computacional, ha sido desarrollado utilizando un entorno en una cuenta básica de Google Colab para aprovechar el uso de una GPU. Esto nos ha permitido acelerar significativamente el proceso de entrenamiento. Para acceder al código completo ir a los códigos del anexo.

6.1. Clasificación de Imágenes con ImageNet: Gatos vs Perros

En este ejemplo [I], exploraremos cómo utilizar el conjunto de datos ImageNet [4] y la técnica de transferencia de aprendizaje para entrenar un modelo de clasificación binaria que distinga entre imágenes de gatos y perros. Utilizaremos el modelo MobileNetV2 preentrenado en ImageNet como punto de partida y adaptaremos las capas finales para nuestro problema específico.

6.1.1. Configuración y Preparación de Datos

En esta sección, configuraremos el entorno y prepararemos los datos para el entrenamiento del modelo.

Comenzaremos importando las bibliotecas necesarias, para la construcción y entrenamiento del modelo.

```
from keras.applications import MobileNetV2
from keras.models import Sequential
from keras.layers import Dense, GlobalAveragePooling2D
from keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.applications.mobilenet_v2 import preprocess_input,
    decode_predictions
import numpy as np
import os
```

Después nos descargaremos una base de datos de perros y gatos para poder realizar el entrenamiento de nuestro modelo. Para ello nos iremos a Kaggle y descargaremos el dataset de *Cat VS Dog Dataset* el cual contiene 12491 imágenes de gatos y 12470 imágenes de perros. Una vez descargado lo separaremos en dos datasets, uno para el entrenamiento en el que dejaremos 10.000 imagénes de cada tipo y el resto para la validación, es decir un split de 80 / 20. El sistema de archivos debería quedar de la siguiente forma:

```
dataset/
    train/
        perro/
            imagen1.jpg
            imagen2.jpg
            ...
        gato/
            imagen1.jpg
            imagen2.jpg
            ...
    validation/
        perro/
            imagen1.jpg
            imagen2.jpg
            ...
        gato/
            imagen1.jpg
            imagen2.jpg
            ...
```

6.1.2. Preparación del Modelo y Aprendizaje por Transferencia

En esta sección, configuraremos el modelo de clasificación utilizando la técnica de transferencia de aprendizaje y prepararemos las capas finales para la clasificación binaria de gatos y perros.

Utilizaremos el modelo pre-entrenado MobileNetV2, que ha sido entrenado en el conjunto de datos ImageNet para tareas de clasificación de imágenes.

```
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape
    =(224, 224, 3))  # Excluir las capas finales y especificar el tama o
    de entrada
```

y congelaremos todas las capas del modelo pre-entrenado, excepto las capas finales.

```
for layer in base_model.layers:
    layer.trainable = False
```

Posteriormente crearemos el modelo final añadiendo nuevas capas para adaptar el modelo a nuestra tarea de clasificación de gatos y perros.

```
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
```

```
    Dense(1, activation='sigmoid')  # Capa final con activaci n sigmoide
    para la clasificaci n binaria
])
```

Para obtener un resumen del modelo podemos hacer *model.summary()* y obtenemos lo siguiente:

```
Model: "sequential"

    Layer (type)                             Output Shape
                Param #

    mobilenetv2_1.00_224 (Functional)        (None, 7, 7, 1280)
              2,257,984

    global_average_pooling2d                 (None, 1280)
                        0
    (GlobalAveragePooling2D)

    dense (Dense)                            (None, 128)
                163,968

    dense_1 (Dense)                          (None, 1)
                    129

 Total params: 2,750,277 (10.49 MB)
 Trainable params: 164,097 (641.00 KB)
 Non-trainable params: 2,257,984 (8.61 MB)
 Optimizer params: 328,196 (1.25 MB)
```

6.1.3. Entrenamiento del Modelo

En esta sección, entrenaremos el modelo utilizando los datos preparados y evaluaremos su rendimiento en un conjunto de datos de validación.

Primero compilaremos el modelo utilizando una función de pérdida binaria y el optimizador adam.

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy'])
```

Finalmente creamos la función de entrenamiento. En ella primero generamos los objetos *train_generator* y *validation_generator* que contienen las imagenes de nuestro dataset más variaciones de las mismas con giros, ampliaciones, etc, un práctica muy común a la hora de trabajar con imagenes para poder duplicar y triplicar el tamaño de nuestros datasets de manera inteligente. Una vez creados dichos objetos entrenamos el modelo con la función implícita de keras con *model.fit()*.

```python
def entrenar_modelo(modelo, directorio_entrenamiento, directorio_validacion, epochs=10, batch_size=32):
    # Configurar generadores de datos para entrenamiento y validaci n con aumento de datos
    train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
    )

    test_datagen = ImageDataGenerator(rescale=1./255)

    train_generator = train_datagen.flow_from_directory(
        directorio_entrenamiento,
        target_size=(224, 224),
        batch_size=batch_size,
        class_mode='binary'  # Especificar la clase binaria (0 para gato, 1 para perro)
    )

    validation_generator = test_datagen.flow_from_directory(
        directorio_validacion,
        target_size=(224, 224),
        batch_size=batch_size,
        class_mode='binary'  # Especificar la clase binaria (0 para gato, 1 para perro)
    )

    # Entrenar el modelo
    modelo.fit(
        train_generator,
        steps_per_epoch=train_generator.samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=validation_generator.samples // batch_size
    )
```

6.1.4. Evaluación y Conclusión

Figura 6.1.1: Predicciones: Gato vs Perro

Podemos observar cómo tras un entrenamiento básico de 1'5h hemos conseguido crear

una red casi perfecta, respecto al tipo de imágenes de nuestro dataset, que diferencie entre perros y gatos. Esto es gracias al aprovechamiento de la red *imagenet* que lleva por detras un entrenamiento de millones de imágenes y horas de entrenamiento con un poder computacional mucho superior al de mi ordenador, por lo que sería impensable obtener estos resultados si quisiera entrenar una red desde cero.

6.2. Análisis de sentimientos con BERT

En el dominio del procesamiento del lenguaje natural (NLP), la adaptación y ajuste fino de modelos preentrenados a tareas específicas han demostrado un éxito significativo. Este enfoque, conocido como aprendizaje por transferencia, aprovecha las características aprendidas de un dominio de problemas para resolver problemas similares con menos gasto computacional que entrenar un modelo desde cero. Este capítulo [J] explora la implementación del aprendizaje por transferencia para el análisis de sentimientos utilizando el modelo BERT (Bidirectional Encoder Representations from Transformers), un método de vanguardia para tareas de NLP.

6.2.1. Configuración y Preparación de Datos

El proceso comienza en un entorno de Google Colab, facilitando el acceso a recursos computacionales potentes como las GPUs. El primer paso implica montar Google Drive para acceder al conjunto de datos almacenado en la nube:

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

El conjunto de datos utilizado es "SMSSpamCollection", que contiene una mezcla de mensajes de spam y ham (no spam). La carga de datos se ejecuta con pandas, una biblioteca de manipulación de datos:

```
import pandas as pd
import os

main_path = "/content/drive/MyDrive/Colab Notebooks/"
file = "SMSSpamCollection.txt"
df = pd.read_csv(os.path.join(main_path, "data", file), sep='\t', names=["
    category", "text"])
df["label"] = df["category"].astype("category").cat.codes
```

Este fragmento carga los datos, asignando códigos categóricos a las etiquetas—'ham' se convierte en 0 y 'spam' en 1, facilitando el entrenamiento del modelo.

6.2.2. Preparación del Modelo y Aprendizaje por Transferencia

El modelo BERT empleado es "distilbert-base-uncased"[19], una versión más ligera de BERT que retiene la mayor parte del rendimiento del modelo original. Este modelo es adecuado para entornos con recursos computacionales limitados, como dispositivos móviles. Antes de utilizar el modelo, se debe cargar el tokenizador específico para BERT:

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-
    uncased')
```

La tokenización [12] convierte el texto en un formato compatible con los requisitos de entrada del modelo. Después de la tokenización, el conjunto de datos se divide en conjuntos de entrenamiento, prueba y validación para garantizar que el rendimiento del modelo sea evaluado adecuadamente:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df['text'], df['label'
    ], test_size=0.25, random_state=0)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size
    =0.50, random_state=0)
```

6.2.3. Entrenamiento del Modelo BERT

Con los datos preparados y el modelo preentrenado cargado, el siguiente paso involucra adaptar BERT a la tarea de análisis de sentimientos. Esto se hace mediante ajuste fino, donde el modelo preentrenado se entrena (o ajusta) más sobre un conjunto de datos específico. Es crucial notar que solo las capas superiores del modelo se entrenan, ya que son más específicas de la tarea, mientras que las capas inferiores permanecen congeladas:

```
from transformers import TFDistilBertForSequenceClassification
model = TFDistilBertForSequenceClassification.from_pretrained('distilbert-
    base-uncased', num_labels=2)

model.layers[0].trainable = False  # Congela las capas del transformador
model.layers[1].trainable = False
model.summary()
```

```
Model: "tf_distil_bert_for_sequence_classification"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 distilbert (TFDistilBertMai  multiple                 66362880
 nLayer)

 pre_classifier (Dense)      multiple                  590592

 classifier (Dense)          multiple                  1538

 dropout_19 (Dropout)        multiple                  0

=================================================================
Total params: 66,955,010
Trainable params: 1,538
Non-trainable params: 66,953,472
_________________________________________________________________
```

Listing 6.1: Resumen del modelo”

Este enfoque reduce el tiempo y el costo computacional de entrenamiento mientras se adapta el modelo para entender efectivamente el contexto de los mensajes de spam versus ham. Posteriormente, el modelo se compila y se entrena usando los conjuntos de datos preparados:

```
model.compile(optimizer="adam")
model.fit(train_dataset, validation_data=val_dataset, epochs=3, batch_size
    =64)
```

```
Epoch 1/3
4179/4179 [==============================] - 1442s 345ms/step - loss:
    0.1396 - val_loss: 0.0739
Epoch 2/3
4179/4179 [==============================] - 1434s 343ms/step - loss:
    0.0734 - val_loss: 0.0596
Epoch 3/3
4179/4179 [==============================] - 1456s 348ms/step - loss:
    0.0645 - val_loss: 0.0587
```

Listing 6.2: Tiempo total de entrenamiento con una cuenta básica de Google Colab: 1h 12m

6.2.4. Evaluación y Conclusión

Una vez entrenado el modelo, se realizan predicciones sobre el conjunto de prueba y se evalúa su desempeño mediante una matriz de confusión:

```
from transformers import TextClassificationPipeline
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

pipeline = TextClassificationPipeline(model=model, tokenizer=tokenizer)
pred = pipeline(X_test)
pred_labels = [int(p["label"][-1]) for p in pred]

cm = confusion_matrix(y_test, pred_labels)
cm_display = ConfusionMatrixDisplay(cm).plot()
```

obteniendo los siguientes resultados

con una precisión superior al 95 %, este modelo demuestra su robustez. Sin embargo, entrenar un modelo de esta envergadura desde cero sería una tarea descomunal, especialmente con recursos computacionales limitados. Por ejemplo, el entrenamiento de solo 1,538 parámetros me ha llevado aproximadamente 1.2 horas. Extrapolando de manera directa, lo cual es meramente ilustrativo ya que las capas más profundas requieren un esfuerzo computacional mayor debido a la complejidad de calcular los gradientes y al manejo de un volumen significativamente mayor de parámetros y variables en memoria, entrenar el modelo completo con 66,955,010 parámetros tomaría cerca de seis años. Esto subraya la magnitud del modelo que estamos utilizando y la inviabilidad de tal empresa con recursos modestos. Además, alcanzar una precisión tan alta con un conjunto de datos limitado sería prácticamente imposible sin recurrir a técnicas avanzadas de aprendizaje

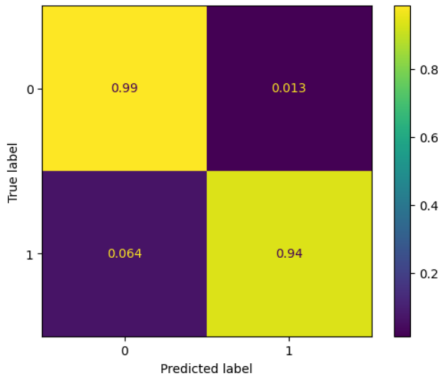


Figura 6.2.1: Matriz de confusión

automático. En este caso específico, estamos utilizando aprendizaje por transferencia con el modelo preentrenado DistilBERT, aplicado al análisis de sentimientos. Estos modelos preentrenados ya poseen una comprensión significativa del lenguaje debido a su entrenamiento previo en grandes corpus de texto, lo que les permite captar matices lingüísticos y contextuales complejos.

Al utilizar un modelo como DistilBERT, podemos ajustar las últimas capas de la red para adaptarlas a nuestras necesidades específicas, en este caso, para diferenciar entre diferentes sentimientos expresados en el texto. Esto nos permite aprovechar el conocimiento preexistente del modelo sobre el lenguaje, ajustando únicamente componentes menores para afinar su rendimiento en tareas específicas. Sin esta base de conocimiento previo, entrenar un modelo desde cero con un conjunto de datos limitado no solo sería impracticable en términos de recursos y tiempo, sino que también correríamos un alto riesgo de sobreajuste (overfitting).

El sobreajuste ocurre cuando un modelo aprende demasiado bien los detalles y el ruido específicos del conjunto de entrenamiento hasta el punto de perder la capacidad de generalizar sobre datos nuevos y no vistos. Esto puede resultar en un modelo que, aunque funcione excepcionalmente bien en los datos de entrenamiento, fallaría al enfrentarse a datos del mundo real o a situaciones fuera de ese conjunto de entrenamiento inicial. Los modelos preentrenados ayudan a mitigar este riesgo al proporcionar una base sólida y generalizada de conocimientos que, con solo pequeños ajustes, se puede adaptar eficazmente a una variedad de tareas sin el peligro inherente de sobreajustar a un conjunto de datos pequeño y potencialmente sesgado.

Parte III

Bibliografía y Anexos

Anexos

C. Backpropagation

El algoritmo de retropropagación es fundamental en el entrenamiento de redes neuronales artificiales y juega un papel crucial en la optimización de sus pesos y sesgos. Desarrollado en la década de 1970 y popularizado por Rumelhart, Hinton y Williams en 1986, este algoritmo ha sido una piedra angular en el éxito de las redes neuronales profundas.

La retropropagación es un método utilizado para calcular eficientemente el gradiente de la función de pérdida de una red neuronal con respecto a cada uno de sus pesos y sesgos. Este gradiente es luego utilizado por algoritmos de optimización, como el descenso del gradiente, para ajustar los parámetros de la red de modo que el error de predicción se minimice.

Denotamos:

- L: número de capas de la red nueronal.
- n_i: número de neuronas en la capa i, donde $i = 1, \ldots, L$.
- $x \in \mathbb{R}^{n_1}$: entrada a la red.
- $y \in \mathbb{R}^{n_L}$: salida objetivo (para la entrada x).
- $g : \mathbb{R}^{n_1} \to \mathbb{R}^{n_L}$: función que simula la red neuronal.
- C: función de pérdida o función de coste.
- $W^l = (w^l_{jk})$: los pesos entre capa l y $l+1$, dónde w^l_{jk} es el peso entre el k-ésimo nodo en la capa l j-ésimo nodo en la capa $l+1$.
- f^l: funciones de activación en la capa l.

En la derivación de la retropropagación se utilizan otras cantidades intermedias; se introducen según sea necesario a continuación. Los bias no reciben un tratamiento especial, ya que corresponden a un peso con una entrada fija de 1. A los efectos de la retropropagación, la función de pérdida específica y las funciones de activación no importan, siempre que estas y sus derivadas puedan evaluarse de manera eficiente. Las funciones de activación tradicionales incluyen, entre otras, sigmoid, tanh y ReLU.

La red general es una combinación de composición de funciones y multiplicación de matrices:

$$g(x) = f^L(W^{L-1} f^{L-1}(W^{L-2} \cdots f^1(W^1 x) \cdots)) \tag{C.1}$$

Para un conjunto de entrenamiento habrá un conjunto de pares de entrada-salida, $\{(x, y) \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_L}\}$. Para cada par de entrada-salida (x, y) en el conjunto de entrenamiento, la pérdida del modelo en ese par es el costo de la diferencia entre la salida proporcionada por la red, $g(x)$ y la salida objetivo y, $C(y, g(x))$. Para ilustrar un posible ejemplo, podemos definir la función de coste con el error cuadrático medio, $C(y, g(x)) := (g(x) - y)^2$.

Pese a parecer que la función de coste depende de la entreda x y la salida deseada y, realmente la variable dependiente en este caso son los pesos ω puesto que queremos ver para qué valores de los pesos se minimiza el error dejando así en cada entrenamiento a x e y como constantes.

Para encontrar el mínimo de la función necesitamos calcular el gradiente de dicha función y por lo tanto encontrar todas las derivadas parciales. Como se puede demostrar / observar, el cálculo individual de todas las derivadas parciales, pese a ser teóricamente posible, necesita de un gasto computacional muy alto. Aquí aparece el algoritmo de Backpropagation en el cual se ayuda de la regla de la cadena reusando cálculos hechos previamente.

La retropropagación se puede expresar para redes simples de retroalimentación en términos de multiplicación de matrices, o más generalmente en términos del gráfico adjunto.

C.1. Algoritmo

En este apartado explicaremos el algoritmo, centrado en un caso básico de una red feedforward, donde los nodos en cada capa están conectados solo a los nodos en la siguiente capa inmediata (sin omitir ninguna capa), y hay una función de pérdida que calcula una pérdida escalar para la salida final, la retropropagación puede ser entendido simplemente por la multiplicación de matrices. Esencialmente, la retropropagación evalúa la expresión de la derivada de la función de costo como un producto de las derivadas entre cada capa de derecha a izquierda, 'hacia atrás', siendo el gradiente de los pesos entre cada capa una simple modificación de los productos parciales.

El objetivo principal de este algoritmo es calcular la derivada parcial de la función de coste respecto a un peso cualquiera de la red, $\frac{\partial C}{\partial \omega^l_{kj}}$. Recordemos primero algunas definiciones:

$$z^{l+1}_k := b^{l+1}_k + \sum_{j=1}^{n_l} \omega^l_{kj} a^l_j \tag{C.2}$$

$$a^l_k := f(z^l_k) \tag{C.3}$$

$$\delta^l_k := \frac{\partial C}{\partial z^l_k} \tag{C.4}$$

$$C := \sum_{i=1}^{n_L} (a^L_i - y_i)^2 = \sum_{i=1}^{n_L} (f(z^L_i) - y_i)^2 \tag{C.5}$$

recordar que la función de coste C no tiene por qué ser la de minimos cuadrados, simplemente la introducimos por el momento para tener una idea más visual. Al ser una función que depende de los pesos, se puede confundir su funcionalidad ya que la notación

completa es $C(\omega_{kj}^l \mid l \in \{1,...,L-1\}, j \in \{1,...,n_l\}, k \in \{1,...,n_{l+1}\})$, la cual puede ser un poco engorrosa.

Tener en cuenta que los sesgos b_k^l también son pesos y tendríamos que calcular sus derivadas parciales, sin embargo pueden calcular igual que los otros simplemente teniendo en cuenta que la entrada de la red será un 1 siempre, por lo que podemos obviar el calculo particular de estas derivadas puesto que no son más que un cálculo particular de los otros.

Aplicando la regla de la cadena tenemos que

$$\frac{\partial C}{\partial \omega_{kj}^l} = \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial \omega_{kj}^l} \overset{(C,2)}{=} \frac{\partial C}{\partial z_k^{l+1}} a_j \overset{(C,4)}{=} \delta_k^{l+1} a_j \tag{C.6}$$

por lo que para calcular la derivada necesitamos calcular los delta. Veamos el delta de la última capa

$$\begin{aligned}
\delta_k^L &:= \frac{\partial C}{\partial z_k^L} \\
&\overset{(C,5)}{=} \frac{\partial}{\partial z_k^L} \left(\sum_{i=1}^{n_L} (a_i^L - y_i)^2 \right) \\
&\overset{(C,3)}{=} \frac{\partial}{\partial z_k^L} \left(\sum_{i=1}^{n_L} (f(z_i^L) - y_i)^2 \right) \\
&= \sum_{i=1}^{n_L} \frac{\partial}{\partial z_k^L} (f(z_i^L) - y_i)^2 \\
&= \frac{\partial}{\partial z_k^L} (f(z_k^L) - y_k)^2 \\
&= 2(f(z_k^L) - y_k) f'(z_k^L) \in \mathbb{R}
\end{aligned} \tag{C.7}$$

que no depende de los pesos. Por otro lado tenemos que

$$z_i^{l+1} = \sum_{s=1}^{n_{l+1}} \omega_{is}^l f(z_s^l) \Rightarrow \frac{\partial z_i^{l+1}}{\partial z_k^l} = \frac{\partial}{\partial z_k^l} \omega_{ik}^l f(z_k^l) = \omega_{ik}^l f'(z_k^l) \tag{C.8}$$

así

$$\begin{aligned}
\delta_k^l &:= \frac{\partial C}{\partial z_k^l} \\
&= \sum_{i=1}^{n_{l+1}} \frac{\partial C}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial z_k^l} \\
&\overset{(C,4)}{=} \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} \frac{\partial z_i^{l+1}}{\partial z_k^l} \\
&\overset{(C,8)}{=} \sum_{i=1}^{n_{l+1}} \left[\delta_i^{l+1} \omega_{ik}^l f'(z_k^l)\right] \\
&= f'(z_k^l) \sum_{i=1}^{n_{l+1}} \left[\delta_i^{l+1} \omega_{ik}^l\right]
\end{aligned} \tag{C.9}$$

Ahora bien puesto que $\delta_k^l = h(\delta_k^{l+1})$ para cierta función h definida en (C,9) y $\delta_k^L \in \mathbb{R}$, ya podemos calcular todos los valores de delta con una simple operación. Y con ello directamente por (C,6) podemos calcular todas las derivadas parciales.

D. Aclaración de $L(\Theta)$

Puede surgir la pregunta de por qué he denotado la función de error como $L(\Theta)$ en lugar de $C(y, \hat{y})$, lo cual parecería razonable para mantener la notación de la sección anterior. La función C denota el error cometido entre dos valores reales, típicamente el valor esperado y y el valor predicho $\hat{y}$. Sin embargo, la función de error en una red neuronal es más compleja que esta simple diferencia.

Para la función de error $L(\Theta)$, asumimos que el valor de salida esperado es fijo, mientras que el valor real de la salida de la red es una expresión completa que depende de los parámetros de la red. Estos parámetros, denotados por Θ, incluyen los pesos y biases de la red. Por lo tanto, $L(\Theta)$ es una función que depende de estos parámetros.

En una red neuronal, los valores de entrada se consideran como variables independientes y los pesos de la red como las variables dependientes que necesitamos ajustar. La función de error $L(\Theta)$ nos permite rastrear cómo los pesos influyen en el rendimiento de la red, facilitando así su modificación para mejorar la precisión del modelo. Mediante el uso de técnicas como la retropropagación [C], podemos actualizar los pesos para minimizar el error y, en consecuencia, mejorar el rendimiento general de la red.

E. Clustering

El *clustering* es una técnica fundamental en el campo del análisis de datos y el aprendizaje automático. Se utiliza para agrupar un conjunto de objetos de manera que los objetos en el mismo grupo (o clúster) sean más similares entre sí que a aquellos en otros grupos. Esta técnica es invaluable en diversas aplicaciones, desde la segmentación de clientes en marketing hasta la detección de patrones en bioinformática.

Matemáticamente, el problema de clustering puede definirse de la siguiente manera: dado un conjunto de n puntos $X = \{x_1, x_2, \ldots, x_n\}$ en un espacio métrico $\mathbb{R}^d$, queremos particionar X en k clústeres $C = \{C_1, C_2, \ldots, C_k\}$ tal que la similitud intra-clúster

se maximice y la similitud inter-clúster se minimice. La similitud entre puntos se mide usualmente mediante una función de distancia.

E.1. Algoritmos de Clustering

Existen varios algoritmos para realizar clustering, entre los cuales los más comunes son:

- K-means
- DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Cada uno de estos algoritmos tiene sus propias características y se aplica en diferentes contextos según la naturaleza de los datos y los objetivos del análisis.

E.2. K-means

El algoritmo K-means es uno de los métodos de clustering más utilizados debido a su simplicidad y eficiencia. Dado un número fijo de clústeres k, el objetivo es minimizar la suma de las distancias cuadradas de cada punto a su centroide correspondiente.

Proceso del Algoritmo

1. **Inicialización:** Seleccionar k puntos aleatorios como centroides iniciales.
2. **Asignación:** Asignar cada punto al clúster cuyo centroide sea el más cercano, usando la distancia euclidiana como métrica.
3. **Actualización:** Calcular los nuevos centroides como la media de los puntos asignados a cada clúster.
4. **Convergencia:** Repetir los pasos de asignación y actualización hasta que los centroides no cambien significativamente entre iteraciones.

Función Objetivo

La función objetivo que K-means minimiza es la suma de las distancias cuadradas dentro de los clústeres:

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

donde μ_i es el centroide del clúster C_i.

E.3. DBSCAN

DBSCAN es un algoritmo de clustering basado en densidad que puede encontrar clústeres de forma arbitraria y manejar ruido en los datos.

Proceso del Algoritmo

1. **Identificación de Puntos Nucleares:** Un punto se considera un núcleo si al menos MinPts puntos están dentro de su vecindario ϵ.
2. **Expansión de Clústeres:** A partir de un punto núcleo, se agregan iterativamente todos los puntos denso-conectados para formar un clúster.
3. **Clasificación de Ruido:** Los puntos que no pertenecen a ningún clúster se clasifican como ruido.

Parámetros Clave

- ϵ: Radio de vecindad.
- MinPts: Número mínimo de puntos en el vecindario para ser considerado un núcleo.

E.4. Evaluación de Clustering

Evaluar la calidad de un clustering es crucial y puede hacerse mediante diversas métricas, tales como:

- **Índice de Dunn:** Mide la compactación y separación de los clústeres.
- **Coeficiente de Silueta:** Evalúa qué tan similar es un punto a su propio clúster en comparación con otros clústeres.
- **Inercia:** Utilizada en K-means, mide la suma de las distancias cuadradas de los puntos a sus centroides.

Códigos

F. Perceptrón simple - Separación lineal

F.1. main.py

```python
from perceptron import Perceptron
import matplotlib.pyplot as plt
import numpy as np
import os
import cv2


# functions
f = lambda x : -2*x + 12
g = lambda x : 1 if x > 0 else -1

def ask_number():
    try:
        n = int(input(" > "))
        return n
    except:
        print("Invalid argument! Try again:")
        ask_number()

def ask_path():
    name = input(" > ")
    if not os.path.exists(f"images/{name}/"):
        os.makedirs(f"images/{name}/")
        os.makedirs(f"images/{name}/frames/")
        return name
    else:
        print("Folder already exists! Please chose an other name:")
        ask_path()

def main():

    # ask for parameters
    print("Number of points:")
    n = ask_number()
    print("Number of epochs:")
    epochs = ask_number()
    print("Step:")
    step = ask_number()
    print("Name to save the video file:")
    name = ask_path()

    # points
    px = 10*np.random.rand(n)
    py = 10*np.random.rand(n)

```

```
    points1 = np.array([(x,y) for x,y in zip(px,py) if y > f(x)])
    points2 = np.array([(x,y) for x,y in zip(px,py) if y <= f(x)])

    # training data
    X = np.array([px, py])
    Y = np.array(list(map(g, (py-np.array(list(map(f, px)))))))

    perceptron = Perceptron(n_inputs=2, lr=0.001)
    imgs = []

    for epoch in range(0,epochs,step):

        perceptron.fit(X, Y, epochs=step)
        p = lambda x : -(perceptron.w[0] * x + perceptron.w0) / perceptron.
    w[1]

        plt.clf()
        plt.title(str(epoch) + " epochs")
        plt.plot(points1[:,0], points1[:,1], "b.")
        plt.plot(points2[:,0], points2[:,1], "r.")
        plt.plot([0,9], [p(0),p(9)], "k-")
        plt.xlim(-1,10)
        plt.ylim(-1,10)

        path = f"images/{name}/frames/{epoch}.jpg"
        plt.savefig(path)
        img = cv2.imread(path)
        imgs.append(img)

    height, width  = img.shape[:2]

    video = cv2.VideoWriter(f'images/{name}/{name}.mp4', cv2.
    VideoWriter_fourcc(*'mp4v'), 5, (width,height))

    for img in imgs:
        video.write(img)

    video.release()




if __name__ == "__main__":
    main()
```

F.2. perceptron.py

```
import numpy as np

class Perceptron:

    def __init__(self, n_inputs, lr):
        self.n = n_inputs
        self.lr = lr
        self.w = np.random.randn(n_inputs)
        self.w0 = np.random.randn(1)

    def forward(self, X):
        pred = np.dot(self.w, X) + self.w0
        return pred

```

```
    def fit(self, X, Y, epochs=1):
        losses = []
        for epoch in range(epochs):
          for i in range(len(X[0])):
            x = X[:,i]
            y = Y[i]
            pred = self.forward(x)
            error = y-pred
            self.w += self.lr * error * x
            self.w0 += self.lr * error
            losses.append(abs(error))
          print(f"[{epoch+1}] last error: {losses[-1]} & mean error: {sum(
    losses)/len(losses)}")
        return losses
```

G. Redes Convolucionales - datos MNIST

G.1. main.py

```
from create_model import create_model
from test_model import test_model



def main():
    print("Chose an option:")
    print(" 1) Train model")
    print(" 2) Test model")
    x = input(" > ")

    if x == "1":
        create_model()
    else:
        test_model()




if __name__ == "__main__":
    main()
```

G.2. create_model.py

```
from matplotlib import pyplot as plt
import torch as T
import os

from nn import Model
from files import save_file, load_file



def load_and_save_data():

    from keras.datasets import mnist
```

```
    (X, Y), (test_X, test_y) = mnist.load_data()

    if not os.path.exists("data"):
        os.mkdir("data")

    save_file(X, "data/train_X")
    save_file(Y, "data/train_y")
    save_file(test_X, "data/test_X")
    save_file(test_y, "data/test_y")

    return X, Y

def create_model():

    # load data MNIST
    if not os.path.exists("data/test_X"):
        X, Y = load_and_save_data()
    else:
        X, Y = load_file("data/train_X"), load_file("data/train_y")

    f = lambda n : [int(i==n) for i in range(10)]
    X = T.tensor(X, dtype=T.float).unsqueeze(1)
    Y = T.tensor(list(map(f, Y)), dtype=T.float)

    # ask for parameters
    print("\nPath to save the model:")
    save_path = input(" > ")
    print("\nNumber of epochs:")
    epochs = int(input(" > "))
    print("\nBatch size:")
    batch_size = int(input(" > "))

    # create & train model
    nn = Model(0.001)
    losses, Ks = nn.fit(X,Y, epochs=epochs, batch_size=batch_size)

    # save results
    if not os.path.exists(f"models/{save_path}"):
        os.mkdir(f"models/{save_path}")

    # plot losses
    plt.subplot(2,1,1)
    k = len(losses) // 100
    y = losses[::k]
    x = range(len(y))
    plt.xticks([])
    plt.plot(x, y)
    plt.title("Training loss")

    plt.subplot(2,1,2)
    k = len(Ks) // 100
    y = Ks[::k]
    x = range(len(y))
    plt.xticks([])
    plt.plot(x, y)
    plt.title("Learning rate modificado (K)")

    plt.savefig(f"models/{save_path}/stats.jpg", dpi=200, bbox_inches='
    tight')
    nn.save_model(f"models/{save_path}/model.pkl")

    # show plot
    plt.show()
```

G.3. test_model.py

```
import torch as T
import numpy as np

from nn import Model
from files import load_file


def test_model():

    # load model
    print("\nModel path:")
    load_path = input(" > ")
    nn = Model(0.001)
    nn.load_model(f"models/{load_path}/model.pkl")

    # load test data
    X_test, y_test = load_file("data/test_X"), load_file("data/test_y")
    X_test = T.tensor(X_test, dtype=T.float).unsqueeze(1).unsqueeze(1).to(
    nn.device)

    # make predictions
    correct = 0
    for x, y in zip(X_test, y_test):
        pred = np.argmax(nn(x)[0].tolist())
        if pred == y:
            correct += 1

    print("Number of predictions:", correct, f" ({100*correct/len(X_test)}
    %)")
```

G.4. nn.py

```
import torch as T
from torch import nn
from torch.nn import functional as F
from torch import optim


class Model(nn.Module):

    def __init__(self, lr):
        super(Model, self).__init__()

        # architecture
        self.layer1 = nn.Conv2d(1,32,3, padding=1)
        self.layer2 = nn.BatchNorm2d(32)
        self.layer3 = nn.ReLU()
        self.layer4 = nn.Conv2d(32,32,3, padding=1)
        self.layer5 = nn.BatchNorm2d(32)
        self.layer6 = nn.ReLU()
        self.layer7 = nn.Conv2d(32,1,3, padding=1)
        self.layer8 = nn.BatchNorm2d(1)
        self.layer9 = nn.ReLU()
        self.layer10 = nn.Flatten()
        self.layer11 = nn.Linear(28*28, 10)
        self.layers = [self.layer1, self.layer2, self.layer3, self.layer4,
    self.layer5, self.layer6, self.layer7, self.layer8, self.layer9, self.
```

```
    layer10, self.layer11]

        # optimizer & loss
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss      = nn.MSELoss()

        # device
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return F.softmax(x, dim=1)

    def fit(self, X, Y, batch_size, epochs):

        n = len(X)
        losses = []
        Ks = []

        c = 0
        for epoch in range(epochs):
            for i in range(0,n,batch_size):
                c += 1

                # get sample
                X_batch = X[i:i+batch_size] if i+batch_size <= n else X[i:]
                Y_batch = Y[i:i+batch_size] if i+batch_size <= n else Y[i:]
                X_batch = X_batch.to(self.device)
                Y_batch = Y_batch.to(self.device)

                # get predictions
                Z_batch = self.forward(X_batch)

                # train
                self.optimizer.zero_grad()
                loss = self.loss(Z_batch, Y_batch).to(self.device)
                losses.append(loss)
                loss.backward()
                self.optimizer.step()

                # Acceso a los gradientes y c lculo de K
                for param in self.parameters():
                    grad = param.grad.data.flatten()[0]  # Gradiente actual
                    state = self.optimizer.state[param]
                    if 'exp_avg' in state and 'exp_avg_sq' in state:
                        exp_avg = state['exp_avg'].flatten()[0]  # m_hat
                        exp_avg_sq = state['exp_avg_sq'].flatten()[0]  # v_hat
                        lr = self.optimizer.param_groups[0]['lr']
                        beta1, beta2 = self.optimizer.param_groups[0]['betas']
                        eps = self.optimizer.param_groups[0]['eps']

                        # C lculo de K para este par metro
                        m_hat = exp_avg / (1 - beta1**(c+1))
                        v_hat = exp_avg_sq / (1 - beta2**(c+1))
                        K = (lr / (T.sqrt(v_hat) + eps)) * m_hat / grad
                    else:
                        K = 0
                    Ks.append(K)
                    break

```

```
            print(f"[{epoch}] -- loss: {losses[-1]} -- mean loss: {sum(losses)/len(losses)}")

        return T.tensor(losses).tolist(), T.tensor(Ks).tolist()

    def save_model(self, path):
        T.save(self.state_dict(), path)

    def load_model(self, path):
        self.load_state_dict(T.load(path))
```

G.5. files.py

```
import pickle


def save_file(data, path):
    file = open(path, "wb")
    pickle.dump(data, file)
    file.close()

def load_file(path):
    file = open(path, "rb")
    data = pickle.load(file)
    file.close()
    return data
```

G.6. ejemplo_layer1.py

```
from nn import Model
import torch as T
from files import load_file
import matplotlib.pyplot as plt
import numpy as np




model = Model(0.001)
model.load_model("models/model_50/model.pkl")

x = load_file("data/test_X")
x = T.tensor(np.array([x[0]]), dtype=T.float).to(model.device)   # shape: torch.Size([1, 28, 28])

y = model.layer1(x)  # shape: torch.Size([32, 28, 28])

def see_layer1():
    for i in range(32):
        plt.subplot(4,8,i+1)
        img = model.layer1.weight[i][0].tolist()
        plt.imshow(img, cmap="gray_r", vmin=-1, vmax=1)
        plt.xticks([])
        plt.yticks([])
    plt.show()

```

```
def see_ej():
    img = x[0].tolist()
    plt.imshow(img, cmap="gray_r", vmin=0, vmax=255)
    plt.xticks([])
    plt.yticks([])
    plt.show()

def see_ej_tras_layer1():
    for i in range(32):
        plt.subplot(4,8,i+1)
        img = y[i].tolist()
        plt.imshow(img, cmap="gray_r", vmin=-1, vmax=1)
        plt.xticks([])
        plt.yticks([])
    plt.show()

def see_ej_tras_layer1_v2():
    for i in range(32):
        plt.subplot(4,8,i+1)
        img = y[i].tolist()
        plt.imshow(img, cmap="gray_r", vmin=0, vmax=255)
        plt.xticks([])
        plt.yticks([])
    plt.show()
```

H. Redes Recurrentes para Predicción de Series Temporales

H.1. main.py

```
# ------------------------------------- GENERACI N DE DATOS ALEATORIA
    -------------------------------------------------

import numpy as np

# Generar datos de precios de ejemplo
np.random.seed(0)
num_datos = 1000
precios = 100 + np.cumsum(np.random.randn(num_datos))

# Funci n de suavizado para simular datos de precios de acciones
def suavizar_datos(datos, ventana=10):
    suavizado = np.convolve(datos, np.ones(ventana)/ventana, mode='valid')
    return suavizado

precios_suavizados = suavizar_datos(precios)

# Visualizar los datos de precios de ejemplo
import matplotlib.pyplot as plt

def plot_grafica():
    plt.plot(precios, label='Datos Originales')
    plt.plot(precios_suavizados, label='Datos Suavizados')
    plt.xlabel('Tiempo')
    plt.ylabel('Precio')
    plt.legend()
    plt.title('Datos de Precios de Acci n')
    plt.show()

```

```
# ------------------------------------ CREAR SECUENCIAS PARA EL MODELO ------------------------------------------------

# Funci n para crear secuencias de datos
def crear_secuencias(datos, longitud_secuencia):
    secuencias = []
    for i in range(len(datos) - longitud_secuencia):
        secuencia = datos[i:i+longitud_secuencia]
        objetivo = datos[i+longitud_secuencia]
        secuencias.append((secuencia, objetivo))
    return secuencias

# Definir la longitud de la secuencia
longitud_secuencia = 24  # Utilizamos 24 horas anteriores para predecir la siguiente

# Crear secuencias de datos
secuencias = crear_secuencias(precios_suavizados, longitud_secuencia)

# Dividir los datos en conjuntos de entrenamiento y prueba
porcentaje_entrenamiento = 0.8
num_entrenamiento = int(porcentaje_entrenamiento * len(secuencias))

datos_entrenamiento = secuencias[:num_entrenamiento]
datos_prueba = secuencias[num_entrenamiento:]

# Convertir los datos a arreglos numpy
X_train, y_train = np.array([x for x, _ in datos_entrenamiento]), np.array([y for _, y in datos_entrenamiento])
X_test, y_test = np.array([x for x, _ in datos_prueba]), np.array([y for _, y in datos_prueba])

# Ajustar las dimensiones de los datos para LSTM (n mero de muestras, longitud de la secuencia, n mero de caracter sticas)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))


# ------------------------------------ CREAR MODELO ------------------------------------------------

from keras.models import Sequential
from keras.layers import LSTM, Dense

# Definir el modelo LSTM
modelo = Sequential()
modelo.add(LSTM(50, input_shape=(longitud_secuencia, 1)))
modelo.add(Dense(1))

# Compilar el modelo
modelo.compile(optimizer='adam', loss='mean_squared_error')

# Entrenar el modelo
modelo.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test), verbose=1)


# ------------------------------------ OBSERVAR RESULTADOS ------------------------------------------------

# Hacer predicciones
predicciones = modelo.predict(X_test)

# Visualizar los resultados
plt.plot(y_test, "k-", label='Datos Reales')
```

```
plt.plot(predicciones, "r-", label='Predicciones')
plt.xlabel('Tiempo')
plt.ylabel('Precio')
plt.legend()
plt.title('Predicciones de Precio de Acci n')
plt.show()
```

I. Clasificación de Imágenes con ImageNet: Gatos vs Perros

I.1. main.py

```
from keras.applications import MobileNetV2
from keras.models import Sequential
from keras.layers import Dense, GlobalAveragePooling2D
from keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.applications.mobilenet_v2 import preprocess_input,
    decode_predictions
import numpy as np
import os

# Cargar el modelo MobileNetV2 preentrenado
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape
    =(224, 224, 3))  # Excluir las capas finales y especificar el tama o
    de entrada

# Congelar todas las capas del modelo preentrenado
for layer in base_model.layers:
    layer.trainable = False

# A adir capas adicionales para la clasificaci n binaria de gato o perro
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')  # Capa final con activaci n sigmoide
    para la clasificaci n binaria
])

# Compilar el modelo
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy'])

def predict(img_path):
    # Cargar una imagen de archivo
    img = image.load_img(img_path, target_size=(224, 224))  # Redimensionar
     la imagen al tama o que espera MobileNetV2
    # Convertir la imagen a un array numpy y preprocesarla
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    # Realizar la predicci n
    preds = model.predict(x)
    return 'Gato' if preds[0][0] < 0.5 else 'Perro'  # Si la probabilidad
    predicha es mayor a 0.5, se considera un gato; de lo contrario, un
    perro


```

```
#
    ------------------------------------------------------------------------------------------------

# ---------------------------------- ENTRENAMIENTO
    ----------------------------------------
#
    ------------------------------------------------------------------------------------------------


dataset_path = "D:\TFG\imagenes\dogcat"

def entrenar_modelo(modelo, directorio_entrenamiento, directorio_validacion
    , epochs=10, batch_size=32):
    # Configurar generadores de datos para entrenamiento y validaci n con
    aumento de datos
    train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
    )

    test_datagen = ImageDataGenerator(rescale=1./255)

    train_generator = train_datagen.flow_from_directory(
        directorio_entrenamiento,
        target_size=(224, 224),
        batch_size=batch_size,
        class_mode='binary'  # Especificar la clase binaria (0 para gato, 1
     para perro)
    )

    validation_generator = test_datagen.flow_from_directory(
        directorio_validacion,
        target_size=(224, 224),
        batch_size=batch_size,
        class_mode='binary'  # Especificar la clase binaria (0 para gato, 1
     para perro)
    )

    # Entrenar el modelo
    modelo.fit(
        train_generator,
        steps_per_epoch=train_generator.samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=validation_generator.samples // batch_size
    )

import pickle

# Guardar los pesos en un archivo utilizando pickle
def guardar_pesos(pesos, nombre_archivo):
    with open(nombre_archivo, 'wb') as f:
        pickle.dump(pesos, f)

"""
entrenar_modelo(model, os.path.join(dataset_path, "train"), os.path.join(
    dataset_path, "train"))
guardar_pesos(model.layers[-2].get_weights(), "pesos_dense_1.pkl")
guardar_pesos(model.layers[-1].get_weights(), "pesos_dense_2.pkl")
"""
```

I.2. test.py

```
from keras.applications import MobileNetV2
from keras.models import Sequential
from keras.layers import Dense, GlobalAveragePooling2D
from keras.preprocessing import image
from keras.applications.mobilenet_v2 import preprocess_input,
    decode_predictions

import pickle, os
import numpy as np


# Cargar los pesos desde un archivo utilizando pickle
def cargar_pesos(nombre_archivo):
    with open(nombre_archivo, 'rb') as f:
        pesos = pickle.load(f)
    return pesos

# Cargar el modelo MobileNetV2 preentrenado
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape
    =(224, 224, 3))  # Excluir las capas finales y especificar el tama o
    de entrada

# Congelar todas las capas del modelo preentrenado
for layer in base_model.layers:
    layer.trainable = False

# A adir capas adicionales para la clasificaci n binaria de gato o perro
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')  # Capa final con activaci n sigmoide
    para la clasificaci n binaria
])

model.layers[-2].build(input_shape=(None, 1280))
model.layers[-2].set_weights(cargar_pesos("model/pesos_dense_1.pkl"))

model.layers[-1].build(input_shape=(None, 128))
model.layers[-1].set_weights(cargar_pesos("model/pesos_dense_2.pkl"))


def predict(img_path):
    # Cargar una imagen de archivo
    img = image.load_img(img_path, target_size=(224, 224))  # Redimensionar
     la imagen al tama o que espera MobileNetV2
    # Convertir la imagen a un array numpy y preprocesarla
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    # Realizar la predicci n
    preds = model.predict(x)
    return 'Gato' if preds[0][0] < 0.5 else 'Perro'  # Si la probabilidad
    predicha es mayor a 0.5, se considera un gato; de lo contrario, un
    perro

# c = lambda x : os.path.join("D:\TFG\imagenes\dogcat\\test\cat",
    "1"+("0000"+str(x))[-4:]+".jpg")
# d = lambda x : os.path.join("D:\TFG\imagenes\dogcat\\test\dog",
    "1"+("0000"+str(x))[-4:]+".jpg")
```

J. Análisis de sentimientos con BERT

J.1. sentimental_analysis.ipynb

Initialize Google Colab

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

In [ ]:
```
main_path = "/content/drive/MyDrive/Colab Notebooks/"
file = "SMSSpamCollection.txt"
```

Load data and make columns: *text* and *label*

In [ ]:
```
import pandas as pd
import os

df = pd.read_csv(os.path.join(main_path, "data", file), sep='\t',
                           names=["category", "text"])
df["label"] = df["category"].astype("category").cat.codes
df.head()
```

Out[ ]:

	category	text	label
0	ham	Go until jurong point, crazy.. Available only ...	0
1	ham	Ok lar... Joking wif u oni...	0
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	1
3	ham	U dun say so early hor... U c already then say...	0
4	ham	Nah I don't think he goes to usf, he lives aro...	0

In [ ]:
```
df.shape
```

Out[ ]:
```
(5572, 3)
```

In [ ]:
```
X=list(df['text'])
y=list(df['label'])
```

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, ran
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size = 0.5
```

Load tokenizer and Bert model

In [ ]:
```
!pip install transformers
```

In [ ]:
```
import transformers
print(transformers.__version__)
```

4.30.2

In [ ]:
```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
```

In [ ]:
```
from transformers import TFDistilBertForSequenceClassification
model = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-
```

Preprocess data

In [ ]:
```
train_encodings = tokenizer(X_train, truncation=True, padding=True)
test_encodings = tokenizer(X_test, truncation=True, padding=True)
val_encodings = tokenizer(X_val, truncation=True, padding=True)
```

In [ ]:
```
import tensorflow as tf

train_dataset = tf.data.Dataset.from_tensor_slices((
    dict(train_encodings),
    y_train
))

test_dataset = tf.data.Dataset.from_tensor_slices((
    dict(test_encodings),
    y_test
))

val_dataset = tf.data.Dataset.from_tensor_slices((
    dict(val_encodings),
    y_val
))
```

Fine-tuned Bert model

In [ ]:
```
model.layers[0].trainable = False
model.layers[1].trainable = False
```

In [ ]:
```
model.summary()
```

```
Model: "tf_distil_bert_for_sequence_classification"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 distilbert (TFDistilBertMai  multiple                 66362880
 nLayer)

 pre_classifier (Dense)      multiple                  590592

 classifier (Dense)          multiple                  1538

 dropout_19 (Dropout)        multiple                  0

=================================================================
Total params: 66,955,010
Trainable params: 1,538
Non-trainable params: 66,953,472
_________________________________________________________________
```

We are going to train only 1,538 parameters. It's very low knowing the task that is been given, but because the Bert architechture part takes care of understanding the lenguage, maybe we dont need much study just to make a simple binary separation.

The fact that i'm choosing to train only those parameters is also because of the trainning time.

In [ ]:
```
model.compile(optimizer="adam")
```

In [ ]:
```
model.fit(train_dataset, validation_data=test_dataset, epochs=3, batch_size=64)
```

```
Epoch 1/3
4179/4179 [==============================] - 1442s 345ms/step - loss: 0.1396 - va
l_loss: 0.0739
Epoch 2/3
4179/4179 [==============================] - 1434s 343ms/step - loss: 0.0734 - va
l_loss: 0.0596
Epoch 3/3
4179/4179 [==============================] - 1456s 348ms/step - loss: 0.0645 - va
l_loss: 0.0587
```

Out[ ]: <keras.callbacks.History at 0x7f6fed5afa60>

Save tokenizer and Model

In [ ]:
```
savepath = "/content/drive/MyDrive/Colab Notebooks/models/sentiment_analysis"
```

In [ ]:
```
model.save_pretrained(savepath)
tokenizer.save_pretrained(savepath)
```

Out[ ]:
```
('/content/drive/MyDrive/Colab Notebooks/ABRICOT/Telcel/NLP/models/sentiment_analysis/tokenizer_config.json',
 '/content/drive/MyDrive/Colab Notebooks/ABRICOT/Telcel/NLP/models/sentiment_analysis/special_tokens_map.json',
 '/content/drive/MyDrive/Colab Notebooks/ABRICOT/Telcel/NLP/models/sentiment_analysis/vocab.txt',
 '/content/drive/MyDrive/Colab Notebooks/ABRICOT/Telcel/NLP/models/sentiment_analysis/added_tokens.json',
 '/content/drive/MyDrive/Colab Notebooks/ABRICOT/Telcel/NLP/models/sentiment_analysis/tokenizer.json')
```

In [ ]:
```
```

Predictions

In [ ]:
```
from transformers import TextClassificationPipeline
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np
```

In [ ]:
```
pipeline = TextClassificationPipeline(model=model, tokenizer=tokenizer)
```

In [ ]:
```
pred = pipeline(X_test)
```

In [ ]:
```
print(pred[0])
int(pred[0]["label"][-1])
```

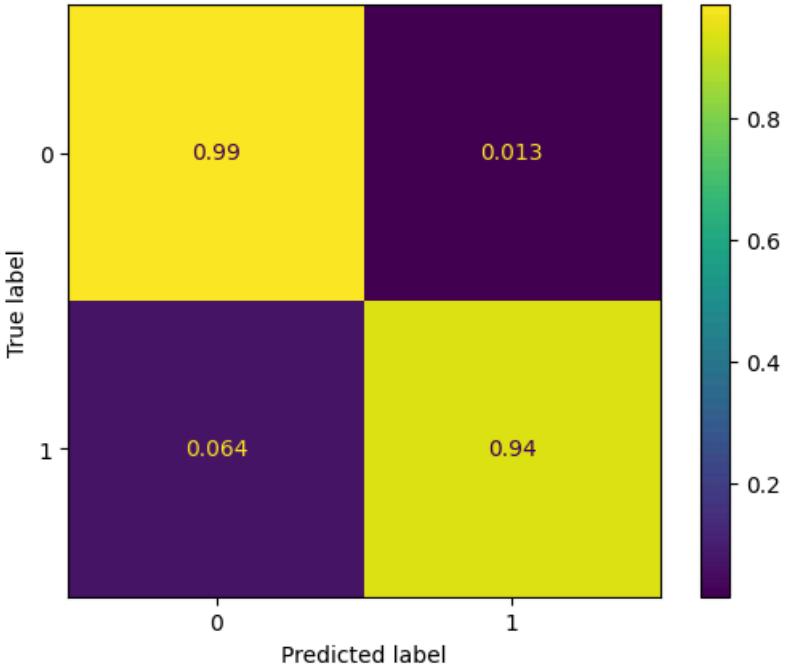
```
{'label': 'LABEL_0', 'score': 0.8320313096046448}
```

Out[ ]:
```
0
```

In [ ]:
```
pred = [int(p["label"][-1]) for p in pred]
```

In [ ]:
```
cm = confusion_matrix(y_test ,pred)
cm2 = np.array([cm[i]/cm[i].sum() for i in range(len(cm))])

cm_display = ConfusionMatrixDisplay(cm2).plot()
cm
```

Out[ ]:
```
array([[594,   8],
       [  6,  88]])
```



We end up with more than 95% accuracy, so they are good results!

Bibliografía

[1] Christopher M Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[2] Jason Brownlee. *Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python*. Machine Learning Mastery, 2019.

[3] Francois Chollet. *Deep Learning with Python*. Manning Publications, 2017.

[4] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". En: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2009, págs. 248-255.

[5] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". En: *arXiv preprint arXiv:1810.04805* (2019).

[6] Xavier Glorot y Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". En: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 2010, págs. 249-256.

[7] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1998.

[9] John Hertz, Anders Krogh y Richard G Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[10] Ryan Nash Keiron O'Shea. "An Introduction to Convolutional Neural Networks". En: *arXiv preprint arXiv:1511.08458* (2015).

[11] Yu Li y Qiang Yang. "Meta-Learning: A Survey". En: *arXiv preprint arXiv:1706.04402* (2017).

[12] Christopher D. Manning e Hinrich Schütze. "Foundations of Statistical Natural Language Processing". En: MIT Press, 1999. Cap. Tokenization, págs. 124-129.

[13] Seymour Papert Marvin Minsky. *Perceptrons: an introduction to computational geometry*. 1969.

[14] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[15] Sinno Jialin Pan y Qiang Yang. "A Survey on Transfer Learning". En: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), págs. 1345-1359.

[16] Eric Rothstein Morris Ralf C. Staudemeyer. "– Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks". En: *arXiv preprint arXiv:1909.09586* (2019).

[17] Stefan Güttel Roberto Cahuantzi Xinye Chen. "A comparison of LSTM and GRU networks for learning symbolic sequences". En: *arXiv preprint arXiv:2107.02248* (2023).

[18] Sebastian Ruder. "An overview of gradient descent optimization algorithms". En: *arXiv preprint arXiv:1609.04747* (2016).

[19] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". En: *arXiv preprint arXiv:1910.01108* (2019).

[20] Lisa Torrey y Jude Shavlik. "Transfer Learning". En: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques.* IGI Global, 2010, págs. 242-264.

[21] Ashish Vaswani et al. "Attention is All You Need". En: *arXiv preprint arXiv:1706.03762* (2017).

[22] Christopher J.C. Burges Yann LeCun Corinna Cortes. "The MNIST Handwritten Digit Database". En: *yann.lecun.com* (2020).