Redes neuronales artificiales.

22 de mayo de 2025

Redes neuronales

Un poco de historia

Las Redes Neuronales Artificiales (o **ANNs por sus siglas en inglés *Artificial Neural Networks***) fueron acuñadas por primera vez en 1943 por el neurofisiólogo Warren McCulloch y el matemático Walter Pitts, en su artículo llamado, *A Logical Calculus of Ideas Immanent in Nervous Activity*. En él, McCulloh y Pits presentan una idea simplificada de cómo las neuronas deben trabajar juntas para resolver cálculos usando lógica proposicional, dando lugar a la primera arquitectura de redes neuronales artificiales. El interés en estas redes se perdió después de los 60's y no se recuperó hasta hace relativamente poco, cuando en 1990 se consideraron como una poderosa alternativa para usar en Machine Learning.

Pero, ¿se perderá el interés nuevamente? ¿o capturarán el interés de la comunidad científica y tendrán gran impacto en nuestra vida diaria? En realidad ya están presentes en nuestras vidas cotidianas, pero podemos pensar que estas tendrán gran influencia en la tecnología por venir debido a varias razones: cada vez más hay una gran cantidad de datos para poder entrenar ANNs, además el gran poder de cómputo que hay actualmente permite entrenar modelos largos y complejos de redesy gracias a esto los algoritmos han sido mejorados hasta alcanzar niveles de eficiencias nunca antes vistos.[1]

Machine Learning

Desde la década pasada, la inteligencia artificial se ha vuelto un tema popular dentro y fuera de la comunidad científica, debido a la abundancia de artículos de tecnología, los journals se han cubierto de los temas **Machine Learning** (**ML**), **Deep Learning** (**DL**) e **Inteligencia Artificial** (**IA**). Estos términos están fuertemente asociados, pero no son lo mismo estrictamente. En términos simplificados, IA es un campo enfocado en automatizar tareas intelectuales realizadas normalmente por humanos, y Machine Learning y Deep Learning son métodos específicos para alcanzar este propósito, es decir, son *ramas* de IA, que a su vez está contenida en el campo de Data Science.

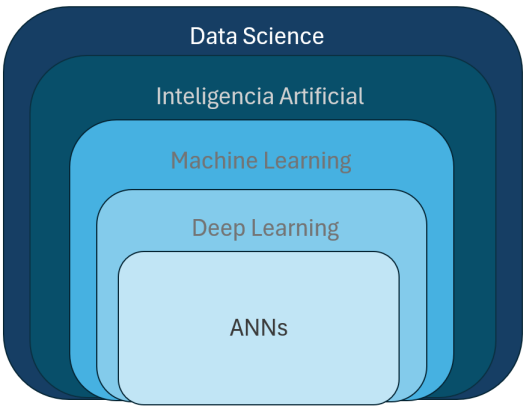

Figura 1: Diagrama ilustrativo de técnicas de Data Science. Data Science incluye a la IA. IA contiene programación clásica y ML, a su vez, Ml contiene varios modelos y métodos, entre ellos, DL y ANN

Aunque el concepto de Machine Learning pudiera parecer alguna fantasía y suene muy alejado del

presente, es algo que ya está en nuestras vidas diarias. La primera herramiento de **ML** fue creada en la década de los 90s del siglo 20, y se trataba de un filtro de spam para el correo electrónico.

Pero, ¿qué es Machine Learning realmente? El ML es la ciencia de programar computadoras para que puedan *aprender* por su cuenta utilizando datos, este campo de estudio da a las computadoras la habilidad de aprender un tema (o habilidades) sin ser explícitamente programadas para ellas. Esto no quiere decir que las computadoras tienen conciencia propia o algo por el estilo, sino que, citando a Tom M. Mitchell, un pionero en el campo de la IA y el ML, una computadora aprende de la experiencia E con respecto a una tarea T y un medidor de performance P, si su performance en T (medido por P) mejora con la experiencia E.

ANNs

Los grandes inventos de la historia de la humanidad han sido inspirados por la naturaleza, así como se encontró inspiración en las aves para hacer aviones, en los pulpos para crear cohetes y en plantas para crear el velcro, resulta lógico poner nuestra atención en el funcionamiento y la arquitectura del cerebro a la hora de crear máquinas inteligentes. Sin embargo, al igual que con otros inventos inspirados en la naturaleza, las Redes Neuronales Artificiales no son exactamente igual que las redes neuronales que existen en el cerebro. Siendo más claros, una ANN es un modelo matemático que consta de capas de unidades de procesamiento donde cada capa toma la salida de la capa anterior y aplica una transformación para producir una salida final. Estas transformaciones son aprendidas a partir de datos.[22]

Las ANNs son el centro del DL debido a su versatilidad, potencia y escalabilidad, lo que las hacen herramientas ideales para resolver problemas grandes y complejos, tales como clasificar miles de millones de imágenes como lo hace Google Images, recomendar los videos más adecuados a cientos de millones de usuarios de YouTube o aprender a derrotar al campeón mundial de Go.[2]. Algunas de las tareas principales que desarrollan son: regresiones, clasificaciones y agrupamiento. Al ser variadas las tareas, y trabajar con distintos tipos de datos, es útil que existan varias clases de redes, cada una de ellas pensada para llevar a cabo cierto tipo de tareas de la manera más óptima.

Arquitectura de una red neuronal

Las redes neuronales están conformadas por elementos que es importante entender, uno de ellos son los **nodos**, la unidad de procesamiento fundamental en la red neuronal. Cuando tenemos varios nodos, se forman capas, es decir, un conjunto de nodos con las que se procesan los datos.

Una red neuronal está compuesta de tres capas principalmente, una es la **capa de entrada**, en ella la información que queremos procesar entra a la ANN los **nodos de entrada** procesan los datos, los analizan y los comunican a la siguiente parte, la **capa oculta** (pueden ser una gran catidad de capas ocultas, no necesariamente una), en la cual analiza la salida de la capa anterior, la procesa y pasa a la última capa. En la **capa de salida** se proporciona un resultado final de todo el procesamiento de los datos que se han realizado en las anteriores partes. En problemas de clasificación de dos alternativas, habrá un **nodo de salida** (como 0 y 1 por ejemplo), sin embargo, en problemas con más de dos alternativas habrán más nodos. [7]

Tipos de redes neuronales según el tipo de flujo de datos.

Redes neuronales pre-alimentadas

Las redes neuronales pre-alimentadas procesan los datos en una dirección, desde el nodo de entrada hasta el nodo de salida. Todos los nodos de una capa están conectados a todos los nodos de la capa siguiente. Este tipo de redes utilizan un proceso de retroalimentación para mejorar las predicciones a lo largo del tiempo.

Algoritmo de retro-propagación

Las redes neuronales artificiales aprenden de forma continua mediante el uso de bucles de retroalimentación de corrección para mejorar su análisis predictivo. En pocas palabras, puede pensar en los datos que fluyen desde el nodo de entrada hasta el nodo de salida a través de muchos caminos diferentes en la red neuronal. Solo un camino es el correcto: el que asigna el nodo de entrada al nodo de salida correcto. Para encontrar este camino, la red neuronal utiliza un bucle de retro-alimentación en donde cada nodo intenta adivinar el siguiente nodo de la ruta, y después se comprueba si la suposición es correcta. Los nodos asignan valores de peso más altos a las rutas que conducen a más suposiciones correctas y valores de peso más bajos a las rutas de los nodos que conducen a suposiciones incorrectas. Para el siguiente punto de datos, los nodos realizan una predicción nueva con las trayectorias de mayor peso y luego repiten el proceso.

Redes neuronales convolucionales

Las redes neuronales convolucionales (**CNNs** por sus siglas Convolutional Neural Networks) son una clase de redes neuronales artificiales de retro-alimentación que se aplican principalmente al análisis de imágenes. [23]

Las capas de estas redes realizan funciones matemáticas específicas, como la síntesis o el filtrado, denominadas convoluciones. Son muy útiles para la clasificación de imágenes debido a que pueden extraer características relevantes de ellas que son útiles para su reconocimiento y clasificación. Cada capa oculta extrae y procesa diferentes características de la imagen, como los bordes, el color y la profundidad. Una vez filtradas las imágenes, estas son más fáciles de procesar sin perder características que son fundamentales para hacer un correcto análisis que nos lleva a una buena predicción.

En los últimos años ha aumentado fuertemente el número de investigaciones que implementa redes neuronales convolucionales para resolver una amplia variedad de problemas. Por ejemplo, en el año 2019 se implementaron CNN en el área de cosmología para estimar parámetros cosmológicos con datos extraídos directamente de simulaciones de las distribuciones de materia oscura y de fotones en el fondo cósmico de microondas. [24]

También, en ese mismo año se construyeron CNNs que predecían la formación estructural no lineal del universo utilizando teoría de perturbaciones.

Otro campo en el que se ha buscado implementar CNNs a lo largo d los años es el de la medicina. Fijémonos en el estudio del cáncer de piel. En 2016, se propuso un modelo basado en una segmentación de imágenes de melanoma basada en la triangulación de Delaunay. Los resultados finales mostraron una sensibilidad del 93,5 %. [27] Después, en 2018 se propuso un método basado en la detección de este cáncer utilizando características morfológicas para la detección del melanoma, con el cual se pudo determinar un sistema de diagnóstico asistido por computadora para la detección del cáncer de piel. [28] Poco después, en 2019 se propuso un algoritmo llamado *whale algorithm* para optimizar CNNs especializadas en la detección de este padecimiento. [26]

En la última década la construcción de ANNs han tenido un marcable éxito debido al desarrollo de herramientas que facilitan y optimizan su desarrollo. Dos de estas herramientas son **PyTorch** y **TensorFlow**.[3]

Redes difusas

El funcionamiento de estas redes se basa en la lógica difusa. Este tipo de lógica agrupa valores de parámetros que cumplen con ciertas características. Supongamos que se tiene un sensor de temperatura, con un microcontrolador podemos encender un ventilador pasando de 25°C, lo que podemos hacer es agrupar a los valores de temperatura arriba de los 25°C y asignarles un 0, y a los valores abajo de ella se les asigna un 1. Esto es parecido a la lógica booleana pero usando agrupaciones de parámetros en lugar de valores puntuales. La operación principal de este tipo de redes es un mapeo de los nodos y sus características a los resultados de un proceso de difusión que comienza en ese nodo. A diferencia de las Redes Convolucionales estándar, los parámetros de las DCNN (Difusion Convolution Neural Network) están vinculados según la profundidad de búsqueda de difusión en lugar de su posición en una cuadrícula. La representación convolucional de difusión es invariante con respecto al índice del nodo en lugar de la posición; en otras palabras, las activaciones convolucionales de difusión de dos gráficos de entrada isomórficos serán las mismas.

Modelos generativos

Uno de los objetivos más esperados por los desarrolladores de modelos de ML es dotar a las computadoras de algoritmos que les permitan entender nuestro mundo. Los modelos generativos parecen ser el enfoque correcto para poder lograr este objetivo. Dentro de esta categoría existen ciertos tipos de enfoque, cada uno de ellos pensado para lograr el objetivo mencionado de diferente manera. [8]

Variational Autoencoders (VAEs)

En los últimos años, los VAEs (Autocodificadotres Variacionales, por sus siglas en inglés) se han vuelto de los enfoques más populares en el aprendizaje no supervisado para distribuciones complejas.[9] Estas mezclan modelos de Redes Neuronales con distribuciones de probabilidad. Su principal uso es reproducir cierto tipo de datos deseado que sean consistentes con el conjunto de datos con el que se entrena a la red. Un uso muy popular de las redes que usan este enfoque, es el de crear imágenes semejantes, como rostros y objetos. Este método se puede usar siempre y cuando e conjunto de datos sea lo suficientemente grande y cumpla con ciertas características.

Figura 2: Ejemplo de uso popular de las VAEs. Imagen tomada de: https://www.cs.us.es/ fsancho/Blog/posts/VAE.md, 10/05/2024.

Generative Adversial Networks (GANs)

Este tipo de modelos plantea un juego entre dos redes separadas: un **generador** y un **discriminador**. El generador produce datos que intentan parecerse a los datos del entrenamiento. Una vez que el discriminador nota una diferencia entre dos distribuciones el generador ajusta sus parámetros tenuemente para producir una nueva salida. Esto en principio, se produce hasta emparejar de manera exacta ambos conjuntos de datos.

Hasta ahora parece que las GANs y las VAEs son exactamente lo mismo, sin embargo, no olvidemos que las VAEs se basan en teoría probabilística para su funcionamiento. Cada uno de estos enfoques tiene sus pros y sus contras.[8] Las VAEs permiten ejecutar tanto el aprendizaje como inferencia Bayesiana en sofisticados modelos probabilísticos gráficos con variables latentes. Sin embargo, las muestras que crea tienden a ser borrosas. Por otra parte, las GANs a menudo generan imágenes más definidas, pero son difíciles de optimizar debido a su método de entrenamiento. [10]

PyTorch

Desarrollado originalmente por Facebook AI Research (ahora Meta), Python se convirtió en código abierto desde 2017 y ha estado bajo la administración de la Fundación PyTorch desde 2022. La fundación sirve de espacio neutral para que la comunidad de DL colabore en el desarrollo del ecosistema de PyTorch. PyTorch es un framework creado para desarrollar DL. Este es de código abierto basado en software utilizado para crear redes neuronales, combinando la biblioteca de aprendizaje automático back-end de Torch con una API de alto nivel basada en Python. Su flexibilidad y facilidad de uso, entre otras ventajas, lo han convertido en uno de los principales marcos de Machine Learning utilizados por comunidades académicas y de investigación.

PyTorch admite una amplia variedad de estructuras de redes neuronales, desde simples algoritmos de regresión lineal hasta complejas redes neuronales convolucionales y modelos transformadores generativos utilizados para tareas como la visión artificial y el procesamiento de lenguaje natural (**NLP** por sus siglas ***Natural Language Processing***). PyTorch, que se basa en el lenguaje de programación Python, ofrece amplias bibliotecas de modelos pre-configurados (e incluso pre-entrenados), permitiendo a los científicos de datos crear y ejecutar sofisticadas redes de aprendizaje profundo minimizando el tiempo y el trabajo dedicados al código y la estructura matemática. PyTorch también permite a los científicos de datos ejecutar y probar partes del código en tiempo real, en lugar de esperar a que se implemente todo el código, lo cual, para los grandes modelos de aprendizaje profundo, puede llevar mucho tiempo de cómputo.

Funcionamiento de PyTorch

En cualquier algoritmo de aprendizaje automático, incluso los aplicados a información aparentemente no numérica como sonidos o imágenes, los datos deben representarse numéricamente. En PyTorch, esto se consigue a través de tensores, que sirven como unidades fundamentales de datos utilizadas para el cálculo en la plataforma.

En el contexto del aprendizaje automático, un tensor es una matriz multidimensional de números que funciona como un dispositivo de contabilidad matemática. Los tensores de PyTorch funcionan de manera similar a las matrices utilizadas en Numpy, pero a diferencia de las matrices, que solo pueden ejecutarse en unidades centrales de procesamiento (CPU), los tensores también pueden ejecutarse en unidades de

procesamiento gráfico (GPU). Las GPU permiten realizar cálculos mucho más rápidos que las CPU, lo que supone una gran ventaja dados los enormes volúmenes de datos y el procesamiento paralelo típicos del aprendizaje profundo.

Además de codificar las entradas y salidas de un modelo, los tensores de PyTorch también codifican los parámetros del modelo: los pesos, sesgos y gradientes que se "aprenden" en el aprendizaje automático. Esta propiedad de los tensores permite la diferenciación automática, que es una de las características más importantes de PyTorch.

Módulos

En términos generales, hay tres clases principales de módulos utilizados para construir y optimizar modelos de DL en PyTorch:

- Los módulos nn se implementan como las capas de una red neuronal. El paquete torch.nn contiene una amplia biblioteca de módulos que realizan operaciones comunes como convoluciones, agrupaciones y regresiones. Por ejemplo, torch.nn.Linear(n,m) denomina un algoritmo de regresión lineal con entradas n y salidas m (cuyas entradas y parámetros iniciales se establecen en líneas de código posteriores).
- El módulo autograd proporciona una forma sencilla de calcular automáticamente los gradientes, utilizados para optimizar los parámetros del modelo mediante el descenso del gradiente, para cualquier función gestionada dentro de una red neuronal. Agregar cualquier tensor con require_grad=True indica a autograd que cada operación en ese tensor debe rastrearse, lo que permite la diferenciación automática.
- Los módulos Optim aplican algoritmos de optimización a esos gradientes. Torch.optim proporciona módulos para varios métodos de optimización, como el descenso del gradiente estocástico (SGD) o la propagación de la raíz cuadrada media (RMSprop), para adaptarse a necesidades específicas de optimización.

Gráficas del cálculo dinámico

Las gráficas de cálculo dinámico (DCG) son la forma en que se representan los modelos de aprendizaje profundo en PyTorch. En términos abstractos, las gráficas de cálculo mapean el flujo de datos entre las diferentes operaciones de un sistema matemático: esencialmente traducen el código de una red neuronal en un diagrama de flujo que indica las operaciones realizadas en cada nodo y las dependencias entre los diferentes niveles de la red (la disposición de los pasos y secuencias que transforman los datos de entrada en datos de salida).

Lo que diferencia a las gráficas de cálculo dinámico (como los utilizados en PyTorch) de los gráficos de cálculo estáticos (como los utilizados en TensorFlow) es que las DCG aplazan la especificación exacta de los cálculos y las relaciones entre ellos hasta el momento de la ejecución. En otras palabras, mientras que una gráfica de cálculo estático requiere que la estructura de toda la red neuronal esté completamente determinada y compilada para poder ejecutarse, los DCG pueden repetirse y modificarse sobre la marcha. De esta manera partes específicas del código de un modelo pueden modificarse o ejecutarse de forma aislada sin tener que reiniciar todo el modelo, lo que, en el caso de los modelos de aprendizaje

profundo de gran tamaño utilizados para tareas sofisticadas de visión artificial y procesamiento de lenguaje natural (NLP), puede suponer una pérdida tanto de tiempo como de recursos informáticos. Las ventajas de esta flexibilidad se extienden al entrenamiento de modelos, ya que las gráficas de cálculo dinámico se generan fácilmente a la inversa durante la retropropagación.

Diferenciación automática

Un método muy utilizado para entrenar redes neuronales, sobre todo en el aprendizaje supervisado, es la retropropagación. En primer lugar, en un paso hacia delante, se alimenta a un modelo con algunas entradas x y se predicen algunas salidas y; trabajando hacia atrás desde esa salida, se utiliza una función de pérdida para medir el error de las predicciones del modelo en diferentes valores de x. Al diferenciar esa función de pérdida para encontrar su derivado, se puede usar descenso del gradiente para ajustar los pesos en la red neural, un nivel a la vez.

El módulo **autograd** de PyTorch potencia su técnica de diferenciación automática usando la regla de la cadena, en la cual se calcula los derivados complejos, los divide en derivados más simples y luego los combina. Autograd calcula y registra automáticamente los gradientes de todas las operaciones ejecutadas en un gráfico de cálculo, lo que reduce enormemente el trabajo de la retropropagación.

Cuando se ejecuta un modelo que ya se ha entrenado, el autograd se convierte en un uso innecesario de recursos de cálculo. Agregar cualquier operación tensorial con requires_grad=False indicará a PyTorch que deje de rastrear gradientes. [4]

Algunas herramientas en el ecosistema PyTorch

- **Torchvision** es un conjunto de herramientas que contiene módulos, estructuras de red y conjuntos de datos para diversas tareas de clasificación de imágenes, detección de objetos y segmentación de imágenes.
- **TorchText** proporciona recursos como conjuntos de datos, transformaciones básicas de procesamiento de textos y modelos pre-entrenados para su uso en el procesamiento de lenguaje natural (NLP).

TensorFlow

Creado por el equipo de Google Brain, TensorFlow es un framework de código abierto hecho para el desarrollo de computación numérica y Machine Learning a gran escala. Su nombre se debe a que opera sobre ***tensores***, que son básicamente matrices multidimensionales. TensorFlow reúne una serie de modelos y algoritmos de Machine Learning y Deep Learning. TensorFlow toma cálculos descritos usando un modelo de tipo de flujo de datos y los asigna en una gran variedad de diferentes plataformas de hadware, corriendo en aparatos móviles con Android o iOS, así como en sistemas de entrenamiento de que usan una sola máquina de uno pocos más GPUs hasta sistemas de entrenamiento de larga escala, corriendo en cientos de máquinas especializadas con miles de GPUs. La gran versatilidad de TensorFlow simplifica de manera significativa en el mundo real el uso de sistemas de ML, pues tenemos sistemas separados de modelos de entrenamiento de larga escala así como algunos de corta escala.[5]

Los cálculos de TensorFlow son expresados de manera visual como gráficas de flujo de datos con un estado, y está enfocado en hacer el sistema lo suficientemente flexible para experimentar de manera

rápida con nuevos modelos con propósito de investigación, así como tener un alto performance para poder entrenar y desplegar modelos de machine learning de manera productiva.

Para escalar ANNs a un número alto de despliegues, TensorFlow permite a los usuarios aprovechar diferentes tipos de paralelismo de manera sencilla. Esto se logra replicando y ejecutando al mismo tiempo el modelo en varios dispositivos, que colaboran para actualizar un conjunto de parámetros compartidos.

TensorBoard

TensorBoard es una herramienta de TensorFlow creada para proporcionar la visualización y dentro de él, más herramientas necesarias para experimentar con el aprendizaje automático. Algunas gráficas que permite visualizar ayudan a seguir métricas tales como la pérdida y la exactitud de un modelo dado. Otra cosa importante es que nos permite visualizar la gráfica del modelo que se ha creado, en él se muestran las operaciones y capas que se siguen. Algo más que se puede realizar dentro de TensorBoard es ver histogramas de pesos, sesgos y otros tensores a medida que cambian a lo largo de las épocas de entrenamiento. Además de proyectar incorporaciones en un espacio de dimensiones más bajas. Algo muy útil es que esta herramienta nos brinda la facilidad de mostrar imágenes, texto y datos de audio que estemos usando para el entrenamiento de un modelo.

Podemos consultar las distribuciones e histogramas de los tensores, tanto en pesos como en bias para cada época. **Embedding Projector** de TensorBoard Puede utilizar el proyector de TensorBoard para visualizar cualquier representación vectorial. Por ejemplo, Word Embeddings o imágenes. [25] [26]

Keras: la API de TensorFlow

Keras es la API (Application Programming Interface) de alto nivel de TensorFlow para construir y entrenar modelos de DL. Se utiiza para diversos campos, con características clave [6]:

- **Amigable con el usuario:**
  Tiene una interfaz simple y optimizada para casos de uso común.
- **Modular y configurable:**
  Los modelos dentro de Keras se fabrican conectando bloques de construcción configurables entre sí.

PyTorch vs TensorFlow

PyTorch y TensorFlow presentan algunas ventajas y desventajas al momento de querer usarlas para ciertos propósitos. Algunas **ventajas de PyTorch** son que este, al estar basado en Python, es directamente integrable con sus código y bibliotecas, además tiene una sintaxis similar a dicho lenguaje, por lo que es simple de aprender. Algo importante son las gráficas de cálculos dinámicas, lo que permite cambiar el comportamiento de una Red Neuronal en tiempo de ejecución, pues mejora la optimización y el performance. Aún con estas ventajas, en entornos de producción no es tan utilizado como lo es en la academia, pues cuenta con interfaces de visualización y supervisión limitadas, y no es una herramienta de desarrollo extremo a extremo.

Por otro lado, TensorFlow cuenta con el respaldo de Google para estarse actualizando frecuentemente con nuevas funciones. Además, se proporciona la herramienta TensorBord para visualizar datos y la

depuración sencilla de los nodos. Otra cosa es importante es que tiene compatibilidad con Keras, lo que permite codificar funcionalidades de alto nivel y usar funciones específicas del sistema, como pipelines y estimadores. Una ventaja grande de TensorFlow es su compatibilidad con los lenguajes C++, JavaScript, Python, C#, Ruby y Swift. Sin embargo, TensoFlow no es tan eficiente como PyTorch, y solo tiene soporte de NVIDIA para GPU y Python para su programación.

Red neuronal básica

Para tener un ejemplo concreto que puede ayudar al entendimiento del funcionamiento de una red neuronal, veamos la siguiente red de una capa utilizando keras.
En python primero se importan las librerías necesarias, en este caso se importan tres bibliotecas de **keras**: *models, layers y optimizers*, la primera se utiliza para crear modelos, la segunda para añadir capas de densidad a la red y la última se importa para optimizar a la red. Además usaremos **numpy** y el módulo **random**. Se dividirá la construcción de esta red por fases.

Fase 1: definir la red

Primero se define la función **neuralnet** (claro que el nombre es arbitrario), cuyas variables serán:

- **X**: el conjunto de deatos con el que se entrena a la red
- **Y**: el conjunto de datos con el que se valida la red,
- **nodes**: el número de nodos que tendrá la capa
- **split_porcent**: el porcentaje de datos usados para entrenamiento y para validación
- **learning_rate**: la taza de aprendizaje para el ajuste de pesos y sesgos
- **epochs**: el número de épocas (iteraciones) con el que se entrenará a la red
- **activation**: la función de activación para los nodos, esta debe ser diferenciable y existen varias ya predeterminadas, como relu, leaky relu, sigmoid, etc.
- **dactivation**: la derivada de la función de activación.

```
def neuralnet(X,Y,nodes,split_porcent,learning_rate,epochs,activation,dactivation
```

Fase 2: definir parámetros

Una vez definido esto, definimos la forma del input y del output:

```
n_input = X.shape[1]
n_output = Y.shape[1]
```

después separamos los datos que se usarán para entrenamiento y los que va a predecir cuando esté entrenada.

```
X_train, Y_train, X_val, Y_val = split(X,Y,split_porcent)
```

Fase 3: introducción de pesos y sesgos

Se definen los pesos y los sesgos para los nodos, estos se definen de manera aleatoria y a lo largo del entrenamiento se irán ajustando

```
w1 = np.random.rand(n_input,nodes)
w2 = np.random.rand(nodes,n_output)
b1 = np.random.rand(nodes)
b2 = np.random.rand(n_output)
```

notemos que w1 y w2 son matrices, de dimensión (número de entradas x número de nodos) y (número de nodos x número de salidas). w1 conecta a la capa de entrada con la capa oculta, y w2 conecta a la capa oculta con la capa de salidas. b1 y b2 son vectores de dimensión dada por el número de nodos y el número de saldas respectivamente.

Fase 4: épocas

Ahora se crea una lista para registrar los errores y se hacen las iteraciones (o épocas)

```
errors=[]
validation=[]
for(i in range(epochs+1)):
    z1 = np.dot(X_train,w1)+b1
    a1 = activation(z1)
    z2 = np.dot(a1,w2)+b2
    a2 = z2
```

lo que sucede aquí es que se hace un producto punto entre los datos de entrenamiento y los pesos, además se le suma el sesgo (que en inicio es aleatorio). Al mismo tiempo se introduce z1 a la función de activación, obteniendo a1. Después, z2 es el resultado de hacer el producto punto entre a1 y x2, y sumarle el sesgo b2. A a2 se le asigna el valor de z2, pues se supone el comportamiento de un problema de tipo lineal.

Fase 5: backpropagation

Ahora se comparan los resultados de la red con los resultados reales esperados

```
delta2 = a2 - y_train
delta1 = (delta2).dot(w2.T)*dactivation(z1) ////////////////////////////

errors.append(error(delta2))
```

Como se puede ver, **delta2** es la diferencia entre los valores arrojados por la red y los datos reales de validación, y **delta1** es el error en la capa oculta, dado por el producto punto entre delta2 y la transpuesta de la matriz de pesos entre la capa oculta y la salida multiplicado por la derivada de la función de activación aplicada a **z1**. Además se guarda el error en cada una de las épocas.

Fase 6: actualización de pesos y sesgos

En esta parte se realiza el ajuste de los pesos asignados a cada uno de los nodos de la capa oculta, esto para reducir los errores encontrados

```
    w2 -= learning_rate * a1.T.dot(delta2)
    b2 -= learning_rate * (delta2).sum(axis=0)

    w1 -= learning_rate * X_train.T.dot(delta1)
    b1 -= learning_rate * (delta1).sum(axis=0)
    print("epoch %d, cos %f" %(int(i),error(delta2)))
```

Fase 7: validación

Una vez terminadas las épocas se realiza la validación para cada uno de sus resultados

```
    z1 = np.dot(X_val, w1) + b1
    a1 = activation(z1)
    z2 = np.dot(a1,w1) + b2
    a2 = z2
    delta = a2 - y_val
    validation.append(error(delta))
```

Fase 8: final

Por último se grafican los errores y la validación encontrados en cada época, además se imprimen los pesos y los sesgos

```
    plt.plot(range(i+1), errors, color = 'darkcyan', label = 'Training')
    plt.plot(range(i+1), validation, color = 'darkred', label = 'Validation')
    return w1, b1, w2, b2
```

La función **error** está dada por

```
def error(C):
    m = len(C)
    e = (1/(2*m))*np.linalg.norm(C)**2 [12]
```

Ejemplo de una red neuronal. MNIST

A continuación se muestra un ejemplo de una red neuronal con un propósito en concreto. Revisaremos el ejemplo de MNIST, un dataset de Jupyter-Notebook. El dataset MNIST es el considerado "Hello World"de la visión artificial. Contiene un conjunto de entrenamiento de 60.000 imágenes de dígitos manuscritos (de 0 a 9), y otro conjunto de pruebas con 10.000 muestras adicionales. Las imágenes originales fueron normalizadas de forma que cupiesen en un cuadrado de 20x20 píxels manteniendo las proporciones de la imagen original, y el resultado se centró en un grid de 28x28 pixeles. Es de estas modificaciones de donde proviene la "M"de "MNIST"(**Modified National Institute of Standards and Technology, Instituto de Estándares y Tecnología de los Estados Unidos**). [18] La finalidad de esta red es, dado un dígito escrito a mano, intentar predecir de qué dígito se trata. Se dividirá la red en etapas para un mejor entendimiento.

Fase 1: importación de librerías y datos

Primero se importan las librerías necesarias, se usarán numpy, matplotlib.plyplot y tensorflow

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
```

Después se importa el dataset MNIST con el nombre **mnist** y se separan los datos en un conjunto de pruebas y uno de testeo, que a su vez se dividen en dos conjuntos: **x** que son datos conocidos y **y** que son los datos target (los que se intentarán predecir)

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Fase 2: separación y normalización de datos

Los valores de cada elemento de las matrices del dataset tienen valores que van de 0 a 255. Es más conveniente normalizar estos valores, para ello simplemente dividimos entre 255 los valores de los datos

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
```

La forma de estos conjuntos nos da una mejor idea de lo que tenemos en cada uno de ellos. Al momento de ejecutar

```
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

obtenemos que el conjunto **x** de entrenamiento tiene la forma (60000, 28, 28), es decir, es un tensor de 60,000 entradas (estas entradas nos dice de qué dígito se trata cada muestra) de dimensión 28 x 28 (el número de pixeles por imagen). Además el target de entrenamiento tiene 60,000 entradas (60000,). El conjunto **x** de testeo tiene la forma (10000, 28, 28), es decir, 10,000 datos de entradas de dimensiones 28 x 28. Finalmente, **y** de testeo tiene la forma (10000,), que son 10,000 entradas cuyos valores se van a predecir.

Fase 3: visualización de datos

Podemos visualizar los datos que tenemos para darnos una mayor idea de qué es con lo que estamos trabajando, para ellos utilazaremos la función **imshow**, que asigna a cada valor de nuestra matriz un color, en este caso utilizaremos **cmap = 'binary'**, que nos dará valores entre blanco y negro.

```
plt.imshow(x_train[0], cmap='binary')
```

obteniendo la siguiente imagen

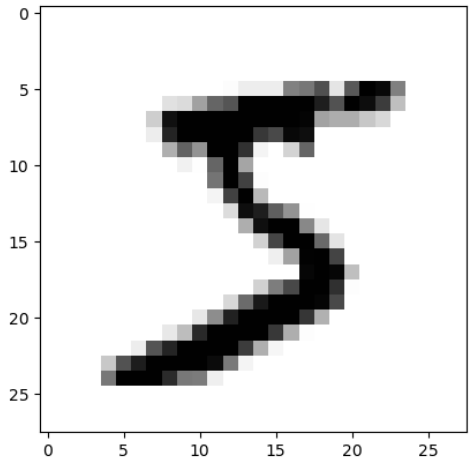

Figura 3: Primera imagen del conjunto **x** de entrenamiento

Fase 4: creación de la red

A continuación se crea la CANN que se utilizará para este ejemplo.

Primero se crea el modelo que llamamos **mod**, se usa la función de keras **Sequential**, que nos permite agregar capas de forma secuencial. Primero tenemos una capa de entrada **Conv2D**, la cual crea un núcleo de convolución que interactúa con la capa de entrada para producir un tensor de salida. Este funciona usando 32 nodos de tamaño 3x3, y se indica que el input es de tamaño 28 x 28 con un solo canal de color, ya que las imágenes con las que trabajamos están en escala de grises.

En esta capa no se agrega una función de acrivación, ya que lo que se busca es procesar los datos sin ningún tipo de procesamiento.

Después se agrega un filtro **MaxPooling** de dimensión 2x2, este lo que hace es ir barriendo la imagen en cuadrados de 2x2 pixeles y tomar el píxel de mayor valor, además a esta capa se le agrega una función **padding = 'same'**, lo que esto hace es básicamente completar con ceros valores que no alcance a cubrir el filtro de 2x2 que utilizamos, pues imaginemos que tenemos una imagen de 28 x 27 pixeles, el filtro que elegimos no podrría trabajar bien en el borde de dicha imagen, por lo que es útil la función padding. Esto último en principio no es necesario, pues las imágenes que hemos tomado son ideales para lo que estamos trabajando, sin embargo, puede ser que las imágenes que se presenten en la fase de test de la red, no tengan el tamaño ideal.

```
mod = tf.keras.Sequential([
```

```
        tf.keras.layers.Conv2D(32, (3,3),
        input_shape=(28,28,1),
        activation='relu'),

        tf.keras.layers.MaxPooling2D(pool_size=(2,2), padding='same')
```

Después de esto, se crea una capa intermedia de convolución, la cual ahora trabaja con 64 filtros de convolución. Además se agrega una función de activación ReLu, es conveniente dicha función, ya que se puede tener un buen ajuste por los valores positivos del input. Esta función de activación está definida como

$$ReLu(x) = max(0, x).$$

Después se agrega nuevamente la función MaxPooling2D, y se agrega un padding='same'.

```
        tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
        tf.keras.layers.MaxPooling2D(pool_size=(2,2), padding='same'),
```

Finalmente, usamos la función **tf.keras.layers.DropOut(0.5)** que hace que se desactiven aleatoriamente la mitad de las neuronas en cada época de entrenamiento. El dropout es generalmente utilizado como un método de regularización de los modelos. El objetivo principal de esta técnica es esencialmente mitigar la posible aparición del fenómeno conocido como overfitting, es decir, que los datos reproducidos por la red solo se ajusten a los datos de entrenamiento y no a los datos que se le pudieran presentar más adelante para la prueba de nuestro modelo.

Después agregamos una capa **Flatten** que asigna a la imagen un vector simple, esto se hace ya que queremos transicionar de capas convolucionales a una capa **Dense** con 100 neuronas y activación relu. Como capa de salida agregamos una capa de tipo **Dense** con 10 nodos y activación **softmax**, esto debido a que nos encontramos con un problema de clasificación con 10 posibles resultados, además, la función softmax nos da probabilidades de que cada entrada se encuentre en una de las posibles categorías, sumando entre ellas 100 %.

```
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(100, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
])
```

Fase 5: entrenamiento y optimización del modelo

En esta fase se compila el modelo creado, utilizando el optimizador **adam**, que es un método que usa el descenso de gradiente, es decir, sigue la dirección en la que la función de pérdida (que en nuestro caso es **sparse_categorical_crossentropy**) decrece. La función de pérdida utilizada en este caso es conveniente debido a que existen más de dos categorías a las que cada una de las muestras del dataset (**x_train**) puede pertenecer (recordemos que van del 0 al 9). Además se usa cómo métrica de rendimiento la exactitud del modelo, es decir, la proporción de predicciones correctas sobre el total de predicciones hechas.

```
mod.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
```

Ahora usamos el método **Model.fit** para ajustar los parámetros del modelo y minimizar la pérdida. Se utiliza **x_train** como los parámetros para los cuales se tendrá que predecir su valor correspondiente en **y_train**. Estos datos se ajustarán durante 20 épocas, pues pasado este número de iteraciones no se ve un aumento notable en la precisión y la función de pérdida aumenta un poco, por lo que se decidió como un número óptimo.

```
model.fit(x_train, y_train, epochs=20)
```

Con esto se obtuvo el siguiente rendimiento en cada época de entrenamiento

```
Epoch 15/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9981 - loss: 0.0065
Epoch 16/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9981 - loss: 0.0068
Epoch 17/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 28s 15ms/step - accuracy: 0.9985 - loss: 0.0051
Epoch 18/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 28s 15ms/step - accuracy: 0.9982 - loss: 0.0059
Epoch 19/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 15ms/step - accuracy: 0.9982 - loss: 0.0057
Epoch 20/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 28s 15ms/step - accuracy: 0.9982 - loss: 0.0068
Epoch 21/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 29s 15ms/step - accuracy: 0.9985 - loss: 0.0059
Epoch 22/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 30s 16ms/step - accuracy: 0.9985 - loss: 0.0061
Epoch 23/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 29s 15ms/step - accuracy: 0.9980 - loss: 0.0066
Epoch 24/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 15ms/step - accuracy: 0.9987 - loss: 0.0057
Epoch 25/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 14ms/step - accuracy: 0.9985 - loss: 0.0048
Epoch 26/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 14ms/step - accuracy: 0.9985 - loss: 0.0056
Epoch 27/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 15ms/step - accuracy: 0.9978 - loss: 0.0086
Epoch 28/28
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 14ms/step - accuracy: 0.9985 - loss: 0.0053
```

Figura 4: Rendimiento de la red en cada época de entrenamiento

Fase 6: validación del modelo con los datasets de testeo

Una vez entrenado el modelo, se usa **model.evaluate** para evaluar la red que se ha creado. Notemos que ahora se utilizan los dataset de testeo.

```
model.evaluate(x_test,  y_test,  verbose=2)
```

Obtuvimos así un 99.29 % de exactitud con nuestro modelo y una pérdida de 5.6 % [19] [20] [21]

Visualización de resultados

Una vez hecho todo el proceso, podemos visualizar qué tan efectivo ha sido el modelo creado. Ajustamos el modelo a todos los datos de prueba. Recordando que la última capa de nuestra red nos daba una probabilidad de que cada muestra perteneciera a una clase, utilizamos pred_test = mod(x_test[:]).numpy() para poder visualizar las probabilidades en un vector de 10 entradas (0 al 10), la clase a la que nuestra red decida que pertenece cada una de las muestras tendrá en valor más alto.

```
pred_test = mod(x_test[:]).numpy()
```

Ahora, utilizando la función **argmax** de numpy para poder transformar estos vectores en etiquetas, que nos dirán de qué número se trata cada una de las muestras. Notemos que argmax justamente toma el valor más alto de cada vector y le asigna un valor numérico correspondiente a su número de entrada.

```
pred_clases = np.argmax(pred_test, axis=1)
```

Ahora ya tenemos dos arrays para poder hacer la comparación. En este caso se realizará una matriz de confusión, pues nos dirá qué valores se confundían con otros durante la predicción. Para hacer esto importamos de la librería **scikitlearn**, las métricas necesarias para hacer esta matriz

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

y ahora se crea la matriz,

```
mat_conf = confusion_matrix(y_test, y_predc)
```

esto lo podemos visualizar ahora con **ConfusionMatrixDisplay**, donde se indica que la matriz que se visualizará será **mat_conf** y se dan etiquetas de 0 al 10

```
mat = ConfusionMatrixDisplay(confusion_matrix=mat_conf, display_labels=np.ara
disp.plot(cmap=plt.cm.Oranges)
```

obteniendo así la siguiente matriz de confusión

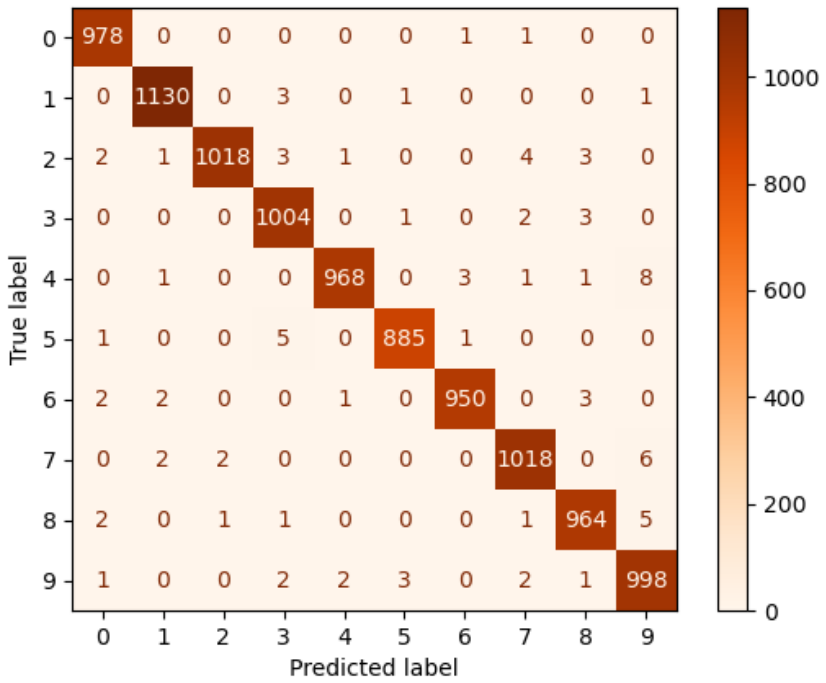

Figura 5: Matriz de confusión de los valores predichos por la red y los valores verdaderos

Resulta útil las matrices de confusión, ya que nos permite visualizar el desempeño de un modelo de aprendizaje. Cada columna de la matriz representa el número de predicciones de cada clase, mientras

que cada fila representa a las instancias en la clase real. Para términos prácticos, nos permite ver qué tipos de aciertos y errores está teniendo nuestro modelo a la hora de pasar por el proceso de aprendizaje con los datos. Por ejemplo, podemos ver que en el eje X se encuentran las clases que predijo el modelo, mientras que en el eje Y vemos las clases reales de cada muestra de nuestro dataset. Fijémonos en el caso particular del 4 del eje Y, nuestro modelo predijo erróneamente que este era un **9**, sin embargo, este es una predicción falsa. En la diagonal del diagrama vemos los valores predichos correctamente, es decir, vemos las veces en qué el modelo acertó en la predicción.

Además de esta forma de visualizar el desempeño de nuestro modelo, es ilustrativo contar con algunas otras métricas que nos permiten ver el performance de la red a lo largo de las épocas de entrenamiento. Resulta natural usar una herramienta poderosa que se mencionó en este mismo texto, TensorBoard. Veamos a continuación algunas de las métricas del modelo que hemos creado.

Implementación de TensorBoard

Para poder visualizar métricas de nuestro modelo con TensorBoard, necesitamos primero importar ciertas librerías, con **datetime** podremos ver la hota y fecha en la que se corre un modelo, además **os** nos permitirá crear directorios y manipular archivos. Una vez hecho esto es necesario crear una ruta de directorio dada por la fecha y hora en tiempo real. Primero combinan los componentes de la ruta *logs*, *fit* y la fecha y hora actual. Además, con *os.system()* se ejecuta el comando del sistema proporcionado. Después se crea un callback de TensorBoard, en el que *log_dir=log_dir* especifica el directorio donde se guardarán los logs. *histogram_freq=1* indica cada cuántas épocas se registrarán los histogramas.

Al final de hacer todo esto, la parte del código que contenía al modelo se ve de la siguiente forma:

```
log_dir = os.path.join("logs", "fit",
datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))

os.makedirs(log_dir, exist_ok=True)

tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

mod = tf.keras.Sequential([
tf.keras.layers.Conv2D(32, (2,2), input_shape=(28,28,1)),
tf.keras.layers.MaxPooling2D(pool_size=(2,2), padding='same'),

tf.keras.layers.Dropout(0.5),
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(pool_size=(2,2), padding='same'),

tf.keras.layers.Flatten(),
tf.keras.layers.Dense(100, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
])
```

además es necesario que se especifique el Callback al momento de entrenar el modelo:

```
mod.fit(x_train, y_train, epochs=35, callbacks=[tensorboard_callback])
```

Y finalmente inicializamos TensorBoard

```
from tensorboard import notebook
```

```
notebook.display(port=6006, height=1000)
```

Ahora ya podemos tener acceso a esta herramienta de visualización. Lo primero que veremos es la exactitud de nuestro modelo a lo largo de las épocas. Esto es una de las tantas métricas que nos brinda TensorBoard.

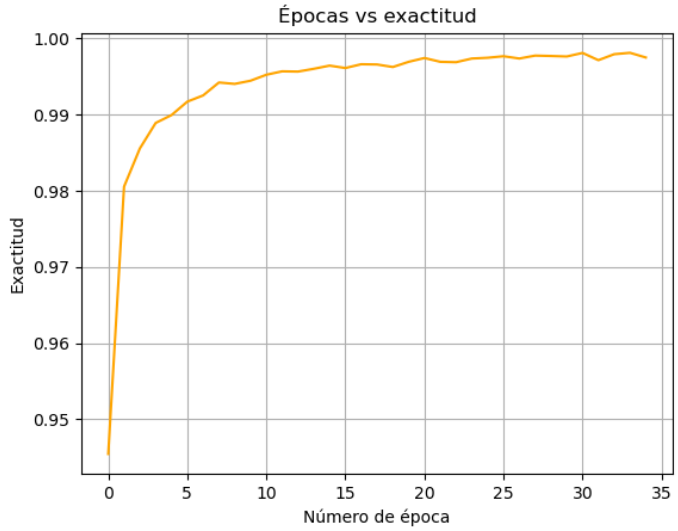

Figura 6: Exactitud vs épocas de entrenamiento del modelo

De la figura 6 podemos notar que las primeras 15 épocas son las de mayor crecimiento, mientras que a partir de la época número 30 la exactitud del modelo no varía tanto para bien, sino que esta fluctúa, por lo que resultaría buena idea entrenar al modelo hasta dicha época.

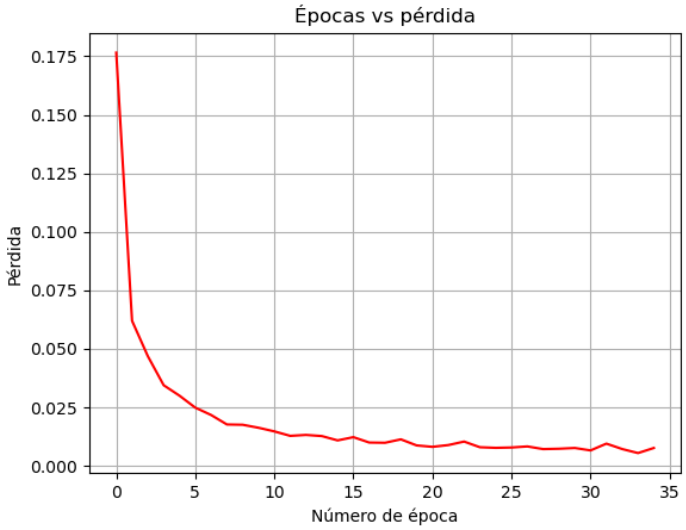

Figura 7: Pérdida vs épocas de entrenamiento del modelo

En la figura 7 podemos ver cuál es la pérdida que tiene el modelo a lo largo de las épocas. Vemos un comportamiento parecido al que tiene la exactitud en la figura anterior. Vemos que la pérdida de

disminuye significativamente hasta la época 20, de esta en adelante vemos que la pérdida se mantiene constante con pequeñas fluctuaciones.

Apéndice A: ScikitLearn

Scikit-Learn es una de estas librerías más populares en Python. Cuenta con algoritmos de **clasificación**, **regresión**, **clustering** y **reducción de dimensionalidad**. Además, presenta la compatibilidad con otras librerías de Python como NumPy, SciPy y matplotlib. La gran versatilidad de Scikit-learn la convierten en la herramienta básica para empezar a programar y estructurar los sistemas de análisis datos y modelado estadístico, pues cuenta con una gran variedad de algoritmos y herramientas, las cuales se combinan con otras estructuras de datos como Pandas o PyBrain.[13]

Clasificación

Los métodos de clasificación se entrenan para, como su nombre lo indica, clasificar datos en categorías bien definidas, las cuales están basadas en etiquetas de datos previos. ScikitLearn brinda modelos para tareas de clasificación [15]. Podemos tomar un ejemplo sencillo donde se busca clasificar flores basadas en varias de sus características, como la longitud y el ancho de sus pétalos y sépalos. Estas flores pueden pertenecer a tres categorías: iris-setosa, iris-versicolor e iris-virginica.
Antes de realizar el modelo de clasificación, es útil realizar un análisis de los datos para saber cuáles nos darán más o menos información para lograr la tarea. Podemos utilizar Seaborn para realizar este análisis, y obtener la siguiente matriz de correlación

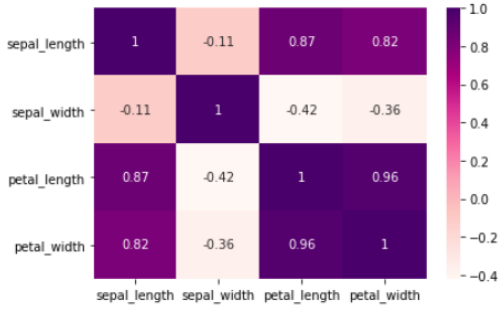

Figura 8: Ejemplo ScikitLearn

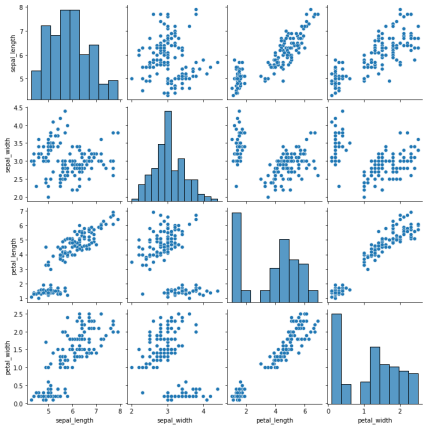

Figura 9: Correlaciones entre características de las flores. Ejemplo de ScikitLearn

Vemos que en efecto hay correlaciones bastante marcadas entre las propiedades de las flores, por lo que podemos usarlas para entrenar a un modelo de clasificación. En este caso particular se usará

el método de K-Neighbrs. Una vez teniendo nuestro conjunto de datos, los separamos entre datos de entrenamiento y datos de prueba.

```
data_test , data_train , target_test , target_train = train_test_split(data ,
target ,
test_size=0.75,
random_state=1)
```

Importamos el método KNeighborsClassifier de sklearn.neighbors, y se entrena nuestro modelo

```
clf.fit(data_train , target_train)
```

notemos que además de separar los dato entre los que son para entrenamiento y para prueba, a su vez los separamos entre datos que nos dan información del target u objetivo.

Una vez hecho lo anterior, se usa el modelo creado:

```
pred = clf.predict(data_test)
```

Particularmente se obtuvieron los siguientes resultados, obteniendo también una comparación entre los datos reales y las predicciones hechas por el modelo [14]:

```
pred == target_test

21      True
70      True
3       True
142     True
30      True
147     True
106    False
47      True
115     True
13      True
88      True
8       True
81      True
60      True
0       True
1       True
57      True
22      True
61      True
63      True
7       True
86      True
96      True
68      True
50      True
101     True
20      True
25      True
134     True
71      True
129     True
79      True
133     True
137     True
72      True
140     True
37      True
Name: species, dtype: bool
```

Figura 10: Predicciones del modelo de los K vecinos más cercanos

Regresión

El método LinearRegression ajusta un modelo lineal con coeficientes w = (w1,. . . , wp) para minimizar la suma residual de cuadrados entre los objetivos observados en el conjunto de datos y los objetivos

predichos por la aproximación lineal. Consta de varios parámetros:

- **fit_intercept** `bool`, *default=True*
  Indica si se debe calcular el intercepto para este modelo. Si se establece en False, no se usará intercepto en los cálculos (es decir, se espera que los datos estén centrados).

- **copy_X** `bool`, *default=True*
  Si es True, X se copiará; de lo contrario, puede ser sobrescrito.

- **n_jobs** `int`, *default=None*
  El número de trabajos a usar para la computación. Esto solo proporcionará una mejora en la velocidad en caso de problemas suficientemente grandes, es decir, si primero n_targets ¿1 y segundo si X es disperso o si positive está configurado en True. None significa 1 a menos que esté en un contexto de joblib.parallel_backend. -1 significa usar todos los procesadores.

- **positive** `bool`, *default=False*
  Cuando se establece en True, fuerza a que los coeficientes sean positivos. Esta opción solo es compatible con matrices densas. [17]

Ejemplo de uso de ScikitLearn para regresiones lineales.

A continuación podemos ver cómo es que se usa ScikitLearn para hacer regresiones lineales y con ellas, predicciones. **Housing_california.csv** es un dataset que contiene información de casas en el estado de California, este es el dataset que utilizaremos para hacer un modelo de regresión lineal con ScikitLearn. Además del precio de cada casa, el dataset contiene características como el número de habitaciones, el número de baños, la cercanía a la playa, etc. Una vez que se limpió el dataset, podemos usar ScikitLearn para hacer una regresión lineal múltiple, que ajusta un hiperplano de dimensión $N+1$ con N el número de características que tenemos en el datase. Después de un análisis se eligen las características que más ayudarán a predecir el precio de las casas, por lo que tenemos en particular un plano de dimensión 6, pues contamos con cinco características: ingreso medio, número de habitaciones, edad media de la casa, número de baños y cercanía al océano.

Una vez que tenemos el dataset limpio, procederemos a crear el modelo de regresión. Para esto, primero crearemos un pipeline que preprocesará los los datos categóricos y numéricos para una mejor ejecución a la hora de crear el modelo propiamente. A continuación se muestra cómo es que se crea el pipeline que preprocesa la información, primero se crean dos listas: una con las características categóricas y otra con las características numéricas. Después de esto se crea un transformador que preprocesa la información de cada una de las columnas.

```
numerical_list=['median_income', 'total_rooms', 'housing_median_age', 'total_b

categorical_list=['ocean_proximity']

num_transformer = Pipeline(steps=[('scaler', StandardScaler())])

cat_transformer = Pipeline(steps=[('onehot', OneHotEncoder())])

transformer = ColumnTransformer(steps = [('num', num_transformer, numerical_l
```

Ahora ya se puede crear el modelo de regresión lineal. A continuación se muestra el modelo que se ha creado,

```
model = Pipeline([('transformer', transformer),
                  ('linear_regrression', LinearRegression() )])
```

donde **Pipeline** se ha importado de **sklearn.pipeline**, y toma como argumento **steps**, el cual indica una secuencia de pasos y un estimador final que se aplica de manera secuencial a los datos. Finalmente se debe entrenar el modelo con los datos que se tienen para después usarlo para predecir el parámetro de los precios de las casas a partir de todas las demás características. 1

Bibliografía

1 Géron, A. (2019, September 5). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media.

2 Deep Learning With TensorFlow: A Review. Bo Pang, Erik Nijkamp, and Ying Nian Wu, 2019

3 TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, 2015.

4 ¿Qué es PyTorch? — IBM. (n.d.). https://www.ibm.com/mx-es/topics/pytorch

5 Guia inicial de TensorFlow 2.0 para principiantes. (n.d.). TensorFlow. https://www.tensorflow.org/tutorials/quickstart/beginner?hl=es-419

6 Clasificacion Basica: Predecir una imagen de moda. (n.d.). TensorFlow. https://www.tensorflow.org/tutorials/keras/classification?hl=es-419

7 ¿Qué son las redes neuronales? — IBM. (n.d.). https://www.ibm.com/mx-es/topics/neural-networks

8 Generative models. OpenAi, https://openai.com/index/generative-models/

9 Tutorial on Variational Autoencoders. CARL DOERSCH. Carnegie Mellon / UC Berkeley August 16, 2016, with very minor revisions on January 3, 2021

10 Lawton, G. (2023, March 8). GANs vs. VAEs: What is the best generative AI approach? Enterprise AI. https://www.techtarget.com/searchenterpriseai/feature/GANs-vs-VAEs-What-is-the-best-generative-AI-approach

11 Vídeo útil para comprender mejor las VAEs: https://youtu.be/dlnA1uiWu90?si=-wTrHsvwynRnMRv6

12 Ejemplo tomado de https://github.com/JuanDDiosRojas/Arts/blob/main/Deep%20Learning%20and%20its%20applications%20to%20cosmology/Observational%20cosmology%20with%20Artificia20Neural%20Networks.ipynb

13 Scikit-Learn, herramienta básica para el Data Science en Python. (2023, August 30). Máster En Data Science. https://www.master-data-scientist.com/scikit-learn-data-science/#:~:text=%C2%BFQu%C3%A9%20es%20Scikit%2DLearn%3F,como%20NumPy%2C%20SciPy%20y%20matplotlib.

14 Ejemplo tomado de https://github.com/israelcampos011/ScikitLearn/blob/main/p%C3%A9talos.ipynb

15 C. (2023, November 17). Classification using Scikit-Learn for example ANN,SVM,DT, — Medium. Medium. https://medium.com/@Coursesteach/guide-to-supervised-learning-with-scikit-learn-part-3-c31b01c547f9

16 Más ejemplos del uso de ScikitLearn: https://github.com/israelcampos011/ScikitLearn

17 LinearRegression. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

18 El dataset MNIST — Interactive Chaos. (n.d.). https://interactivechaos.com/es/manual/tutorial-de-deep-learning/el-dataset-mnist

19 https://github.com/israelcampos011/TensorFlow

20 https://github.com/tensorflow/docs-l10n/blob/master/site/es-419/tutorials/quickstart/beginner.ipynb

21 Team, K. (n.d.). Keras: Deep Learning for humans. https://keras.io/

22 ¿Qué es una red neuronal? - Explicación de las redes neuronales artificiales - AWS. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/neural-network/#:~:text=A%20neural%20network%20is%20a,that%20resembles%20the%20human%20brain.

23 Handbook of Research on Information Security in Biomedical Signal Processing. (2018). In Advances in information security, privacy, and ethics book series. https://doi.org/10.4018/978-1-5225-5152-

24 Ravanbakhsh, S., Oliva, J., Fromenteau, S., et al. 2017, arXiv:https://arxiv.org/pdf/1711.02033

25 TensorBoard — TensorFlow. (n.d.). TensorFlow. https://www.tensorflow.org/tensorboard?hl=es-419

26 Henríquez, E. B. (2023, June 29). Supervisa tu entrenamiento de redes neuronales con TensorBoard. Telefónica Tech. https://telefonicatech.com/blog/supervisa-tu-entrenamiento-de-redes-neuron

[27] Z. Ni, Y, Cai. Department of Thoracic Surgery. (2019) Skin Cancer Diagnosis Based on Optimized Convolutional Neural Network.