



Esta obra está bajo una
[Licencia Creative Commons Atribución-NoComercial-SinDerivar 4.0 Internacional](#)



Organizadores



Asociación Española de Contabilidad
y Administración de Empresas

Colaborador



Módulo 8. Minería de Datos II

Curso Modular Big Data y Data Science Aplicados a la Economía y a la Administración y Dirección de Empresas

Pablo Sánchez Cabrera

Ángel Rodríguez Chicote

Alfonso Carabantes Álamo

2024-01-01

Índice

Introducción	5
1 Deep Learning	7
1.1 Introducción	7
1.2 Principales arquitecturas y software de Deep Learning	8
1.2.1 Principales arquitecturas	8
1.2.2 Software	11
1.3 Conceptos básicos de las Redes Neuronales	17
1.3.1 Datos	19
1.3.2 Arquitectura de red	20
1.3.3 Función de coste y pérdida	20
1.3.4 Optimizador	22
1.3.5 Función de activación	25
1.3.6 Regularización	26
1.3.7 Dropout	26
1.3.8 Dropconnect	27
1.3.9 Inicialización de pesos	27
1.3.10 Batch normalization	27
1.4 Redes Neuronales Convolucionales	28
1.4.1 Introducción	28
1.4.2 Clasificación de imágenes	28
1.4.3 Clasificación de textos	37
1.5 Redes Recurrentes	42
1.6 Redes recurrentes: Elman, Jordan, LSTM y GRU	42
1.6.1 Forecasting en series de tiempo	42
1.6.2 Clasificación y generación de textos	42
1.7 Autoencoders	42
1.7.1 Bases del Autoencoder	42
1.7.2 Casos de uso	47
1.8 Arquitecturas preentrenadas	47
1.8.1 Paquetes específicos en Python	48
1.8.2 Tensorflow-Hub	49
1.8.3 Arquitecturas Zero-shot	49
1.8.4 Detección de objetos	50
1.8.5 Conversión de voz a texto	50
1.9 Aprendizaje por Refuerzo	50
1.9.1 Introducción	50
1.9.2 Formalismo Matemático	51
1.9.3 Taxonomía de Algoritmos	55
1.9.4 Q-Learning (value)	57
1.9.5 DQN (Deep Q-Learning)	58
1.9.6 Listado Algoritmos	61

2 Modelos Gráficos Probabilísticos y Análisis Causal	66
2.1 Redes Bayesianas	66
2.1.1 Modelo Naive Bayes: Hipótesis Map y Teorema de Bayes	67
2.1.2 Modelo Naive-Bayes	70
2.2 Modelos Bayesianos	75
2.2.1 Formulación general	75
2.2.2 Independencia condicional e inferencia de la red	76
2.2.3 Aprendizaje de las redes bayesianas	77
2.2.4 Clasificadores	79
2.3 Modelos Ocultos de Markov	83
2.3.1 Cadenas de Markov	83
2.3.2 Cadena de Markov absorvente	85
2.3.3 Modelos Ocultos de Markov	89
3 Algoritmos Genéticos	97
3.1 Introducción	97
3.2 Fundamentos teóricos	98
3.2.1 Codificación de los datos	99
3.2.2 Algoritmo	100
3.2.3 Otros Operadores	104
3.2.4 Parámetros necesarios al aplicar Algoritmos Genéticos	104
3.3 Casos de uso	105
3.3.1 Selección de Variable	105
3.3.2 Entrenamiento de Red Neuronal	107
3.3.3 Arquitectura de RedNeuronal	109
3.4 Algoritmos genéticos con R	110
3.4.1 Paquetes para usar en R	110
3.4.2 Selección de variables	110
3.4.3 Entrenamiento de Red Neuronal	110
3.5 Algoritmos genéticos en Python	110
3.5.1 Paquetes para usar en Python	110
3.5.2 Optimización de funciones	110
3.5.3 Entrenamiento de Red Neuronal	110
4 Lógica Difusa	113
4.1 Contexto y conceptos clave	113
4.1.1 Operaciones sobre conjuntos difusos	114
4.1.2 Funciones de membresía y modelamiento difuso	117
4.2 Ejemplos prácticos	120
4.3 Fuzzy C-Means	122
4.4 Fuzzy Matching	123
4.4.1 Ejemplo 1	124
4.4.2 Ejemplo 2	124
Appendices	126
A Anexo 1	126

Introducción

Anotación¹ Diagrama Conceptual Módulo 8: Minería de Datos II

Referecnia bibliografica [4]

Referencia gráfico Figure 1 Otra referencia a gráfico 1

Formulas con numeración y referencia

$$\frac{\partial C}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial C^2} + rS \frac{\partial C}{\partial S} = rC \quad (0.1)$$

Ver la fórmula 0.1 Ver la fórmula ?? o ver la fórmula Equation 0.1

i Nota:

Note that there are five types of callouts, including: note, warning, important, tip, and caution.

! Atención!

Atención!!!

! Recordad

Importante

? Truco

This is an example of a callout with a title.

! Expand To Learn About Collapse

This is an example of a ‘folded’ caution callout that can be expanded by the user. You can use `collapse="true"` to collapse it by default or `collapse="false"` to make a collapsible callout that is expanded by default.

¹Anotación de prueba

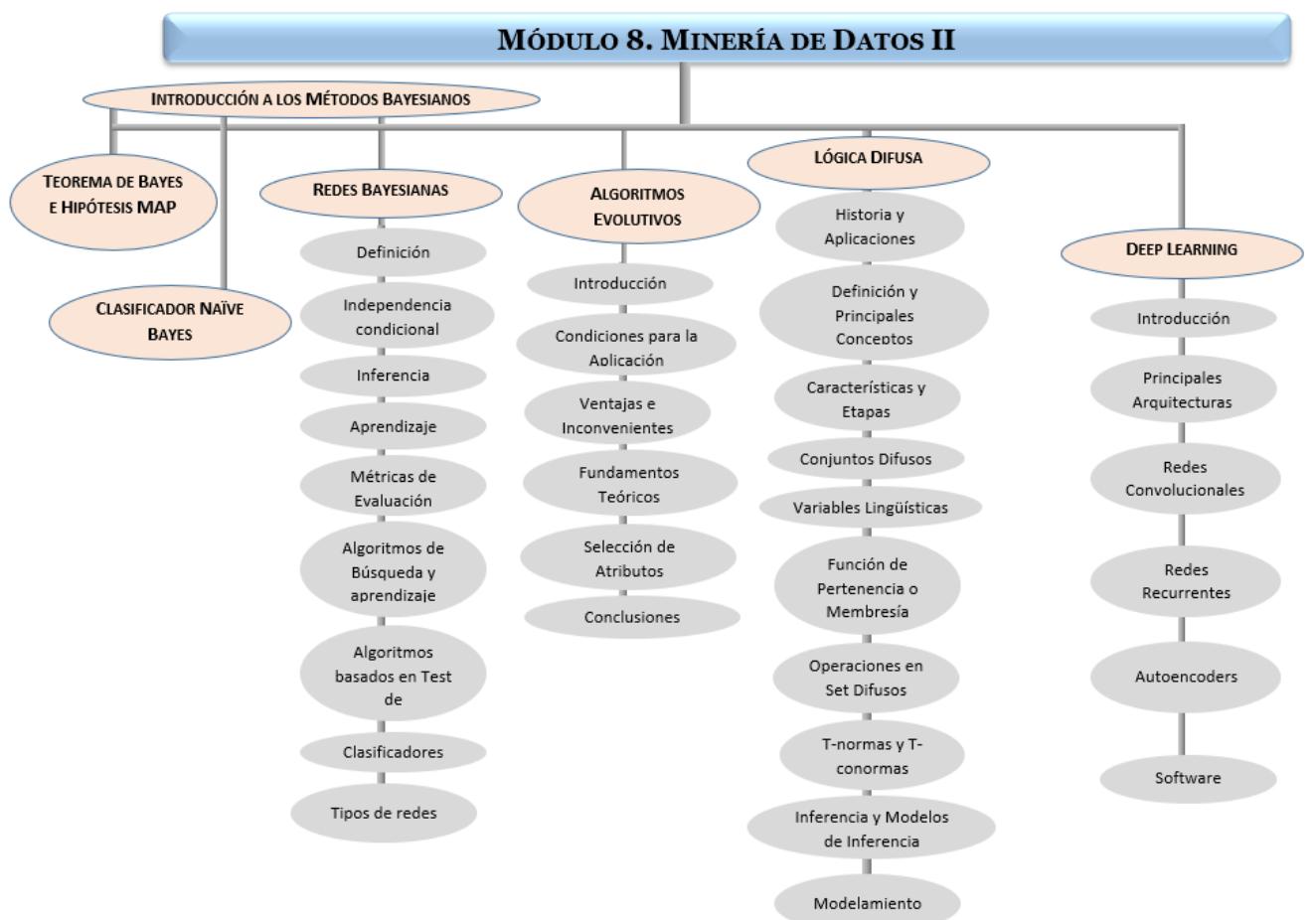


Imagen 1: Diagrama Conceptual

1 Deep Learning

1.1 Introducción

El **deep learning**, también conocido como aprendizaje **profundo**, es una disciplina que busca emular el funcionamiento del cerebro mediante el uso de hardware y software, generando inteligencia artificial. Este enfoque se materializa en redes neuronales artificiales (RNA), que emplean una abstracción jerárquica para representar datos en múltiples niveles. El proceso implica la utilización de arquitecturas de varias capas, donde cada una aprende patrones más complejos, favoreciendo el aprendizaje útil. Generalmente, se emplea aprendizaje no supervisado para guiar el entrenamiento de las capas intermedias. Aunque derivado del machine learning, el deep learning se distingue por su arquitectura en capas, incluyendo redes convolucionales y recurrentes, en contraste con métodos más simples como el Perceptrón Multicapa de una sola capa. Su avance se vio inicialmente obstaculizado por problemas de estancamiento en mínimos locales, resueltos mediante preentrenamiento no supervisado de las capas. Este enfoque ha impulsado un rápido crecimiento en el desarrollo de arquitecturas y algoritmos de RNA en los últimos años, manteniendo la esencia del aprendizaje jerárquico y profundo.

Las redes neuronales tienen una amplia gama de aplicaciones en diversos campos, desde la clasificación y regresión de datos hasta la identificación de imágenes, texto y audio.

En la *identificación de imágenes*, por ejemplo, pueden reconocer animales, señales de tráfico, frutas, caras humanas e incluso tumores malignos en radiografías. A medida que se combinan estas capacidades, se pueden abordar problemas más complejos como la detección de objetos y personas en imágenes o el etiquetado de escenas. Con el *análisis de videos*, las redes neuronales pueden contar personas, reconocer objetos y señales de tráfico, o detectar comportamientos como llevar un arma.

Cuando se trata de *datos de texto*, las redes neuronales se utilizan en sistemas de traducción, chatbots y conversión de texto a audio. En el caso de *datos de audio*, se emplean en sistemas de traducción, altavoces inteligentes y conversión de audio a texto.

Para **trabajar con redes neuronales**, es crucial representar los datos de entrada numéricamente, incluso convirtiendo variables categóricas en valores numéricos y normalizando los datos entre 0 y 1. Esto facilita la convergencia hacia soluciones óptimas. Es importante que los datos sean números en coma flotante, sobre todo si se van a trabajar con *GPUs*, *Graphics Process Units*, ya que permitirán hacer un mejor uso de los múltiples cores que les permiten operar en coma flotante de forma paralela. Actualmente, hay toda una serie de mejoras en las GPUs que permite aumentar el rendimiento de las redes neuronales como son el uso de operaciones en *FP16* (Floating Point de 16 bits en lugar de 32) de forma que pueden hacer dos operaciones de forma simultánea (el formato estándar es *FP32*) y además con la reducción de memoria (punto muy importante) al meter en los 32 bits 2 datos en lugar de sólo uno. También se han añadido técnicas de *Mixed Precision* (Narang et al. 2018), los *Tensor Cores* (para las gráficas de NVIDIA) son otra de las mejoras que se han ido incorporando a la GPUs y que permiten acelerar los procesos tanto de entrenamiento como de predicción con las redes neuronales.

1.2 Principales arquitecturas y software de Deep Learning

1.2.1 Principales arquitecturas

Actualmente existen muchos tipos de estructuras de redes neuronales artificiales dado que logran resultados extraordinarios en muchos campos del conocimiento. Los primeros éxitos en el aprendizaje profundo se lograron a través de las investigaciones y trabajos de Geoffre Hinton (2006) que introduce las Redes de Creencia Profunda en cada capa de la red de una Máquina de Boltzmann Restringida (RBM) para la asignación inicial de los pesos sinápticos. Hace tiempo que se está trabajando con arquitecturas como los Autoencoders, Hinton y Zemel (1994), las RBMs de Hinton y Sejnowski (1986) y las DBNs (Deep Belief Networks), Hinton et al. (2006) y otras como las redes recurrentes y convolucionales. Estas técnicas constituyen en sí mismas arquitecturas de redes neuronales, aunque también algunas de ellas, como se ha afirmado en la introducción, se están empleando para inicializar los pesos de arquitecturas profundas de redes neuronales supervisadas con conexiones hacia adelante.

Redes Convolucionales

Las redes neuronales convolucionales (CNNs) han transformado el panorama del Deep Learning, destacándose por su habilidad para extraer características de alto nivel a través de la operación de convolución. Diseñadas específicamente para el procesamiento de imágenes, las CNNs son altamente eficientes en tareas de clasificación y segmentación en el ámbito de la visión artificial.

Inspiradas en el funcionamiento de la corteza visual del cerebro humano, estas redes representan una evolución del perceptrón multicapa. Aunque su uso se popularizó en la década de 1990 con el desarrollo de sistemas de lectura de cheques por parte de AT&T, las CNNs han experimentado una evolución significativa desde entonces.

Su arquitectura se compone de capas de convolución, responsables de transformar los datos de entrada, y capas de pooling, encargadas de resumir la información relevante. Posteriormente, se aplican capas densamente conectadas para obtener el resultado final.

El auge de las CNNs se vio impulsado por iniciativas como la competencia ILSVRC, que propiciaron avances considerables en este campo. Entre los modelos más destacados se encuentran LeNet-5, AlexNet, VGG, GoogLeNet y ResNet, muchos de los cuales están disponibles como modelos preentrenados para su integración en diversas aplicaciones. Estos modelos, con estructuras de capas más complejas, representan el estado del arte en reconocimiento visual y están al alcance de cualquier investigador interesado en el Deep Learning.

Más allá de las arquitecturas conocidas, han surgido modelos más avanzados como DenseNet y EfficientNet, que optimizan el rendimiento y la eficiencia computacional. La transferencia de aprendizaje se ha convertido en una herramienta fundamental, permitiendo adaptar modelos preentrenados a tareas específicas con conjuntos de datos más pequeños, agilizando el entrenamiento y mejorando la generalización.

Las CNNs encuentran un amplio uso en tareas de segmentación semántica y detección de objetos, impulsadas por técnicas como U-Net y Mask R-CNN. Adicionalmente, métodos de aprendizaje débilmente supervisado y autoetiquetado están permitiendo entrenar modelos con datos etiquetados de manera menos precisa o incluso sin etiquetar.

Para mejorar la interpretabilidad de las CNNs, se han propuesto técnicas de visualización de atención visual, que permiten identificar las partes de una imagen que son más relevantes para la predicción del modelo.

Estos avances impulsan el continuo desarrollo de las CNNs, expandiendo su aplicación a diversos campos como el diagnóstico médico, la conducción autónoma y la robótica. La investigación activa en este campo

sigue explorando nuevas formas de mejorar la eficiencia, la precisión y la interpretabilidad de las CNNs para abordar desafíos cada vez más complejos en el procesamiento de imágenes y otros tipos de datos.

Autoencoders

Los Autoencoders (AE) son una clase de redes neuronales dentro del ámbito del Deep Learning, caracterizadas por su enfoque en el aprendizaje no supervisado. Aunque se mencionaron por primera vez en la década de 1980, ha sido en los últimos años donde han experimentado un notable interés y desarrollo. La arquitectura de un AE consiste en dos partes principales: el encoder y el decoder. El encoder se encarga de codificar o comprimir los datos de entrada, mientras que el decoder se encarga de regenerar los datos originales en la salida, lo que resulta en una estructura simétrica.

Durante el entrenamiento, el AE aprende a reconstruir los datos de entrada en la capa de salida de la red, generalmente implementando restricciones como la reducción de elementos en las capas ocultas del encoder. Esto evita simplemente copiar la entrada en la salida y obliga al modelo a aprender representaciones más significativas de los datos. Entre las aplicaciones principales de los AE se encuentran la reducción de dimensiones y compresión de datos, la búsqueda de imágenes, la detección de anomalías y la eliminación de ruido.

Además de los autoencoders estándar, existen varias variaciones que han surgido para abordar diferentes desafíos y aplicaciones específicas, como los Variational Autoencoders (VAE), los Sparse Autoencoders, los Denoising Autoencoders y los Contractive Autoencoders. Estas variaciones amplían el alcance y la versatilidad de los autoencoders en una variedad de contextos de aprendizaje automático, desde la compresión de datos hasta la generación de nuevas muestras y la detección de anomalías en conjuntos de datos complejos.

Redes Recurrentes

Las redes neuronales recurrentes (RNNs) revolucionaron el panorama del machine learning, posicionándose como una herramienta fundamental para procesar y analizar datos secuenciales. A diferencia de las redes neuronales tradicionales con una estructura de capas fija, las RNNs poseen una arquitectura flexible que les permite incorporar información del pasado, presente y futuro, lo que las convirtió en una gran apuesta ante tareas como el procesamiento del lenguaje natural, el reconocimiento de voz y la predicción de series temporales.

Gracias a su capacidad de memoria interna, las RNNs pueden capturar dependencias temporales en los datos secuenciales, una característica crucial para modelar el comportamiento de fenómenos que evolucionan con el tiempo. Esta característica las diferencia de las redes neuronales clásicas, que no tienen en cuenta el contexto temporal de la información.

La familia de las RNNs abarca diversas arquitecturas, cada una con sus propias fortalezas y aplicaciones. Entre las más populares encontramos las redes de Elman, las redes de Jordan, las redes Long Short-Term Memory (LSTM) y las redes Gated Recurrent Unit (GRU) que, introducidas en 2015 son una alternativa más ligera y eficiente a las LSTM.

El campo de las RNNs ha experimentado un rápido crecimiento en los últimos años, impulsado por avances en investigación y la disponibilidad de conjuntos de datos masivos. Entre las mejoras más notables encontramos las redes neuronales convolucionales recurrentes (CRNNs), las redes neuronales con atención y la integración del aprendizaje por refuerzo. Estas mejoras han ampliado aún más las capacidades de las RNNs, permitiéndolas abordar tareas cada vez más complejas y desafiantes.

Redes Generativas Adversarias

Las Generative Adversarial Networks (GAN) representan una innovadora aplicación del deep learning en la generación de contenido sintético, incluyendo imágenes, videos, música y caras extremadamente realistas. La arquitectura de una GAN consiste en dos componentes principales: un generador y un discriminador. El

generador se encarga de crear nuevos datos sintéticos, como imágenes, a partir de un vector aleatorio en el espacio latente. Por otro lado, el discriminador tiene la tarea de distinguir entre datos reales y sintéticos, es decir, determinar si una imagen proviene del conjunto de datos original o si fue creada por el generador.

El generador se implementa típicamente utilizando una red neuronal convolucional profunda, con capas especializadas que aprenden a generar características de imágenes en lugar de extraerlas de una imagen de entrada. Algunas de las capas más comunes utilizadas en el modelo del generador son la capa de muestreo (UpSampling2D) que duplica las dimensiones de la entrada, y la capa convolucional de transposición (Conv2DTranspose) que realiza una operación de convolución inversa para generar datos sintéticos.

La idea clave detrás de las GAN es el entrenamiento adversarial, donde el generador y el discriminador compiten entre sí en un juego de suma cero. Mientras el generador trata de engañar al discriminador generando datos cada vez más realistas, el discriminador mejora su capacidad para distinguir entre datos reales y sintéticos. Este proceso de competencia continua lleva a la generación de datos sintéticos de alta calidad que son indistinguibles de los datos reales para el discriminador.

En los últimos años, las GAN han experimentado avances significativos en términos de nuevas arquitecturas y técnicas de entrenamiento. Por ejemplo, se han desarrollado variantes como las Conditional GAN (cGAN), que permiten controlar las características de los datos generados, y las Progressive GAN (ProgGAN), que generan imágenes de mayor resolución de forma progresiva. Además, se han propuesto técnicas de regularización, como la penalización del gradiente o la normalización espectral, para mejorar la estabilidad y la calidad de las GAN generadas.

Las GANs han abierto un abanico de posibilidades en diversos campos como el ámbito de la generación de texto así como aplicaciones en la realidad aumentada donde permiten integrar elementos sintéticos en el mundo real de forma realista, como la creación de avatares virtuales o la superposición de información sobre objetos físicos. Asimismo, de los videojuegos, las GANs se utilizan para desarrollar personajes, escenarios y objetos virtuales de alta calidad para experiencias de juego más inmersivas.

Boltzmann Machine y Restricted Boltzmann Machine

El aprendizaje de la denominada máquina de Boltzmann (BM) se realiza a través de un algoritmo estocástico que proviene de ideas basadas en la mecánica estadística. Este prototipo de red neuronal tiene una característica distintiva y es que el uso de conexiones sinápticas entre las neuronas es simétrico.

Las neuronas son de dos tipos: visibles y ocultas. Las neuronas visibles son las que interactúan y proveen una interface entre la red y el ambiente en el que operan, mientras que las neuronas actúan libremente sin interacciones con el entorno. Esta máquina dispone de dos modos de operación. El primero es la condición de anclaje donde las neuronas están fijas por los estímulos específicos que impone el ambiente. El otro modo es la condición de libertad, donde tanto las neuronas ocultas como las visibles actúan libremente sin condiciones impuestas por el medio ambiente. Las máquinas restringidas de Boltzmann (RBM) solamente toman en cuenta aquellos modelos en los que no existen conexiones del tipo visible-visible y oculta-oculta. Estas redes también asumen que los datos de entrenamiento son independientes y están idénticamente distribuidos.

Una forma de estimar los parámetros de un modelo estocástico es calculando la máxima verosimilitud. Para ello, se hace uso de los Markov Random Fields (MRF), ya que al encontrar los parámetros que maximizan los datos de entrenamiento bajo una distribución MRF, equivale a encontrar los parámetros θ que maximizan la verosimilitud de los datos de entrenamiento, Fischer e Igel (2012). Maximizar dicha verosimilitud es el objetivo que persigue el algoritmo de entrenamiento de una RBM. A pesar de utilizar la distribución MRF, computacionalmente hablando se llega a ecuaciones inviables de implementar. Para evitar el problema anterior, las esperanzas que se obtienen de MRF pueden ser aproximadas por muestras extraídas de distribuciones basadas en las técnicas de Markov Chain Monte Carlo Techniques (MCMC). Las técnicas de MCMC utilizan un algoritmo denominado muestreo de Gibbs con el que obtenemos una secuencia de observaciones o muestras que se aproximan a partir de una distribución de verosimilitud de múltiples variables

aleatorias. La idea básica del muestreo de Gibss es actualizar cada variable posteriormente en base a su distribución condicional dado el estado de las otras variables.

Deep Belief Network

Una red Deep Belief Network tal como demostró Hinton se puede considerar como un “apilamiento de redes restringidas de Boltzmann”. Tiene una estructura jerárquica que, como es sabido, es una de las características del deep learning. Como en el anterior modelo, esta red también es un modelo en grafo estocástico, que aprende a extraer una representación jerárquica profunda de los datos de entrenamiento. Cada capa de la RBM extrae un nivel de abstracción de características de los datos de entrenamiento, cada vez más significativo; pero para ello, la capa siguiente necesita la información de la capa anterior lo que implica el uso de las variables latentes.

Estos modelos caracterizan la distribución conjunta h_k entre el vector de observaciones x y las capas ocultas, donde $x = h_0$, es una distribución condicional para las unidades visibles limitadas sobre las unidades ocultas que pertenecen a la RBM en el nivel k , y es la distribución conjunta oculta visible en la red RBM del nivel superior o de salida.

El entrenamiento de esta red puede ser híbrido, empezando por un entrenamiento no supervisado para después aplicar un entrenamiento supervisado para un mejor y más óptimo ajuste, aunque pueden aplicarse diferentes tipos de entrenamiento, Bengio et al. (2007) y Salakhutdinov (2014). Para realizar un entrenamiento no supervisado se aplica a las redes de creencia profunda con Redes restringidas de Boltzmann el método de bloque constructor que fue presentado por Hinton (2006) y por Bengio (2007).

1.2.2 Software

Como se verá en los siguientes epígrafes, la opción preferida para este módulo de Deep Learning es el software llamado Keras que está programado en Python. En términos de eficiencia y de aprendizaje Keras presenta unas ventajas importantes que se especifican más adelante.

Aunque nuestra preferencia a nivel formativo es el **uso de Keras y Tensorflow**, a continuación serán descritos los principales softwares con los que poder realizar implementaciones de arquitecturas de aprendizaje profundo: *TensorFlow*, *Keras*, *Pytorch*, *MXNET*, *Caffe* y *JAX*.

Por su parte, también presentaremos **Colaboratory Environment (Colab)**, una herramienta de Google que dispone en la web y que no requiere ninguna instalación en nuestros ordenadores. Esta propuesta de Google resulta muy interesante dado que no requiere coste alguno, se puede ejecutar desde cualquier lugar aumentando nuestros recursos a la hora de trabajar con Deep Learning y admitiendo a su vez la implementación tanto de código Python como de R.

TensorFlow

TensorFlow es una biblioteca de código abierto para el cálculo numérico desarrollada por Google. Es una de las herramientas de Deep Learning más populares y ampliamente utilizadas, conocida por su flexibilidad, escalabilidad y comunidad activa. TensorFlow ofrece una amplia gama de funciones para construir, entrenar y desplegar modelos de Deep Learning, incluyendo:

- Soporte para una variedad de arquitecturas de redes neuronales: permite construir una amplia gama de arquitecturas de redes neuronales, desde redes convolucionales y recurrentes hasta modelos de atención y redes generativas adversarias (GANs)
- Escalabilidad a grandes conjuntos de datos: está diseñado para manejar grandes conjuntos de datos y puede distribuirse en múltiples GPUs o TPU para acelerar el entrenamiento de modelos

- Amplia gama de herramientas de visualización y depuración: proporciona una variedad de herramientas para visualizar y depurar modelos de Deep Learning, lo que facilita la identificación y resolución de problemas
- Gran comunidad y recursos: cuenta con una gran y activa comunidad de desarrolladores y usuarios que proporcionan soporte y comparten recursos

Pytorch

PyTorch es una biblioteca de código abierto para el aprendizaje automático desarrollada por Facebook. Es conocida por su sintaxis intuitiva y facilidad de uso, lo que la convierte en una opción popular para investigadores y desarrolladores principiantes. PyTorch ofrece características similares a TensorFlow, incluyendo:

- Soporte para una variedad de arquitecturas de redes neuronales: permite construir una amplia gama de arquitecturas de redes neuronales, desde redes convolucionales y recurrentes hasta modelos de atención y GANs
- Ejecución dinámica de gráficos: utiliza un motor de ejecución de gráficos dinámico, lo que permite modificar los modelos durante el entrenamiento, lo que facilita la experimentación y el ajuste fino
- Amplia gama de bibliotecas y herramientas de terceros: se beneficia de un ecosistema rico de bibliotecas y herramientas de terceros que amplían sus capacidades
- Facilidad de uso: tiene una sintaxis similar a Python, lo que la hace fácil de aprender y usar para desarrolladores con experiencia en Python

Keras

Keras es una biblioteca de código abierto para el aprendizaje automático de alto nivel que se ejecuta sobre TensorFlow o PyTorch. Es conocida por su simplicidad y facilidad de uso, lo que la convierte en una opción popular para principiantes y para desarrollar prototipos de modelos rápidamente. Keras ofrece una interfaz de alto nivel que abstrae las complejidades de las bibliotecas subyacentes, como TensorFlow o PyTorch, lo que permite a los usuarios centrarse en la construcción y el entrenamiento de modelos sin necesidad de profundizar en los detalles de implementación. Entre las principales características de Keras destaca:

- Simplicidad: tiene una sintaxis intuitiva y fácil de aprender, lo que la hace ideal para principiantes y para desarrollar prototipos de modelos rápidamente
- Facilidad de uso: ofrece una API de alto nivel que abstrae las complejidades de las bibliotecas subyacentes, como TensorFlow o PyTorch, lo que permite a los usuarios centrarse en la construcción y el entrenamiento de modelos sin necesidad de profundizar en los detalles de implementación
- Flexibilidad: permite construir una amplia gama de modelos de Deep Learning, desde redes neuronales convolucionales y recurrentes hasta modelos de atención y redes generativas adversarias (GANs)
- Modularidad: al ser una biblioteca modular que permite a los usuarios combinar diferentes componentes para construir sus modelos personalizados
- Soporte para múltiples plataformas: se puede utilizar en una variedad de plataformas, incluyendo Windows, macOS y Linux.

JAX

JAX, desarrollada por Google Research, se posiciona como una biblioteca de Python para el aprendizaje automático y el cálculo numérico, diseñada para ofrecer un rendimiento y una flexibilidad excepcionales, especialmente en el entrenamiento de modelos de deep learning en aceleradores como GPUs y TPUs.

Su enfoque se basa en la composición de funciones puras y transformaciones automáticas de gradiente, lo que la convierte en una herramienta ideal para implementar algoritmos de aprendizaje automático diferenciables y de alto rendimiento. Entre sus características destacadas encontramos:

- Autodiferenciación: calcula automáticamente gradientes (autodiferenciación), simplificando el desarrollo de modelos de deep learning
- Composición eficiente de transformaciones: combina operaciones elementales en funciones compuestas para un procesamiento eficiente
- Integración con frameworks: se integra con frameworks de deep learning como TensorFlow y PyTorch, aprovechando las ventajas de cada uno
- Paralelización y distribución: permite ejecutar operaciones en paralelo y de manera distribuida, ideal para grandes conjuntos de datos
- Altas prestaciones para el entrenamiento de modelos: sobresale por su capacidad de computación de alto rendimiento, haciéndola ideal para entrenar modelos de deep learning complejos de manera eficiente. Así, se ha convertido en una opción atractiva para aquellos que manejan grandes conjuntos de datos y buscan optimizar el tiempo de entrenamiento
- Flexibilidad para la investigación y experimentación: facilita la implementación de nuevas arquitecturas y algoritmos, permitiendo explorar diferentes enfoques y optimizar el rendimiento de los modelos
- Personalización de flujos de trabajo: permite definir funciones y transformaciones personalizadas, proporcionando un control preciso sobre el pipeline de trabajo. Esto resulta útil para adaptar el proceso de entrenamiento a necesidades específicas y optimizar el rendimiento para tareas concretas

Mxnet

MXNet es una biblioteca de código abierto para el aprendizaje automático desarrollada por Apache Software Foundation. Es conocida por su escalabilidad, flexibilidad y soporte para múltiples lenguajes de programación, incluyendo Python, R y C++. MXNet ofrece características similares a TensorFlow y PyTorch, incluyendo:

- Soporte para una variedad de arquitecturas de redes neuronales: permite construir una amplia gama de arquitecturas de redes neuronales, desde redes convolucionales y recurrentes hasta modelos de atención y GANs
- Escalabilidad a grandes conjuntos de datos: está diseñado para manejar grandes conjuntos de datos y puede distribuirse en múltiples GPUs o TPU para acelerar el entrenamiento de modelos
- Soporte para múltiples lenguajes de programación: se puede utilizar con Python, R y C++, lo que lo hace accesible a una amplia gama de desarrolladores
- Flexibilidad: permite a los usuarios personalizar y extender la biblioteca para satisfacer sus necesidades específicas

Caffe

Caffe es un marco de código abierto para el aprendizaje profundo desarrollado por la Universidad de California, Berkeley. Es conocido por su simplicidad, velocidad y eficiencia, lo que lo convierte en una opción popular para aplicaciones de Deep Learning en tiempo real. Caffe ofrece características similares a TensorFlow y PyTorch, incluyendo:

- Soporte para una variedad de arquitecturas de redes neuronales: permite construir una amplia gama de arquitecturas de redes neuronales, desde redes convolucionales y recurrentes hasta modelos de atención y GANs
- Entrenamiento rápido y eficiente: está optimizado para el rendimiento y la eficiencia, lo que lo hace ideal para aplicaciones de aprendizaje profundo

1.2.2.1 Google Colab

El entorno Colab (Google Colaboratory) es una potente herramienta de google para ejecutar código incluido el deep Dearning y que está disponible en la web (<https://colab.research.google.com/>). Se ha desarrollado

para Python, pero actualmente también se puede ejecutar código de R. Esta funcionalidad puede importar un conjunto de datos de imágenes, entrenar un clasificador con este conjunto de datos y evaluar el modelo con tan solo usar unas pocas líneas de código. Los cuadernos de Colab ejecutan código en los servidores en la nube de Google, lo que nos permite aprovechar la potencia del hardware de Google, incluidas las GPU y TPU, independientemente de la potencia de tu equipo. Lo único que se necesita es un navegador.

Con Colab se puede aprovechar toda la potencia de las bibliotecas más populares de Python para analizar y visualizar datos. La celda de código de abajo utiliza NumPy para generar datos aleatorios y Matplotlib para visualizarlos. Para editar el código, solo se tiene que hacer clic en la celda.

```

import torch
import torch.utils.data as Data
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms

import numpy as np
import matplotlib.pyplot as plt

```

{#fig-colab_1}

Este es el menú principal de colab desde donde podemos gestionar nuestros proyectos:

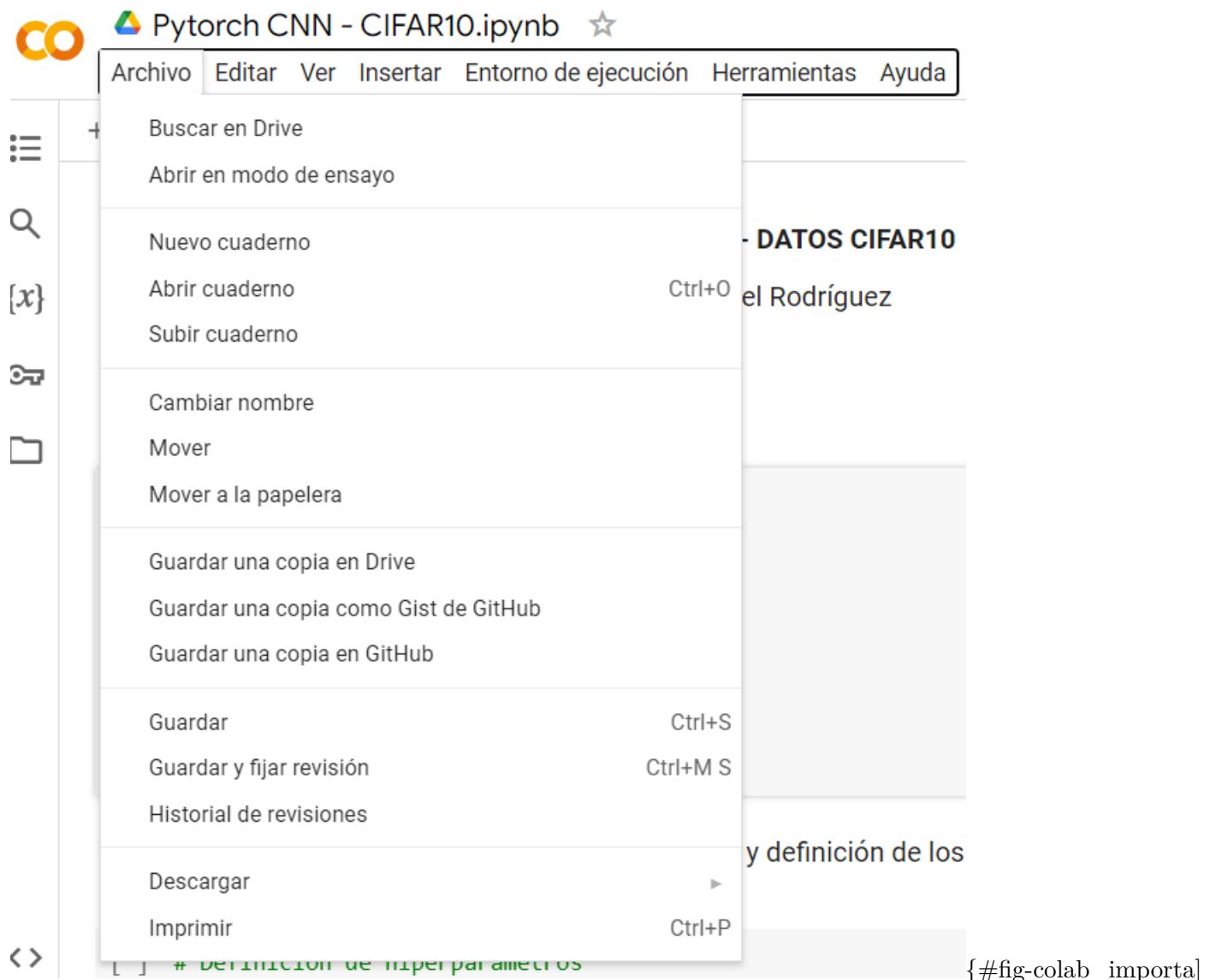
```

colab_nuevo_fichero]

```

{#fig-colab_nuevo_fichero}

Desde el menú Archivo, como en la mayor parte de los programas, podemos llevar a cabo las operaciones habituales de abrir y guardar los ficheros en diferentes formatos. En este caso se pueden abrir ficheros de Jupyter/Python desde cualquier dispositivo externo, desde el repositorio Drive o de Github:



Si queremos subir un fichero que tenemos en nuestro ordenador vamos a Archivo/Subir cuaderno y podemos elegir nuestro archivo cuando se despliegue la siguiente pantalla:

Abrir cuaderno

- Ejemplos >
- Recientes >
- Google Drive >
- GitHub >
- Subir >**



Explorar

o arrastra un archivo aquí

Cancelar

{#fig-

colab_importa_archivo]

Como se ha comentado también se pueden importar archivos desde GitHub introduciendo la url de GitHub:

Abrir cuaderno

- Ejemplos >
- Recientes >
- Google Drive >
- GitHub >**
- Subir >

Escribe una URL de GitHub o busca por organización o usuario

https://github.com/pablosaca/TUTOR_UNED



Incluir repositorios privados

Repositorio: [pablosaca/TUTOR_UNED](#)

Rama: [master](#)

Google Drive > [pablosaca/TUTOR_UNED](#)

master

- GitHub >**

Ruta

[docs/user_guides/User-Guide.ipynb](#)



[notebooks/Desarrollo.ipynb](#)

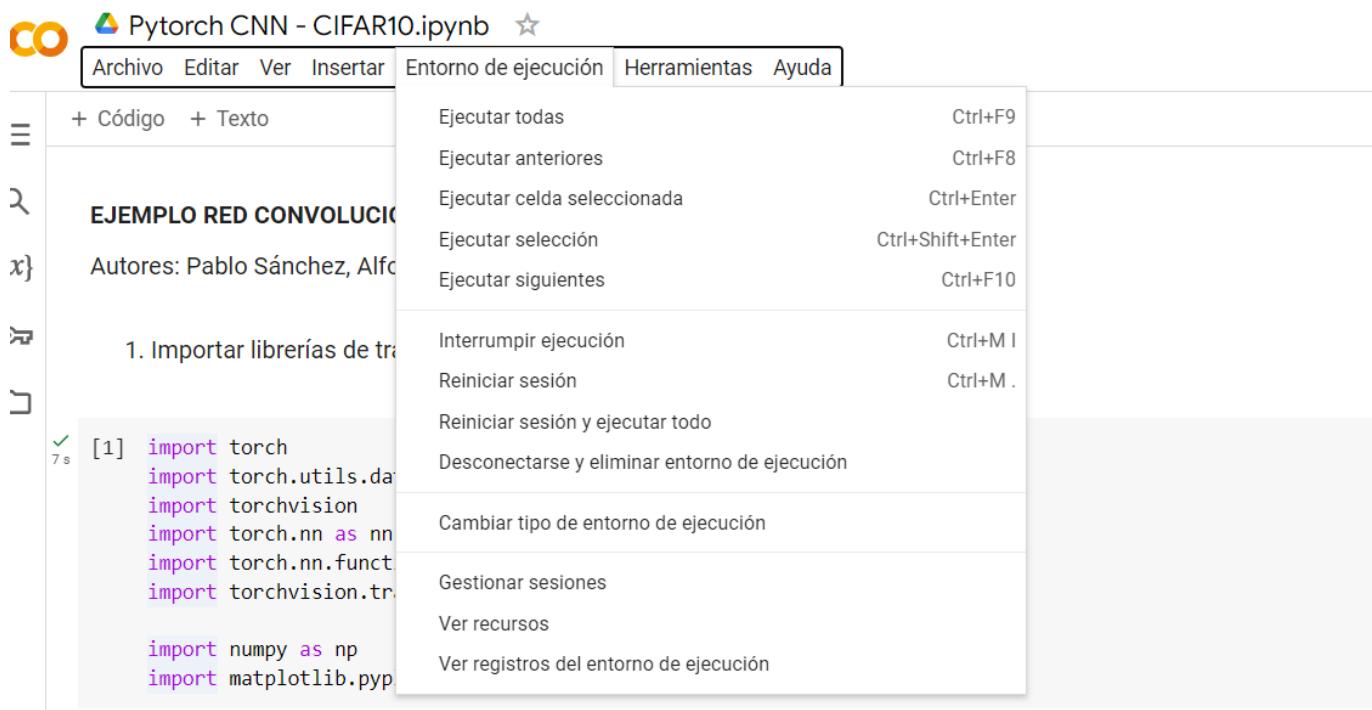


Cancelar

{#fig-

```
colab_importa_github]
```

Por último, *colab* nos permite tener acceso tanto a GPUs de forma como a CPUs más potentes que nuestro ordenador de escritorio de forma gratuita.



```
colab_entorno_ejecucion_inicio]
```

{#fig-}

1.3 Conceptos básicos de las Redes Neuronales

Vamos a hacer una revisión de las redes neuronales para posteriormente poder abordar los diferentes tipos de redes neuronales que se utilizan en Deep Learning. Algunos de los avances más recientes en varios de los diferentes componentes que forman parte de las redes neuronales están recopilados en (Gu et al. 2017)

Las redes neuronales artificiales tienen sus orígenes en el Perceptrón, que fue el modelo creado por Frank Rosenblatt en 1957 y basado en los trabajos que previamente habían realizado Warren McCullon (neurofisiólogo) y Walter Pitts (matemático).

El Perceptrón está construido por una neurona artificial cuyas entradas y salida pueden ser datos numéricos, no como pasaba con la neurona de McCulloch y Pitts (eran sólo datos lógicos). Las neuronas pueden tener pesos y además se le aplica una función de activación Sigmoid (a diferencia de la usada anteriormente al Paso binario).

En esta neurona nos encontramos que se realizan los siguientes cálculos:

$$z = \sum_{i=1}^n w_i x_i + b_i$$

$$\hat{y} = \delta(z)$$

donde representan los datos numéricos de entrada, son los pesos, es el sesgo (bias), es la función de activación y finalmente es el dato de salida.

El modelo de perceptrón es el más simple, en el que hay una sola capa oculta con una única neurona.

El siguiente paso nos lleva al Perceptrón Multicapa donde ya pasamos a tener más de una capa oculta, y además podemos tener múltiples neuronas en cada capa oculta.

Cuando todas las neuronas de una capa están interconectadas con todas las de la siguiente capa estamos ante una red neuronal densamente conectada. A lo largo de las siguientes secciones nos encontraremos con redes en las que no todas las neuronas de una capa se conectan con todas de la siguiente.

Veamos como describiríamos ahora los resultados de las capas

$$z_j^{(l)} = \sum_{i=1}^{n_j} w_{ij}^{(l)} a_i^{(l-1)} + b_i^{(l)} a_j^{(l)} = \delta^{(l)}(z_j^{(l)})$$

donde $a_i^{(l-1)}$ representan los datos de la neurona i en la capa $l-1$ (siendo $a_i^0 = x_i$ los valores de entrada), $w_{ij}^{(l)}$ son los pesos en la capa l , $b_i^{(l)}$ es el sesgo (bias) en la capa l , $\delta^{(l)}$ es la función de activación en la capa l (puede que cada capa tenga una función de activación diferente), n_j es el número de neurona de la capa anterior que conectan con la j y finalmente $a_j^{(l)}$ es el dato de salida de la capa l . Es decir, en cada capa para calcular el nuevo valor necesitamos usar los valores de la capa anterior.

Aplicaciones de las Redes Neuronales

Cada día las redes neuronales están más presentes en diferentes campos y ayudan a resolver una gran variedad de problemas. Podríamos pensar que de forma más básica una red neuronal nos puede ayudar a resolver problemas de regresión y clasificación, es decir, podríamos considerarlo como otro modelo más de los existentes que a partir de unos datos de entrada somos capaces de obtener o un dato numérico (o varios) para hacer una regresión (calcular el precio de una vivienda en función de diferentes valores de la misma) o que somos capaces de conseguir que en función de los datos de entrada nos deje clasificada una muestra (decidir si conceder o no una hipoteca en función de diferentes datos del cliente).

Si los datos de entrada son imágenes podríamos estar usando las redes neuronales como una forma de identificar esa imagen:

- Identificando que tipo de animal es
- Identificando que señal de tráfico es
- Identificando que tipo de fruta es
- Identificando que una imagen es de exterior o interior de una casa
- Identificando que es una cara de una persona
- Identificando que una imagen radiográfica represente un tumor maligno
- Identificando que haya texto en una imagen

Luego podríamos pasar a revolver problemas más complejos combinando las capacidades anteriores:

- Detectar los diferentes objetos y personas que se encuentran en una imagen
- Etiquetado de escenas (aula con alumnos, partido de fútbol, etc...)

Después podríamos dar el paso al video que lo podríamos considerar como una secuencia de imágenes:

- Contar el número de personas que entran y salen de una habitación
- Reconocer que es una carretera

- Identificar las señales de tráfico
- Detectar si alguien lleva un arma
- Seguimiento de objetos
- Detección de estado/actitud de una persona
- Reconocimiento de acciones (interpretar lenguaje de signos, interpretar lenguaje de banderas)
- Vehículos inteligentes

Si los datos de entrada son secuencias de texto

- Sistemas de traducción - Chatbots (resolución de preguntas a usuarios)
- Conversión de texto a audio

Si los datos de entrada son audios

- Sistemas de traducción
- Altavoces inteligentes
- Conversión de audio a texto

A continuación, pasamos a revisar diferentes elementos de las redes neuronales que suelen ser comunes a todos los tipos de redes neuronales.

1.3.1 Datos

Cuando se trabaja con redes neuronales necesitamos representar los valores de las variables de entrada en forma numérica. En una red neuronal todos los datos son siempre numéricos. Esto significa que todas aquellas variables que sean categóricas necesitamos convertirlas en numéricas.

Además, es muy conveniente normalizar los datos para poder trabajar con valores entre 0 y 1, que van a ayudar a que sea más fácil que se pueda converger a la solución. Es importante que los datos seán números en coma flotante, sobre todo si se van a trabajar con GPUs (Graphics Process Units), ya que permitirán hacer un mejor uso de los multiples cores que les permiten operar en coma flotante de forma paralela. Actualmente, hay toda una serie de mejoras en las GPUs que permite aumentar el rendimiento de las redes neuronales como son el uso de operaciones en FP16 (Floating Point de 16 bits en lugar de 32) de forma que pueden hacer dos operaciones de forma simultánea (el formato estándar es FP32) y además con la reducción de memoria (punto muy importante) al meter en los 32 bits 2 datos en lugar de sólo uno. También se han añadido técnicas de Mixed Precision (Narang et al. 2018), los Tensor Cores (para las gráficas de NVIDIA) son otra de las mejoras que se han ido incorporando a la GPUs y que permiten acelerar los procesos tanto de entrenamiento como de predicción con las redes neuronales.

El primer objetivo será convertir las variables categóricas en variables numéricas, de forma la red neuronal pueda trabajar con ellas. Para realizar la conversión de categórica a numérica básicamente tenemos dos métodos para realizarlo:

- Codificación one-hot.
- Codificación entera.

La **codificación one-hot** consiste en crear tantas variables como categorías tenga la variable, de forma que se asigna el valor 1 si tiene esa categoría y el 0 si no la tiene.

La **codificación entera** lo que hace es codificar con un número cada categoría. Realmente esta asignación no tiene ninguna interpretación numérica ya que en general las categorías no tienen porque representar un orden al que asociarlas.

Normalmente se trabaja con codificación one-hot para representar los datos categóricos de forma que será necesario preprocessar los datos de partida para realizar esta conversión, creando tantas variables como categorías haya por cada variable.

Si nosotros tenemos nuestra muestra de datos que tiene n variables $x = \{x_1, x_2, \dots, x_n\}$ de forma que x_{n-2}, x_{n-1}, x_n son variables categóricas que tienen k, l, m número de categorías respectivamente, tendremos finalmente las siguientes variables sólo numéricas:

$$x = \{x_1, x_2, \dots, x_{(n-2)_1}, \dots, x_{(n-2)_k}, x_{(n-1)_1}, \dots, x_{(n-1)_l}, x_{n_1}, \dots, x_{n_m}\}$$

De esta forma, se aumentarán el número de variables con las que vamos a trabajar en función de las categorías que tengan las variables categóricas. Normalmente nos encontramos que en una red neuronal las variables de salida son:

- un número (regresión)
- una serie de números (regresión múltiple)
- un dato binario (clasificación binaria)
- una serie de datos binarios que representa una categoría de varias (clasificación múltiple)

1.3.2 Arquitectura de red

Para la construcción de una red neuronal necesitamos definir la arquitectura de esa red. Esta arquitectura, si estamos pensando en una red neuronal densamente conectada, estará definida por la cantidad de capas ocultas y el número de neuronas que tenemos en cada capa. Más adelante veremos que dependiendo del tipo de red neuronal podrá haber otro tipo de elementos en estas capas.

1.3.3 Función de coste y pérdida

Otro de los elementos clave que tenemos que tener en cuenta a la hora de usar nuestra red neuronal son las **funciones de pérdida y funciones de coste (objetivo)**.

La función de pérdida va a ser la función que nos dice cómo de diferente es el resultado del dato que nosotros queríamos conseguir respecto al dato original. Normalmente se suelen usar diferentes tipos de funciones de pérdida en función del tipo de resultado con el que se vaya a trabajar.

La función de coste es la función que vamos a tener que **optimizar** para conseguir el mínimo valor posible, y que recoge el valor de la función de pérdida para toda la muestra.

Tanto las funciones de pérdida como las funciones de coste, son funciones que devuelven valores de \mathbb{R} .

Si tenemos un problema de **regresión** en el que tenemos que predecir un valor o varios valores numéricos, algunas de las funciones a usar son:

- **Error medio cuadrático (L_2^2)**

$$\mathcal{L}_{\text{MSE}}(y, \hat{y}) = \|\hat{y} - y\|^2 = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

donde \hat{y} y y son vectores de tamaño n , y es el valor real e \hat{y} es el valor predicho

- **Error medio absoluto (L_1)**

$$\mathcal{L}_{\text{MAE}}(y, \hat{y}) = |\hat{y} - y| = \sum_{i=0}^n |\hat{y}_i - y_i|$$

donde \hat{y} y y son vectores de tamaño n , y es el valor real e \hat{y} es el valor predicho

Para los problemas de **clasificación**:

- **Binary Crossentropy (Sólo hay dos clases)**

$$\mathcal{L}_{\text{CRE}}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

y es el valor real e \hat{y} es el valor predicho

- **Categorical Crosentropy (Múltiples clases representadas como one-hot)**

$$\mathcal{L}_{\text{CAE}}(y_c, \hat{y}_c) = - \sum_{c=1}^k y_c \log(\hat{y}_c)$$

y_c es el valor real para la clase c e \hat{y}_c es el valor predicho para la clase c

- **Sparse Categorical Crossentropy (Múltiples clases representadas comp un entero)**

$$\mathcal{L}_{\text{SCAE}}(y_c, \hat{y}_c) = - \sum_{c=1}^k y_c \log(\hat{y}_c)$$

y_c es el valor real para la clase c e \hat{y}_c es el valor predicho para la clase c

- **Kullback-Leibler Divergence**

Esta función se usa para calcular la diferencia entre dos distribuciones de probabilidad se usa por ejemplo en algunas redes como **Variational Autoencoders** (Doersch 2016 Modelos GAN (Generative Adversarial Networks))

$$\mathcal{D}_{\text{KL}}(p\|q) = -H(p(x)) - E_p[\log q(x)]$$

$$= \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

$$\mathcal{L}_{\text{vae}}(y, \hat{y}) = E_{z \sim q_\phi(z|x)} [\log p_\theta(x | z)] - \mathcal{D}_{\text{KL}}(q_\phi(z | x) \| p(z))$$

- **Hinge Loss**

$$\mathcal{L}_{\text{hinge}}(y, \hat{y}) = \max(0, 1 - y * \hat{y})$$

Las correspondientes **funciones de coste** que se usarían, estarían asociadas a todas las muestras que se estén entrenando o sus correspondientes batch, así como posibles términos asociados a la regularización para evitar el sobreajuste del entrenamiento. Es decir, la función de pérdida se calcula para cada muestra, y la función de coste es la media de todas las muestras.

Por ejemplo, para el **Error medio cuadrático** (L_2) tendríamos el siguiente valor:

$$\mathcal{J}_{\text{MSB}}(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}_{\text{MSE}}(y_i, \hat{y}_i) = \frac{1}{m} \sum_{i=1}^m \|\hat{y}_i - y_i\|^2 = \frac{1}{n} \sum_{i=1}^m \sum_{j=1}^n (\hat{y}_{ji} - y_{ji})^2$$

1.3.4 Optimizador

El **Descenso del gradiente** es la versión más básica de los algoritmos que permiten el aprendizaje en la red neuronal haciendo el proceso de **backpropagation** (propagación hacia atrás). A continuación veremos una breve explicación del algoritmo así como algunas variantes del mismo recogidas en (Ruder 2017).

Recordamos que el descenso del gradiente nos permitirá actualizar los parámetros de la red neuronal cada vez que demos una pasada hacia delante con todos los datos de entrada, volviendo con una pasada hacia atrás.

$$w_t = w_{t-1} - \alpha \nabla_w \mathcal{J}(w)$$

donde \mathcal{J} es la **función de coste**, α es el parámetro de **ratio de aprendizaje** que permite definir como de grandes se quiere que sean los pasos en el aprendizaje.

Cuando lo que hacemos es actualizar los parámetros para cada pasada hacia delante de una sola muestra, estaremos ante lo que llamamos **Stochastic Gradient Descent** (SGD). En este proceso convergerá en menos iteraciones, aunque puede tener alta varianza en los parámetros.

$$W_t = W_{t-1} - \alpha \nabla_w \mathcal{J}(w, x(i), y(i))$$

donde $x(i)$ e $y(i)$ son los valores en la pasada de la muestra i .

Podemos buscar un punto intermedio que sería cuando trabajamos por lotes y cogemos un bloque de datos de la muestra, les aplicamos la pasada hacia delante y aprendemos los parámetros para ese bloque. En este caso lo llamaremos **Mini-batch Gradient Descent**

$$W_t = W_{t-1} - \alpha \nabla_w \mathcal{J}(w, B(i))$$

donde $B(i)$ son los valores de ese batch .

En general a estos métodos nos referiremos a ellos como **SGD**.

Sobre este algoritmo base se han hecho ciertas mejoras como:

Learning rate decay Podemos definir un valor de decenso del ratio de aprendizaje, de forma que normalmente al inicio de las iteraciones de la red neuronal los pasos serán más grandes, pero conforme nos acercamos a la solución optima deberemos dar pasos más pequeños para ajustarnos mejor.

$$W_t = w_{t-1} - \alpha_t \nabla_w \mathcal{J}(w_{t-1})$$

donde α_t ahora se irá reduciendo en función del valor del **decay**.

Momentum El **momentum** se introdujo para suavizar la convergencia y reducir la alta varianza de SGD.

$$\begin{aligned} V_t &= \gamma v_{t-1} + \alpha V_w J(w_{t-1}, x, y) \\ W_t &= w_{t-1} - v_t \end{aligned}$$

donde v_t es lo que se llama el **vector velocidad** con la dirección correcta.

NAG (Nesterov Accelerated Gradient) Ahora daremos un paso más con el NAG, calculando la función de coste junto con el vector velocidad.

$$\begin{aligned} V_t &= \gamma v_{t-1} + \alpha V_w J(w_{t-1} - \gamma v_{t-1}, x, y) \\ W_t &= w_{t-1} - v_t \end{aligned}$$

donde ahora vemos que la función de coste se calcula usando los parámetros de w_t sumado a γv_{t-1}

Veamos algunos algoritmos de optimización más que, aunque provienen del SGD, se consideran independientes a la hora de usarlos y no como parámetros extras del SGD.

Adagrad (Adaptive Gradient) Esta variante del algoritmo lo que hace es adaptar el ratio de aprendizaje para cada uno de los pesos en lugar de que sea global para todos.

$$W_{t,i} = w_{t-1,i} - \frac{\alpha}{\sqrt{G_{t-1,i,j} + \epsilon}} \nabla_{w_{t-1}} J(w_{t-1,i}, x, y)$$

donde tenemos que $G_t \in R^{d \times d}$ es una matriz diagonal donde cada elemento es la suma de los cuadrados de los gradientes en el paso $t-1$, y es un término de suavizado para evitar divisiones por 0.

RMSProp (Root Mean Square Propagation) En este caso tenemos una variación del Adagrad en el que intenta reducir su agresividad reduciendo monotonamente el ratio de aprendizaje. En lugar de usar el gradiente acumulado desde el principio de la ejecución, se restringe a una ventana de tamaño fijo para los últimos n gradientes calculando su media. Así calcularemos primero la media en ejecución de los cuadros de los gradientes como:

$$E[g^2]_{t-1} = \gamma E[g^2]_{t-2} + (1 - \gamma) g_{t-1}^2$$

y luego ya pasaremos a usar este valor en la actualización

$$w_{t,i} = w_{t-1,i} - \frac{\alpha}{\sqrt{E[g^2]_{t-1} + \epsilon}} \nabla_{w_{t-1}} \mathcal{J}(w_{t-1,i}, x, y)$$

AdaDelta

Aunque se desarrollaron de forma simultánea el AdaDelta y el RMSProp son muy parecidos en su primer paso inicial, llegando el de AdaDelta un poco más lejos en su desarrollo.

$$w_{t,i} = w_{t-1,i} - \frac{\alpha}{\sqrt{E[g^2]_{t-1} + \epsilon}} \nabla_{w_{t-1}} \mathcal{J}(w_{t-1,i}, x, y)$$

y luego ya pasaremos a usar este valor en la actualización

$$\begin{aligned} w_{t,i} &= w_{t-1,i} - \frac{\alpha}{\sqrt{E[g^2]_{t-1} + \epsilon}} \nabla_{w_{t-1}} \mathcal{J}(w_{t-1,i}, X, y) \\ \Delta w_t &= -\frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned}$$

Adam (Adaptive Moment Estimation)

$$\begin{aligned} G_t &= \nabla_{w_t} \mathcal{J}(w_t) \\ M_{t-1} &= \beta_1 m_{t-2} + (1 - \beta_1) g_{t-1} \\ v_{t-1} &= \beta_2 v_{t-2} + (1 - \beta_2) g_{t-1}^2 \end{aligned}$$

donde m_{t-1} y V_{t-1} son estimaciones del primer y segundo momento de los gradientes respectivamente, y β_1 y β_2 parámetros a asignar.

$$\widehat{M}_{t-1} = \frac{m_{t-1}}{1 - \beta_1^{t-1}} \widehat{V}_{t-1} = \frac{v_{t-1}}{1 - \beta_2^{t-1}} W_t = w_{t-1} - \frac{\alpha}{\sqrt{\widehat{v}_{t-1} + \epsilon}} \widehat{m}_{t-1}$$

Adamax

$$G_t = \nabla_{w_t} \mathcal{J}(w_t) M_{t-1} = \beta_1 m_{t-2} + (1 - \beta_1) g_{t-1} V_{t-1} = \beta_2 v_{t-2} + (1 - \beta_2) g_{t-1}^2 U_{t-1} = \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

donde m_{t-1} y V_{t-1} son estimaciones del primer y segundo momento de los gradientes respectivamente, y β_1 y β_2 parámetros a asignar.

$$\widehat{M}_{t-1} = \frac{m_{t-1}}{1 - \beta_1^{t-1}} W_t = w_{t-1} - \frac{\alpha}{u_{t-1}} \widehat{m}_{t-1}$$

Nadam (Nesterov-accelerated Adaptive Moment Estimation) Combina Adam y NAG.

$$\begin{aligned} G_t &= \nabla_{w_t} \mathcal{J}(w_t) \\ M_{t-1} &= \gamma m_{t-2} + \alpha g_{t-1} \\ w_t &= w_{t-1} - m_{t-1} \end{aligned}$$

1.3.5 Función de activación

Las funciones de activación dentro de una red neuronal son uno de los elementos clave en el diseño de la misma. Cada tipo de función de activación podrá ayudar a la convergencia de forma más o menos rápida en función del tipo de problema que se plantea. En una red neuronal las funciones de activación en las capas ocultas van a conseguir establecer las restricciones **no lineales** al pasar de una capa a la siguiente, normalmente se evita usar la función de activación lineal en las capas intermedias ya que queremos conseguir transformaciones no lineales.

A continuación, exponemos las principales funciones de activación en las capas ocultas:

- **Paso binario** (Usado por los primeros modelos de neuronas)

$$F(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

- **Identidad**

$$F(x) = x$$

- **Sigmoid (Logística)**

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

- **Tangente Hiperbólica (Tanh)**

$$F(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

- **Softmax**

$$F(x_i) = \frac{e^{x_i}}{\sum_{j=0}^k e^{x_j}}$$

$$F(x) = \max(0, x)$$

- **ReLU (Rectified Linear Unit)** $f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$

- **LReLU (Leaky Rectified Linear Unit)** $F(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

- **PReLU (Parametric Rectified Linear Unit)** $F(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

- **RReLU (Randomized Rectified Linear Unit)** $F(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

*La diferencia entre LReLU, PReLU y RReLU es que en LReLU el parámetro es uno que se asigna fijo, en el caso de PReLU el parámetro también se aprende durante el entrenamiento y finalmente en RReLU es un parámetro con valores entre 0 y 1, que se obtiene de un muestreo en una distribución normal.

Se puede profundizar en este grupo de funciones de activación en (Xu et al. 2015)

- **ELU (Exponential Linear Unit)** $F(\alpha, x) = \begin{cases} \alpha(e^{x-1}) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

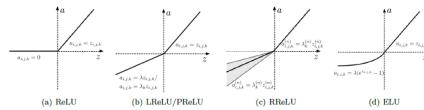


Imagen 1.1: Funciones ReLU

Función de activación en salida

En la capa de salida tenemos que tener en cuenta cual es el tipo de datos final que queremos obtener, y en función de eso elegiremos cual es la función de activación de salida que usaremos. Normalmente las funciones de activación que se usarán en la última capa seran:

- **Lineal** con una unidad, para regresión de un solo dato numérico $F(x) = x$ donde x es un valor escalar.
- **Lineal** con multiples unidades, para regresión de varios datos numéricos $F(x) = x$ donde x es un vector.
- **Sigmoid** para clasificación binaria $F(x) = \frac{1}{1+e^{-x}}$
- **Softmax** para calsifiación múltiple $F(x_i) = \frac{e^{x_i}}{\sum_{j=0}^k e^{x_j}}$

1.3.6 Regularización

Las técnicas de regularización nos permiten conseguir mejorar los problemas que tengamos por sobreajuste en el entrenamiento de nuestra red neuronal.

A continuación, vemos algunas de las técnicas de regularización existentes en la actualidad:

- **Norma LP** Básicamente estos métodos tratan de hacer que los pesos de las neuronas tengan valores muy pequeños consiguiendo una distribución de pesos más regular. Esto lo consiguen al añadir a la función de pérdida un coste asociado a tener pesos grandes en las neuronas. Este peso se puede construir o bien con la **norma L1** (proporcional al valor absoluto) o con la **norma L2** (proporcional al cuadrado de los coeficientes de los pesos). En general se define la norma LP

$$E(w, \mathbf{y}, \hat{\mathbf{y}}) = \mathcal{L}(w, \mathbf{y}, \hat{\mathbf{y}}) + \lambda R(w)$$

$$R(w) = \sum_j \|w_j\|_p^p$$

Para los casos más habituales tendríamos la norma **L1** y **L2**.

$$R(w) = \sum_j \|w_j\|^2$$

$$R(w) = \sum_j |w_j|$$

1.3.7 Dropout

Una de las técnicas de regularización que más se están usando actualmente es la llamada **Dropout**, su proceso es muy sencillo y consiste en que en cada iteración de forma aleatoria se dejan de usar un porcentaje de las neuronas de esa capa, de esta forma es más difícil conseguir un sobreajuste porque las neuronas no son capaces de memorizar parte de los datos de entrada.

1.3.8 Dropconnect

El Dropconnect es otra técnica que va un poco más allá del concepto de Dropout y en lugar de usar en cada capa de forma aleatoria una serie de neuronas, lo que se hace es que de forma aleatoria se ponen los pesos de la capa a cero. Es decir, lo que hacemos es que hay ciertos enlaces de alguna neurona de entrada con alguna de salida que no se activan.

1.3.9 Inicialización de pesos

Cuando empieza el entrenamiento de una red neuronal y tiene que realizar la primera pasada hacia delante de los datos, necesitamos que la red neuronal ya tenga asignados algún valor a los pesos.

Se pueden hacer inicializaciones del tipo:

- **Ceros** Todos los pesos se inicializan a 0.
- **Unos** Todos los pesos se inicializan a 1.
- **Distribución normal**. Los pesos se inicializan con una distribución normal, normalmente con media 0 y una desviación alrededor de 0,05. Es decir, valores bastante cercanos al cero.
- **Distribución normal truncada**. Los pesos se inicializan con una distribución normal, normalmente con media 0 y una desviación alrededor de 0,05 y además se truncan con un máximo del doble de la desviación. Los valores aun son más cercanos a cero.
- **Distribución uniforme**. Los pesos se inicializan con una distribución uniforme, normalmente entre el 0 y el 1.
- **Glorot Normal** (También llamada Xavier normal) Los pesos se inicializan partiendo de una distribución normal truncada en la que la desviación es donde es el número de unidades de entrada y fanout es el número de unidades de salida. Ver (Glorot and Bengio 2010)
- **Glorot Uniforme** (También llamada Xavier uniforme) Los pesos se inicializan partiendo de una distribución uniforme donde los límites son $[-\text{limit}, +\text{limit}]$ donde $\text{limit} = \sqrt{\frac{6}{\text{fanin} + \text{fanout}}}$ donde fanin y es el número de unidades de entrada y fanout es el número de unidades de salida. Ver (Glorot and Bengio 2010)

1.3.10 Batch normalization

Hemos comentado que cuando entrenamos una red neuronal los datos de entrada deben ser todos de tipo numérico y además los normalizamos para tener valores “cercanos a cero”, teniendo una media de 0 y varianza de 1, consiguiendo uniformizar todas las variables y conseguir que la red pueda converger más fácilmente.

Cuando los datos entran a la red neuronal y se comienza a operar con ellos, se convierten en nuevos valores que han perdido esa propiedad de normalización. Lo que hacemos con la normalización por lotes (batch normalization) (Ioffe and Szegedy 2015) es que añadimos un paso extra para normalizar las salidas de las funciones de activación. Lo normal es que se aplicara la normalización con la media y la varianza de todo el bloque de entrenamiento en ese paso, pero normalmente estaremos trabajando por lotes y se calculará la media y varianza con ese lote de datos.

1.4 Redes Neuronales Convolucionales

1.4.1 Introducción

Esta arquitectura de redes de neuronas convolucionales, CNN, Convolutional Neural Networks es en la actualidad el campo de investigación más fecundo dentro de las redes neuronales artificiales de Deep learning y donde los investigadores, empresas e instituciones están dedicando más recursos e investigación. Para apoyar esta aseveración, en google trend se observa que el término convolutional neural network en relación con el concepto de artificial neural network crece y está por encima desde el año 2016. Es en este último lustro donde el Deep learning ha tomado una importancia considerable.

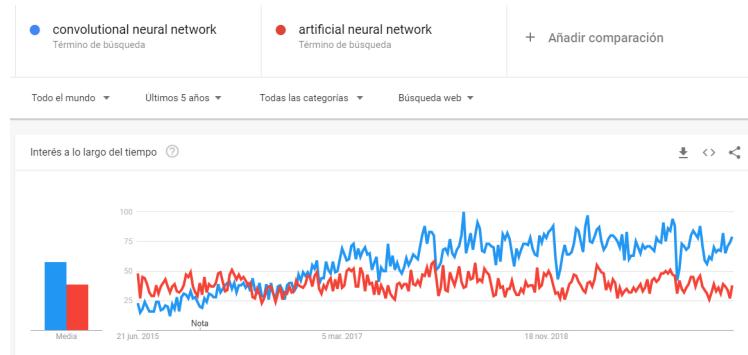


Imagen 1.2: búsqueda de términos de redes neuronales en google trend

Las **redes convolucionales** son actualmente utilizadas para diferentes propósitos: tratamiento de imágenes(visión por computador, extracción de características, segmentación, etc.), generación y clasificación de texto(o audio), predicción de series temporales, etc. En este capítulo veremos su aplicación en clasificación de imágenes y de texto.

1.4.2 Clasificación de imágenes

En este modelo de redes convolucionales las neuronas se corresponden a campos receptivos similares a las neuronas en la corteza visual de un cerebro humano. Este tipo de redes se han mostrado muy efectivas para tareas de detección y categorización de objetos y en la clasificación y segmentación de imágenes. Por ejemplo, estas redes en la década de 1990 las aplicó AT & T para desarrollar un modelo para la lectura de cheques. También más tarde se desarrollaron muchos sistemas OCR basados en CNN. En esta arquitectura cada neurona de una capa no recibe conexiones entrantes de todas las neuronas de la capa anterior, sino sólo de algunas. Esta estrategia favorece que una neurona se especialice en una región del conjunto de números (píxeles) de la capa anterior, lo que disminuye notablemente el número de pesos y de operaciones a realizar. Lo más normal es que neuronas consecutivas de una capa intermedia se especialicen en regiones solapadas de la capa anterior.

Una forma intuitiva para entender cómo trabajan estas redes neuronales es ver cómo nos representamos y vemos las imágenes. Para reconocer una cara primero tenemos que tener una imagen interna de lo que es una cara. Y a una imagen de una cara la reconocemos porque tiene nariz, boca, orejas, ojos, etc. Pero en muchas ocasiones una oreja está tapada por el pelo, es decir, los elementos de una cara se pueden ocultar de alguna manera. Antes de clasificarla, tenemos que saber la proporción y disposición y también cómo se relacionan las partes entre sí.

Para saber si las partes de la cara se encuentran en una imagen tenemos que identificar previamente líneas, bordes, formas, texturas, relación de tamaño, etcétera. En una red convolucional, cada capa lo que va a ir aprendiendo son los diferentes niveles de abstracción de la imagen inicial. Para comprender mejor el concepto anterior hemos seleccionado esta imagen de Raschka y Mirjalili (2019) donde se observa como partes del perro se transforman en neuronas del mapa de características

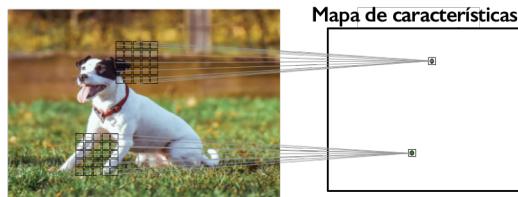


Imagen 1.3: Correspondencia de zonas de la imagen y mapa de características

Fuente: Raschka y Mirjalili (2019)

El objetivo de las redes CNN es aprender características de orden superior utilizando la operación de convolución.

Puesto que las redes neuronales convolucionales pueden aprender relaciones de entrada-salida (donde la entrada es una imagen en este caso), en la convolución, cada pixel de salida es una combinación lineal de los pixeles de entrada.

La **convolución** consiste en **filtrar** una imagen utilizando una **máscara**. Diferentes máscaras producen distintos resultados. Las máscaras representan las conexiones entre neuronas de capas anteriores. Estas capas aprenden progresivamente las características de orden superior de la entrada sin procesar.

Las redes neuronales convolucionales se forman usando dos tipos de capas: convolucionales y pooling. La capa de convolución transforma los datos de entrada a través de una operación matemática llamada convolución. Esta operación describe cómo fusionar dos conjuntos de información diferentes. A esta operación se le suele aplicar una función de transformación, generalmente la RELU. Después de la capa o capas de convolución se usa una capa de pooling, cuya función es resumir las respuestas de las salidas cercanas. Antes de obtener el output unimos la última capa de pooling con una red densamente conectada. Previamente se ha aplazado (Flattening) la última capa de pooling para obtener un vector de entrada a la red neural final que nos ofrecerá los resultados.

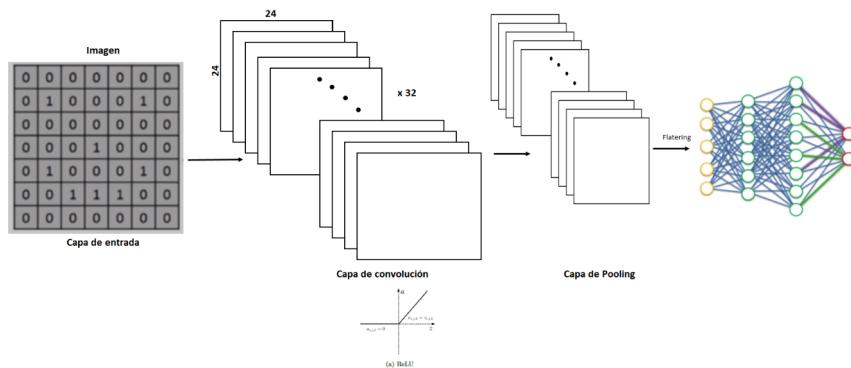


Imagen 1.4: Arquitectura de una CNN

Las redes neuronales convolucionales debido a su forma de concebirse son aptas para poder aprender a clasificar todo tipo de datos donde éstos estén distribuidos de una forma continua a lo largo del mapa de entrada, y a su vez sean estadísticamente similares en cualquier lugar del mapa de entrada. Por esta razón,

son especialmente eficaces para clasificar imágenes. También pueden ser aplicadas para la clasificación de series de tiempo o señales de audio.

En relación con el color y la forma de codificarse, en las redes convolucionales se realiza en tensores 3D, dos ejes para el ancho (width) y el alto (height) y el otro eje llamado de profundidad (depth) que es el canal del color con valor tres si trabajamos con imágenes de color RGB (Red, Green y Blue) rojo, verde y azul. Si disponemos de imágenes en escala de grises el valor de depth es uno. La base de datos MNIST (National Institute of Standards and Technology database) con la que trabajaremos en este epígrafe contiene imágenes de 28 x 28 píxeles, los valores de height y de width son ambos 28, y al ser una base de datos en blanco y negro el valor de depth es 1.

Las imágenes son matrices de píxeles que van de cero a 255 y que para la red neuronal se normalizan para que sus valores oscilen entre cero y uno.

1.4.2.1 Convolución

En las redes convolucionales todas las neuronas de la capa de entrada (los píxeles de las imágenes) no se conectan con todas las neuronas de la capa oculta del primer nivel como lo hacen las redes clásicas del tipo perceptrón multicapa o las redes que conocemos de forma genérica como redes densamente conectadas. Las conexiones se realizan por pequeñas zonas de la capa de entrada.

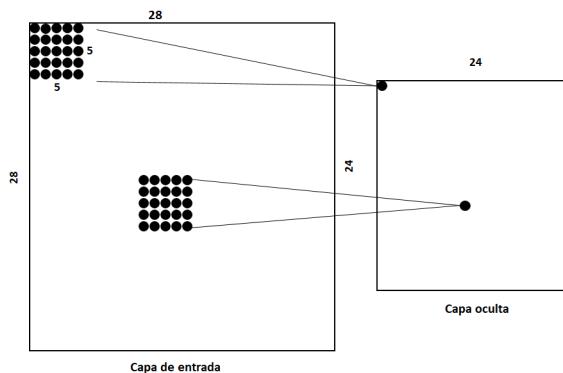


Imagen 1.5: Conexión de las neuronas de la capa de entrada con la capa oculta

Veamos un ejemplo para la base de datos de los dígitos del 1 a 9. Vamos a conectar cada neurona de la capa oculta con una región de 5 x 5 neurona, es decir, con 25 neuronas de la capa de entrada, que podemos denominarla ventana. Esta ventana va a ir recorriendo todo el espacio de entrada de 28 x 28 empezando por arriba y desplazándose de izquierda a derecha y de arriba abajo. Suponemos que los desplazamientos de la ventana son de un paso (un pixel) aunque este es un parámetro de la red que podemos modificar (en la programación lo llamaremos stride).

Para conectar la capa de entrada con la de salida utilizaremos una matriz de pesos (W) de tamaño 3 x 3 que recibe el nombre de filtro (filter) y el valor del sesgo. Para obtener el valor de cada neurona de la capa oculta realizaremos el producto escalar entre el filtro y la ventana de la capa de entrada. Utilizamos el mismo filtro para obtener todas las neuronas de la capa oculta, es decir en todos los productos escalares siempre utilizamos la misma matriz, el mismo filtro.

Se definen matemáticamente estos productos escalares a través de la siguiente expresión:

$$Y = X * W \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

Matriz X	Matriz W	Resultado
0 0 0 0 0	0,3 0,0 0,2	1,6 4,2 1,6
0 2 2 2 1	0,0 0,8 0,0	4,0 3,6 3,8
0 4 0 3 0	0,6 0,0 0,4	0,8 2,6 4,0
0 1 1 5 0		
0 0 0 0 0		

Como en este tipo de red un filtro sólo nos permite revelar una característica muy concreta de la imagen, lo que se propone es usar varios filtros simultáneamente, uno para cada característica que queramos detectar. Una forma visual de representarlo (si suponemos que queremos aplicar 32 filtros) es como se muestra a continuación:

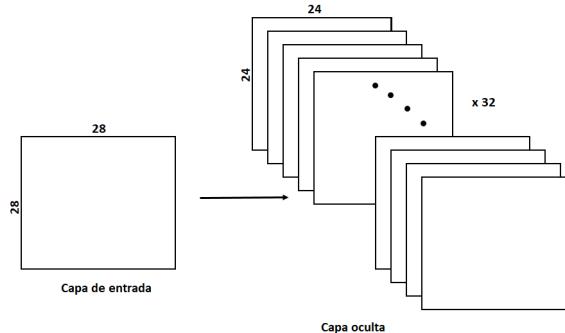


Imagen 1.6: Primera capa de la red convolucional con 32 filtros

Al resultado de la aplicación de los diferentes filtros se les suele aplicar la función de activación denominada RELU y que ya se comentó en la introducción.

Una interesante fuente de información es la documentación del software gratuito GIMP donde expone diferentes efectos que se producen en las imágenes al aplicar diversas convoluciones.

Un ejemplo claro y didáctico lo podemos obtener de la documentación del software libre de dibujo y tratamiento de imágenes denominado GIMP (<https://docs.gimp.org/2.6/es/plug-in-convmatrix.html>). Algunos de estos efectos nos ayudan a entender la operación de los filtros en las redes convolucionales y cómo afectan a las imágenes, en concreto, el ejemplo que presenta lo realiza sobre la figura del Taj Mahal.

El filtro enfocar lo que consigue es afinar los rasgos, los contornos lo que nos permite agudizar los objetos de la imagen. Toma el valor central de la matriz de cinco por cinco lo multiplica por cinco y le resta el valor de los cuatro vecinos. Al final hace una media, lo que mejora la resolución del pixel central porque elimina el ruido o perturbaciones que tiene de sus píxeles vecinos.

El filtro enfocar (Sharpen)

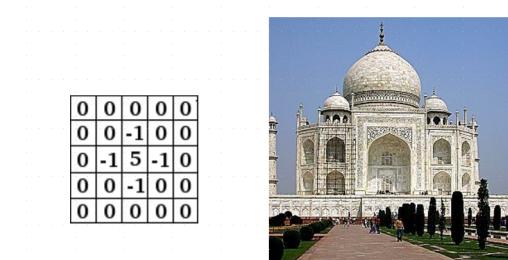


Imagen 1.7: Filtro Enfocar

Lo contrario al filtro enfocar lo obtenemos a través de la matriz siguiente, difuminando la imagen al ser estos píxeles mezclados o combinados con los píxeles cercanos. Promedia todos los píxeles vecinos a un pixel dado lo que implica que se obtienen bordes borrosos.

Filtro desenfocar

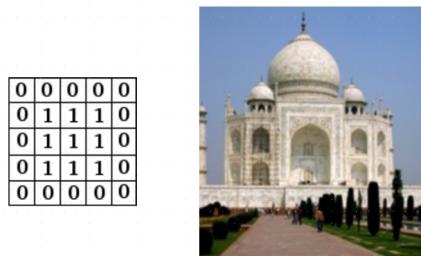


Imagen 1.8: Filtro DesEnfocar

Filtro Detectar bordes (Edge Detect)

Este efecto se consigue mejorando los límites o las aristas de la imagen. En cada píxel se elimina su vecino inmediatamente anterior en horizontal y en vertical. Se eliminan las similitudes vecinas y quedan los bordes resaltados. Al pixel central se le suman los cuatro píxeles vecinos y lo que queda al final es una medida de cómo de diferente es un píxel frente a sus vecinos. En el ejemplo, al hacer esto da un valor de cero de ahí que se observen tantas zonas oscuras.

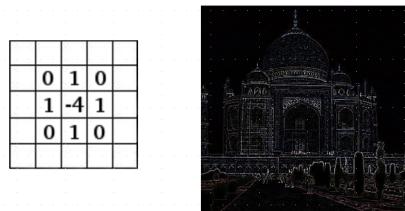


Imagen 1.9: Filtro Detectar Bordes

Filtro Repujado (Emboss)

En este filtro se observa que la matriz es simétrica y lo que intenta a través del diseño del filtro es mejorar los píxeles centrales y de derecha abajo restándole los anteriores. Se obtiene lo que en fotografía se conoce como un claro oscuro. Trata de mejorar las partes que tienen mayor relevancia.

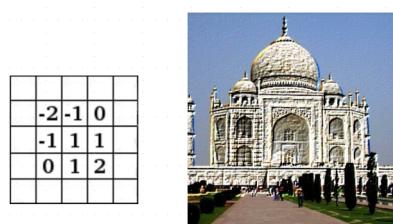


Imagen 1.10: Filtro Emboss

1.4.2.2 Pooling

Con la operación de **pooling** se trata de condensar la información de la capa convolucional. A este procedimiento también se le conoce como **submuestreo**.

Es simplemente una operación en la que reducimos los parámetros de la red. Se aplica normalmente a través de dos operaciones: **max-pooling** y **mean-pooling**, que también es conocido como average-pooling. Tal y como se observa en la imagen siguiente, desde la capa de convolución se genera una nueva capa aplicando la operación a todas las agrupaciones, donde previamente hemos elegido el tamaño de la región; en la figura siguiente es de tamaño 2, con lo que pasamos de un espacio de 24 x 24 neuronas a la mitad, 12 x 12 en la capa de pooling.

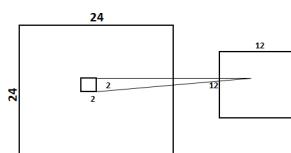


Imagen 1.11: Etapa de pooling de tamaño 2 x 2

Vamos a estudiar el pooling suponiendo que tenemos una imagen de 5 x 5 píxeles y que queremos efectuar una agrupación max-pooling. Es la más utilizada, ya que obtiene buenos resultados. Observamos los valores de la matriz y se escoge el valor máximo de los cuatro bloques de matrices de dos por dos. Max Pooling

En la agrupación Average Pooling la operación que se realiza es sustituir los valores de cada grupo de entrada por su valor medio. Esta transformación es menos utilizada que el max-pooling.

La transformación max-pooling presenta un tipo de invarianza local: pequeños cambios en una región local no varían el resultado final realizado con el max – pooling: se mantiene la relación espacial. Para ilustrar este concepto hemos escogido la imagen que presenta Torres (2020) donde se ilustra como partiendo de una matriz de 12 x 12 que representa al número 7, al aplicar la operación de max-pooling con una ventana de 2 x 2 se conserva la relación espacial.

2	0	1	5
3	1	0	3
4	7	5	2
1	1	0	3

Max Pooling de 2 x 2

3	5
7	5

Imagen 1.12: Max Pooling

2	0	1	5
3	1	0	3
4	7	5	2
1	1	0	3

Average Pooling de 2 x 2

1,5	2,25
3,25	2,5

Imagen 1.13: Average Pooling

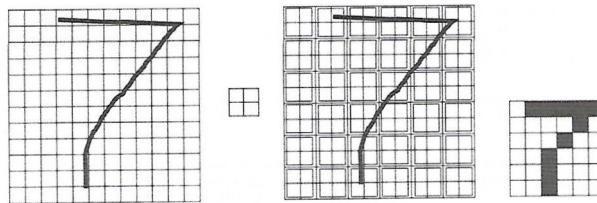


Imagen 1.14: Mantenimiento del pooling con la transformación

1.4.2.3 Padding

Para explicar el concepto del **Padding** vamos a suponer que tenemos una imagen de 5×5 píxeles, es decir 25 neuronas en la capa de entrada, y que elegimos, para realizar la convolución, una ventana de 3×3 . El número de neuronas de la capa oculta resultará ser de nueve. Enumeramos los píxeles de la imagen de forma natural del 1 al 25 para que resulte más sencillo de entender.

Resultado del recorrido de la ventana de 3×3																															
Imagen					1 2 3			2 3 4			3 4 5			6 7 8			7 8 9			8 9 10			11 12 13			12 13 14			13 14 15		
1	2	3	4	5	1	2	3	2	3	4	3	4	5	6	7	8	7	8	9	8	9	10	11	12	13	12	13	14	13	14	15
6	7	8	9	10	6	7	8	7	8	9	8	9	10	11	12	13	12	13	14	13	14	15	16	17	18	17	18	19	16	17	18
11	12	13	14	15	11	12	13	12	13	14	13	14	15	16	17	18	17	18	19	18	19	20	21	22	23	22	23	24	21	22	23
16	17	18	19	20	16	17	18	17	18	19	18	19	20	21	22	23	22	23	24	23	24	25	26	27	26	27	28	25	26	27	
21	22	23	24	25	21	22	23	22	23	24	23	24	25	26	27	28	27	28	29	28	29	30	31	32	31	32	33	30	31	32	

Imagen 1.15: Operación de convolución con una ventana de 3×3

Pero si queremos obtener un tensor de salida que tenga las mismas dimensiones que la entrada podemos llenar la matriz de ceros antes de deslizar la ventana por ella. Vemos la figura siguiente donde ya se ha llenado de valores cero y obtenemos, después de deslizar la ventana de 3×3 de izquierda a derecha y de arriba abajo, las veinticinco matrices de la figura nº 71

0	0	0	0	0	0	0	0
0	1	2	3	4	5	0	0
0	6	7	8	9	10	0	0
0	11	12	13	14	15	0	0
0	16	17	18	19	20	0	0
0	21	22	23	24	25	0	0
0	0	0	0	0	0	0	0

Imagen 1.16: Imagen con relleno de ceros

1.4.2.4 Stride

Hasta ahora, la forma de recorrer la matriz a través de la ventana se realiza desplazándola de **un solo paso**, pero podemos cambiar este hiperparámetro conocido como **stride**. Al aumentar el paso se decrementa la información que pasará a la capa posterior. A continuación, se muestra el resultado de las cuatro matrices que obtenemos con un stride de valor 3.

Finalmente, para resumir, una **red convolucional** contiene los siguientes elementos:

- **Entrada:** Son el número de píxeles de la imagen. Serán alto, ancho y profundidad. Tenemos un solo color (escala de grises) o tres: rojo, verde y azul.
- **Capa de convolución:** procesará la salida de neuronas que están conectadas en «regiones locales» de entrada (es decir píxeles cercanos), calculando el producto escalar entre sus pesos (valor de pixel) y una pequeña región a la que están conectados. En este epígrafe se presentan las imágenes con 32 filtros, pero puede realizarse con la cantidad que deseemos.

0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
0 1 2	1 2 3	2 3 4	3 4 5	4 5 0
0 6 7	6 7 8	7 8 9	8 9 10	9 10 0
0 6 7	6 7 8	7 8 9	8 9 10	9 10 0
0 11 12	11 12 13	12 13 14	13 14 15	14 15 0
0 6 7	6 7 8	7 8 9	8 9 10	9 10 0
0 11 12	11 12 13	12 13 14	13 14 15	14 15 0
0 16 17	16 17 18	17 18 19	18 19 20	19 20 0
0 21 22	21 22 23	22 23 24	23 24 25	24 25 0
0 16 17	16 17 18	17 18 19	18 19 20	19 20 0
0 21 22	21 22 23	22 23 24	23 24 25	24 25 0
0 0 0	0 0 0	0 0 0	0 0 0	0 0 0

Imagen 1.17: Operación de convolución con ventana 3 x 3 y padding

Resultado del recorrido de la ventana de 3 x 3 con un stride de 2				
<i>Imagen</i>			1 2 3	3 4 5
1	2	3	6	7 8
6	7	8	8	9 10
11	12	13	13	14 15
16	17	18	17	18 19
21	22	23	22	23 24
21	22	23	23	24 25
21	22	23	24	25

Imagen 1.18: Operación de convolución con una ventana de 3 x 3 y stride 2

- **Capa RELU** Se aplicará la función de activación en los elementos de la matriz.
- **Pooling (agrupar) o Submuestreo:** Se procede normalmente a una reducción en las dimensiones alto y ancho, pero se mantiene la profundidad.
- **Capa tradicional.** Se finalizará con la red de neuronas feedforward (Perceptrón multicapa que se denomina normalmente como red densamente conectada) que vinculará con la última capa de subsampling y finalizará con la cantidad de neuronas que queremos clasificar. En el gráfico siguiente se muestran todas las fases de una red neuronal convolucional.

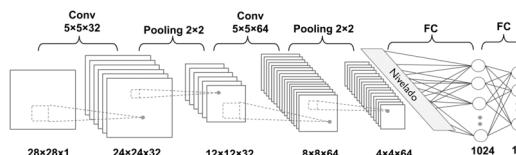


Imagen 1.19: Operación de convolución completa

1.4.2.5 Redes convolucionales con nombre propio

Existen en la actualidad muchas arquitecturas de redes neuronales convolucionales que ya están preparadas, probadas, disponibles e incorporadas en el software de muchos programas como Keras y Tensorflow.

Vamos a comentar algunos de estos modelos, bien por ser los primeros, o por sus excelentes resultados en concursos como el ILSVRC (Large Scale Visual Recognition Challenge).

Estas estructuras merecen atención dado que son excelentes para estudiarlas e incorporarlas por su notable éxito. El ILSVRC fue un concurso celebrado de 2011 a 2016 de donde nacieron las principales aportaciones efectuadas en las redes convolucionales. Este concurso fue diseñado para estimular la innovación en el campo de la visión computacional. Actualmente se desarrollan este tipo de concursos a través de la plataforma web: <https://www.kaggle.com/> Para ver más prototipos de redes convolucionales y los últimos avances y consejos sobre las redes convolucionales se puede consultar el siguiente artículo “Recent Advances in Convolutional Neural Networks” de Jiuxiang. G. et al. (2019)

Los cinco modelos más destacados hasta el año 2017 son los siguientes: LeNet-5, Alexnet, GoogLeNet, VGG y Restnet.

- **LeNet-5.** Este modelo de Yann LeCun de los años 90 consiguió excelentes resultados en la lectura de códigos postales consta de imágenes de entrada de 32 x 32 píxeles seguida de dos etapas de convolución – pooling, una capa densamente conectada y una capa softmax final que nos permite conocer los números o las imágenes.
- **AlexNet.** Fue la arquitectura estrella a partir del año 2010 en el ILSVRC y popularizada en el documento de 2012 de Alex Krizhevsky, et al. titulado "Clasificación de ImageNet con redes neuronales convolucionales profundas". Podemos resumir los aspectos clave de la arquitectura relevantes en los modelos modernos de la siguiente manera:
 - Empleo de la función de activación ReLU después de capas convolucionales y softmax para la capa de salida.
 - Uso de la agrupación máxima en lugar de la agrupación media.
 - Utilización de la regularización de Dropout entre las capas totalmente conectadas.
 - Patrón de capa convolucional alimentada directamente a otra capa convolucional.
 - Uso del aumento de datos (Data Aumentation,)
- **VGG.** Este prototipo fue desarrollado por un grupo de investigación de Geometría Visual en Oxford. Obtuvo el segundo puesto en la competición del año 2014 del ILSVRC. Las aportaciones principales de la investigación se pueden encontrar en el documento titulado " Redes convolucionales muy profundas para el reconocimiento de imágenes a gran escala " desarrollado por Karen Simonyan y Andrew Zisserman. Este modelo contribuyó a demostrar que la profundidad de la red es una componente crítica para alcanzar unos buenos resultados. Otra diferencia importante con los modelos anteriores y que actualmente es muy utilizada es el uso de un gran número de filtros y de tamaño reducido. Estas redes emplean ejemplos de dos, tres e incluso cuatro capas convolucionales apiladas antes de usar una capa de agrupación máxima. En esta arquitectura el número de filtros aumenta con la profundidad del modelo. El modelo comienza con 64 y aumenta a través de los filtros de 128, 256 y 512 al final de la parte de extracción de características del modelo. Los investigadores evaluaron varias variantes de la arquitectura si bien en los programas sólo se hace referencia a dos de ellas que son las que aportan un mayor rendimiento y que son nombradas por las capas que tienen: VGG-16 y VGG-19.
- **GoogLeNet.** GoogLeNet fue desarrollado por investigadores de Google Research. de Google, que con su módulo denominado de inception reduce drásticamente los parámetros de la red (10 veces menos que AlexNet) y de ella han derivado varias versiones como la Inception-v4. Esta arquitectura ganó la competición en el año 2014 y su éxito se debió a que la red era mucho más profunda (muchas más capas) y como ya se ha indicado introdujeron en el modelo las subredes llamadas inception. Las aportaciones principales en el uso de capas convolucionales fueron propuestos en el documento de 2015 por Christian Szegedy, et al. titulado " Profundizando con las convoluciones ". Estos autores introducen una arquitectura llamada "inicio" y un modelo específico denominado GoogLenet. El módulo inicio es un bloque de capas convolucionales paralelas con filtros de diferentes tamaños y una capa de agrupación máxima de 3×3 , cuyos resultados se concatenan. Otra decisión de diseño fundamental en el modelo inicial fue la conexión de la salida en diferentes puntos del modelo que lograron realizar con la creación de pequeñas redes de salida desde la red principal y que fueron entrenadas para hacer una predicción. La intención era proporcionar una señal de error adicional de la tarea de clasificación en diferentes puntos del modelo profundo para abordar el problema de los gradientes de fuga.
- **Red Residual o ResNet.** Esta arquitectura gano la competición de 2015 y fue creada por el grupo de investigación de Microsoft. Se puede ampliar la información en He, et al. en su documento de 2016 titulado " Aprendizaje profundo residual para el reconocimiento de la imagen ". Esta red es extremadamente profunda con 152 capas, confirmando al pasar los años que las redes son cada vez más profundas, más capas, pero con menos parámetros que estimar. La cuestión clave del diseño de esta red es la incorporación de la idea de bloques residuales que hacen uso de conexiones directa. Un

bloque residual, según los autores, “es un patrón de dos capas convolucionales con activación ReLU donde la salida del bloque se combina con la entrada al bloque, por ejemplo, la conexión de acceso directo” Otra clave, en este caso para el entrenamiento de la red tan profunda es lo que llamaron skip connections que implica que la señal con la que se alimenta una capa también se agregue a una capa que se encuentre más adelante. Resumiendo, las tres principales aportaciones de este modelo son:

- Empleo de conexiones de acceso directo.
- Desarrollo y repetición de los bloques residuales.
- Modelos muy profundos (152 capas) Aunque se encuentran otros modelos que también son muy populares con 34, 50 y 101 capas.

Una buena parte de los modelos comentados se incluyen en la librería de Keras y se pueden encontrar en la siguiente dirección de internet: <https://keras.io/api/applications/> Según los autores del programa Keras: “Las aplicaciones Keras son modelos de aprendizaje profundo que están disponibles junto con pesos preentrenados. Estos modelos se pueden usar para predicción, extracción de características y ajustes. Los pesos se descargan automáticamente cuando se crea una instancia de un modelo. Se almacenan en ~ / .keras / models /. Tras la creación de instancias, los modelos se construirán de acuerdo con el formato de datos de imagen establecido en su archivo de configuración de Keras en ~ / .keras / keras.json. Por ejemplo, si ha configurado image_data_format = channel_last, cualquier modelo cargado desde este repositorio se construirá de acuerdo con la convención de formato de datos TensorFlow, “Altura-Ancho-Profundidad”.

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	95.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	45.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6

Imagen 1.20: Modelos preentrenados en Keras

1.4.3 Clasificación de textos

Las redes convolucionales son actualmente utilizadas para diferentes propósitos: tratamiento de imágenes (visión por computador, extracción de características, segmentación, etc.), generación y clasificación de texto (o audio), predicción de series temporales, etc. En este caso, veremos en detalle un ejemplo de clasificación de texto.

Se presenta a continuación una aplicación práctica de clasificación de texto multiclas a partir de redes Convolucionales de una dimensión. Para ello, se utiliza una bbdd referida a las reclamaciones de los usuarios ante una entidad bancaria en función del tipo de producto.

En primer lugar, se importan las librerías a utilizar y se le el fichero:

```

1 import tensorflow as tf
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 import re
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import accuracy_score, confusion_matrix
9 import warnings
10 warnings.filterwarnings('ignore')

```

El fichero de trabajo contiene una serie de reclamaciones que no vienen acompañadas con su texto asociado. Se considera que lo más adecuado es excluir tales instancias del dataset de partida.

```

1 datos = pd.read_csv('C:/DEEP LEARNING/consumer_complaints.csv')
2 datos = datos[['product', 'consumer_complaint_narrative']] # variables de
3 interés
4 datos =
5 datos.dropna(subset=['product', 'consumer_complaint_narrative']).reset_index
6 (drop=True) # registros con texto no informado son eliminados de la muestra
7 print('Tamaño de los datos:', datos.shape)
8 Tamaño de los datos: (66806, 2)
9 sns.countplot(y='product', data=datos, order =
10 datos['product'].value_counts().index)
11 plt.xlabel('Reclamaciones'), plt.ylabel('Producto')
12 plt.show()

```

Como puede verse, se parte de once tipos de productos diferentes; si bien, para varios de ellos el número de reclamaciones no es considerado significativo por el área legal de la entidad. Por ello, y en base a la similitud de los productos, se agrupan las cuatro categorías con un menor número reclamaciones en

- Prepaid card: se incluye en la categoría de “Credit card”
- Payday loan: se incluye en la categoría “Bank account or service”
- Money transfers y Other financial service: forman un grupo conjunto denominado “Money transfers and Other financial service”

```

1 # agrupaciones
2 datos['product'] = np.where(datos['product']=='Payday loan', 'Bank account
3 or service', datos['product']) # préstamos
4 datos['product'] = np.where(datos['product']=='Prepaid card', 'Credit
5 card', datos['product']) # créditos
6 tipo_producto = ['Money transfers','Other financial service'] # otros
7 servicios financieros
8 datos['product'] = np.where(datos['product'].isin(tipo_producto), 'Money
9 transfers and Other', datos['product'])
10 sns.countplot(y='product', data=datos, order =
11 datos['product'].value_counts().index)
12 plt.xlabel('Reclamaciones'), plt.ylabel('Producto')
13 plt.show()

```

De esta forma, el número de grupos ha sido distribuido de una forma más equitativa. A modo de ejemplo, se muestra una de las reclamaciones:

```

1 def plot_reclamaciones(df, elemento):
2     df = df.loc[elemento].to_list()
3     return df
4 print('Producto:', plot_reclamaciones(datos, 100)[0])
5 print('Reclamacion:', plot_reclamaciones(datos, 100)[1])

```

Como puede verse, la reclamación 101 del dataset está asociada a un préstamo al consumo. Sin embargo, lo verdaderamente interesante del texto de ejemplo es la necesidad de realizar un preprocessado a los textos puesto que algunos símbolos, caracteres o, incluso palabras, no son relevantes para que la red sea capaz de interpretar el contenido del mismo. Por tanto, se lleva a cabo lo siguiente:

- Conversión del texto a minúsculas
- Exclusión del texto el contenido cifrado (XXXX)
- Eliminación de caracteres extraños
- Para poder hacer este preprocessado de textos se hace uso del paquete re de Python.

```

1 def preprocessado(reclamacion):
2     reclamacion = reclamacion.lower() # texto en minúsculas
3     reclamacion = reclamacion.replace('x', '') # cambio X por espacio
4     reclamacion = re.compile('[/(){}\\[]\\|@,;]').sub('', reclamacion) # símbolos extraños
5     ↵ (1)
6     reclamacion = re.compile('[^0-9a-z #+_]').sub(' ', reclamacion) # símbolos extraños
7     ↵ (2)
8     return reclamacion
9 datos['consumer_complaint_narrative'] =
10    ↵ datos['consumer_complaint_narrative'].apply(preprocessado) # aplicación de la función

```

Se presenta de nuevo el ejemplo anterior para ver el resultado del procesamiento de textos realizado.

```

1 print('Producto:', plot_reclamaciones(datos, 100)[0])
2 print('Reclamacion:', plot_reclamaciones(datos, 100)[1])

```

```

1 seed=123
2 tf.random.set_seed(seed)
3 np.random.seed(seed)
4 X_texto = datos['consumer_complaint_narrative']
5 Y_label = pd.get_dummies(datos['product']).values # las categorías son
6 convertidas a variable dummy
7 X_train_text, X_test_text, Y_train, Y_test =
8 train_test_split(X_texto,Y_label, test_size = 0.2, random_state = seed)
9 print('Entrenamiento:', X_train_text.shape)
10 print('Test:', X_train_text.shape)

```

Antes de definir la arquitectura de la red, se lleva a cabo la conversión del texto a variables numéricas que es el input que puede leer la red. Para ello, se realiza:

- La vectorización del texto asociado a las reclamaciones.
- El truncamiento y relleno de las secuencias de entrada para igualar la longitud en la modelización.

```

1 MAX_NB_WORDS = 25000 # frecuencia de palabras
2 MAX_SEQUENCE_LENGTH = 200 # número de palabras en cada reclamacion
3 EMBEDDING_DIM = 150 # dimensión del embedding
4
5 tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=MAX_NB_WORDS,
6 filters='!"#$%&()*+,-./:;=>?@[\\]^_`{|}~', lower=True)
7 tokenizer.fit_on_texts(X_train_text.values)
8 word_index = tokenizer.word_index
9 print('Tokens:', len(word_index))
10
11 X_train = tokenizer.texts_to_sequences(X_train_text.values)
12 X_train = tf.keras.preprocessing.sequence.pad_sequences(X_train,
13 maxlen=MAX_SEQUENCE_LENGTH)
14 print('Datos de entrada:', X_train.shape)

```

Por último, se crea la red neuronal siguiendo el método funcional. La red tiene las siguientes capas:

- Entrada: de 200 neuronas pues corresponde con la longitud de las secuencias
- Embedding: de dimensión 200 y toma como input el número máximo de palabras (25.000)
- Convolucional: de 64 neuronas
- MaxPooling:
- Densa: de 32 neuronas y con función de activación “relu”
- Salida: capa densa con 8 neuronas (número de categorías del target) y función de activación “softmax”

```

1 # capa de entrada
2 inputs = tf.keras.Input(shape=(X_train.shape[1],))
3 embedding = tf.keras.layers.Embedding(input_dim=MAX_NB_WORDS,
4 output_dim=EMBEDDING_DIM)(inputs)
5 capa_conv = tf.keras.layers.Conv1D(filters=64,
6 kernel_size=3,
7 padding='valid',
8 activation='relu')(embedding)
9 max_pooling = tf.keras.layers.GlobalMaxPooling1D()(capa_conv)
10 capa_densa = tf.keras.layers.Dense(units=32,
11 activation='relu',
12 kernel_regularizer=tf.keras.regularizers.l2(0.01))(max_pooling)
13 out = tf.keras.layers.Dense(units=Y_train.shape[1],
14 activation='softmax')(capa_densa)
15 modelo = tf.keras.Model(inputs=inputs, outputs=out)

```

El summary nos muestra el número de parámetros por capa y el número de parámetros total. Puede verse que el alto número de parámetros viene, principalmente, por la capa de Embedding.

```
1 modelo.summary()
```

La métrica utilizada para evaluar el desempeño es el accuracy y, como es un problema de clasificación multiclase, como función de pérdida categorical_crossentropy. Por su parte, se emplea Adam para la utilización del algoritmo de propagación del error hacia atrás (parámetros por defecto).

```

1 modelo.compile(loss='categorical_crossentropy', optimizer='adam',
2 metrics=['accuracy'])

```

Para el proceso de entrenamiento de la red destacar:

- Un máximo de 10 épocas y actualización de los pesos cada 128 muestras
- Reserva del 20% del dataset para ser usado como validación
- Uso de parada temprana para recoger el mejor modelo posible en el proceso iterativo

```

1 epochs = 10
2 batch_size = 128
3 history = modelo.fit(X_train, Y_train,
4 epochs=epochs,
5 batch_size=batch_size,
6 validation_split=0.2,
7
8 callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss',

```

Una vez realizado el entrenamiento, se visualiza el proceso para conocer su convergencia

```

1 # construcción de un data.frame
2 df_train=pd.DataFrame(history.history)
3 df_train['epochs']=history.epoch
4 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
5 ax1.plot(df_train['epochs'], df_train['loss'], label='train_loss')
6 ax1.plot(df_train['epochs'], df_train['val_loss'], label='val_loss')
7 ax2.plot(df_train['epochs'], df_train['accuracy'], label='train_acc')
8 ax2.plot(df_train['epochs'], df_train['val_accuracy'], label='val_acc')
9 ax1.legend()
10 ax2.legend()
11 plt.show()

```

Finalmente, se estima la bondad de ajuste con la muestra de test. Para ello, como esta muestra hace referencia a la puesta en producción del modelo, es necesario crear las secuencias de este nuevo dataset en función de la tokenización del modelo.

```

1 X_test = tokenizer.texts_to_sequences(X_test_text)
2 X_test = tf.keras.preprocessing.sequence.pad_sequences(X_test,
3 maxlen=MAX_SEQUENCE_LENGTH)

```

Y ahora ya sí, se realizan las predicciones y se evaluar el performance del modelo creado en la muestra de test.

```

1 # uso de argmax para pasar de probabilidad a estimación final
2 Y_test_pred = np.argmax(modelo.predict(X_test), axis=1) # predicción de la
3 etiqueta
4 Y_test_label = np.argmax(Y_test, axis=1) # obtención de las etiquetas sin
5 dummy
6 print('accuracy - test:', np.round(accuracy_score(Y_test_label,
7 Y_test_pred),5))
8 sns.heatmap(confusion_matrix(Y_test_pred, Y_test_label), annot = True,
9 fmt='.'0f') # matriz de confusión
10 plt.title('Matriz de confusión - test')
11 plt.show()

```

1.5 Redes Recurrentes

1.6 Redes recurrentes: Elman, Jordan, LSTM y GRU

1.6.1 Forecasting en series de tiempo

1.6.2 Clasificación y generación de textos

1.7 Autoencoders

1.7.1 Bases del Autoencoder

Los Autoencoders (AE) son uno de los tipos de redes neuronales que caen dentro del ámbito del Deep Learning, en la que nos encontramos con un modelo de aprendizaje no supervisado. Ya se empezó a hablar de AE en la década de los 80 (Bourlard and Kamp 1988), aunque es en estos últimos años donde más se está trabajando con ellos. La arquitectura de un AE es una Red Neuronal Artificial (ANN por sus siglas en inglés) que se encuentra dividida en dos partes, encoder y decoder (Charte et al. 2018), (Goodfellow, Bengio, and Courville 2016). El encoder va a ser la parte de la ANN que va codificar o comprimir los datos de entrada, y el decoder será el encargado de regenerar de nuevo los datos en la salida. Esta estructura de codificación y decodificación le llevará a tener una estructura simétrica. El AE es entrenado para ser capaz de reconstruir los datos de entrada en la capa de salida de la ANN, implementando una serie de restricciones (la reducción de elementos en las capas ocultas del encoder) que van a evitar que simplemente se copie la entrada en la salida. Si recordamos la estructura de una ANN clásica o también llamada Red Neuronal Densamente Conectada (ya que cada neurona conecta con todas las de la siguiente capa) nos encontramos en que en esta arquitectura, generalmente, el número de neuronas por capa se va reduciendo hasta llegar a la capa de salida que debería ser normalmente un número (si estamos en un problema regresión), un dato binario (si es un problema de clasificación). Figura nº 86: Red Neuronal Clásica

Fuente: Elaboración propia Si pensamos en una estructura básica de AE en la que tenemos una capa de entrada, una capa oculta y una capa de salida, ésta sería su representación:

Figura nº 87: Autoencoder básico

Fuente: Elaboración propia Donde los valores de son los datos de entrada y los datos son la reconstrucción de los mismos después de pasar por la capa oculta que tiene sólo dos dimensiones. El objetivo del entrenamiento de un AE será que estos valores de sean lo más parecidos posibles a los . Según (Charte et al. 2018) los AE se pueden clasificar según el tipo de arquitectura de red en:

- Incompleto simple
- Incompleto profundo
- Extra dimensionado simple
- Extra dimensionado profundo

Figura nº 88: Tipos de Autoencoders por arquitectura

Fuente: Elaboración propia Cuando hablamos de Incompleto nos referimos a que tenemos una reducción de dimensiones que permite llegar a conseguir una “compresión” de los datos iniciales como técnica para que aprenda los patrones internos. En el caso de Extra dimensionado es cuando subimos de dimensión para conseguir que aprenda esos patrones. En este último caso sería necesario aplicar técnicas de regularización para evitar que haya un sobreajuste en el aprendizaje. Cuando hablamos de Simple estamos haciendo referencia a que hay una única capa oculta, y en el caso de Profundo es que contamos con más de una capa oculta. Normalmente se trabaja con las arquitecturas de tipo Incompleto profundo, sobre todo cuando se está trabajando con tipos de datos que son imágenes. Aunque también podríamos encontrar una combinación de Incompleto con Extra dimensionado profundo cuando trabajamos con tipos de datos que no son imágenes y así crecer en la primera o segunda capa oculta, para luego reducir. Esto nos permitiría por ejemplo adaptarnos a estructuras de AE en las que trabajemos con número de neuronas en una capa que sean potencia de 2, y poder construir arquitecturas dinámicas en función del tamaño de los datos, adaptándolos a un tamaño prefijado. A continuación, vemos un gráfico de una estructura mixta Extra dimensionado - Incompleto profundo. Figura nº 89: Autoencoder Mixto (Incompleto y Extra dimensionado)

Fuente: Elaboración propia Idea intuitiva del uso de Autoencoders Si un AE trata de reproducir los datos de entrada mediante un encoder y decoder, ¿que nos puede aportar si ya tenemos los datos de entrada? Ya hemos comentado que la red neuronal de un AE es simétrica y está formada por un encoder y un decoder, además cuando trabajamos con los AE que son incompletos, se está produciendo una reducción del tamaño de los datos en la fase de codificación y de nuevo una regeneración a partir de esos datos más pequeños al original. Ya tenemos uno de los conceptos más importantes de los AE que es la reducción de dimensiones de los datos de entrada. Estas nuevas variables que se generan una vez pasado el encoder se les suele llamar el espacio latente. Este concepto de reducción de dimensiones en el mundo de la minería de datos lo podemos asimilar rápidamente a técnicas como el Análisis de Componentes Principales (PCA), que nos permite trabajar con un número más reducido de dimensiones que las originales. Igualmente, esa reducción de los datos y la capacidad de poder reconstruir el original podemos asociarlo al concepto de compresión de datos, de forma que con el encoder podemos comprimir los datos y con el decoder los podemos descomprimir. En este caso habría que tener en cuenta que sería una técnica de compresión de datos con pérdida de información (JPG también es un formato de compresión con pérdida de compresión). Es decir, con los datos codificados y el AE (pesos de la red neuronal), seríamos capaces de volver a regenerar los datos originales. Otra de las ideas alrededor de los AE es que, si nosotros tenemos un conjunto de datos de la misma naturaleza y los entrenamos con nuestro AE, somos capaces de construir una red neuronal (pesos en la red neuronal) que es capaz de reproducir esos datos a través del AE. Que ocurre si nosotros metemos un dato que no era de la misma naturaleza que los que entrenaron el AE, lo que tendremos entonces es que al recrear los datos originales no va a ser posible que se parezca a los datos de entrada. De forma que el error que vamos a tener va a ser mucho mayor por no ser datos de la misma naturaleza. Esto nos puede llevar a construir un AE que permita detectar anomalías, es decir, que seamos capaces de detectar cuando un dato es una anomalía porque realmente el AE no consigue tener un error lo bastante pequeño. Según lo visto de forma intuitiva vamos a tener el encoder que será el encargado de codificar los datos de entrada y luego tendremos el decoder que será el encargado de realizar la decodificación y conseguir acercarnos al dato original . Es decir intentamos conseguir . Si suponemos un Simple Autoencoder en el que tenemos una única capa oculta, con una función de activación intermedia y una función de activación de salida y los parámetros y representan los parámetros de la red neuronal en cada capa, tendríamos la siguiente expresión: [190] [191]

Así tendremos que donde será la reconstrucción de Una vez tenemos la idea intuitiva de para qué nos puede ayudar un AE, recopilamos algunos de los principales usos sobre los que actualmente se está trabajando. Más adelante ,comentaremos algunos de ellos con más detalle. Principales Usos de los Autoencoders A continuación, veamos la explicación de cuales son algunos de los principales usos de los autoencoders: Reducción de dimensiones / Compresión de datos En la idea intuitiva de los AE ya hemos visto claro que se pueden usar para la reducción de dimensiones de los datos de entrada. Si estamos ante unos datos de entrada de tipo estructurado estamos en un caso de reducción de dimensiones clásico, en el que queremos disminuir el número de variables con las que trabajar. Muchas veces este tipo de trabajo se hace mediante el PCA (Análisis de Componente Principales, por sus siglas en inglés), sabiendo que lo que se realiza es una transformación lineal de los datos, ya que conseguimos unas nuevas variables que son una combinación lineal de las mismas. En el caso de los AE conseguimos mediante las funciones de activación no lineales (simgmoide, ReLu, tanh, etc) combinaciones no lineales de las variables originales para reducir las dimensiones. También existen versiones de PCA no lineales llamadas Kernel PCA que mediante las técnicas de kernel son capaces de construir relaciones no lineales. En esta línea estamos viendo que cuando el encoder ha actuado, tenemos unos nuevos datos más reducidos y que somos capaces de prácticamente volver a reproducir teniendo el decoder. Podríamos pensar en este tipo de técnica para simplemente comprimir información. Hay que tener en cuenta que este tipo de técnicas no se pueden aplicar a cualquier dato que queramos comprimir, ya que debemos haber entrenado al AE con unos datos de entrenamiento que ha sido capaz de obtener ciertos patrones de ellos, y por eso es capaz luego de reproducirlos. Búsqueda de imágenes Cuando pensamos en un buscador de imágenes nos podemos hacer a la idea que el buscar al igual que con el texto nos va a mostrar entradas que seán imágenes parecidas a la que estamos buscando. Si construimos un autoencoder, el encoder nos va a dar unas variables con información para poder recrear de nuevo la imagen. Lo que parece claro es que si hay muy poca distancia entre estas variables y otras la reconstrucción de la imagen será muy parecida. Así nosotros podemos entrenar el AE con nuestro conjunto de imágenes, una vez tenemos el AE pasamos el encoder a todas las imágenes y las tenemos todas en ese nuevo espacio de variables. Cuando queremos buscar una imagen, le pasamos el autoencoder, y ya buscamos las más cercanas a nuestra imagen en el espacio de variables generado por el encoder. Detección de Anomalías Cuando estamos ante un problema de clasificación y tenemos un conjunto de datos que está muy desbalanceado, es decir, tenemos una clase mayoritaria que es mucho más grande que la minoritaria (posiblemente del orden de más del 95%), muchas veces es complicado conseguir un conjunto de datos balanceado que sea realmente bueno para hacer las predicciones. Cuando estamos en estos entornos tan desbalanceados muchas veces se dice que estamos ante un sistema para detectar anomalías. Un AE nos puede ayudar a detectar estas anomalías de la siguiente forma:

- Tomamos todos los datos de entrenamiento de la clase mayoritaria (o normales) y construimos un AE para ser capaces de reproducirlos. Al ser todos estos datos de la misma naturaleza conseguiremos entrenar el AE con un error muy pequeño.
- Ahora tomamos los datos de la clase minoritaria (o a nomalías) y los pasamos a través del AE obteniendo unos errores de reconstrucción.
- Definimos el umbral de error que nos separará los datos normales de las anomalías, ya que el AE sólo está entrenado con los normales y conseguirá un error más alto con las anomalías al reconstruirlas.
- Cogemos los datos de test y los vamos pasando por el AE, si el error es menor del umbral, entonces será de la clase mayoritaria. Si el error es mayor que el umbral, entonces estaremos ante una anomalía.

Eliminación de ruido Otra de las formas de uso de los autoencoders en tratamiento de imágenes es para eliminar ruido de las mismas, es decir poder quitar manchas de las imágenes. La forma de hacer esto es la siguiente:

- Partimos de un conjunto de datos de entrenamiento (imágenes) a las que le metemos ruido, por ejemplo, modificando los valores de cada pixel usando una distribución normal, de forma que obtenemos unos datos de entrenamiento con ruido.
- Construimos el AE de forma que los datos de entrada son los que tienen ruido, pero los de salida vamos a forzar que sean los originales. De forma que intentamos que aprendan a reconstruirse como los que no tienen ruido.
- Una vez que tenemos el AE y le pasamos datos de test con ruido, seremos capaces de reconstruirlos sin el ruido.

Modelos generativos Cuando hablamos de modelos generativos, nos referimos a AE que son capaces de generar cosas nuevas a las que existían. De forma que mediante técnicas como los Variational Autoencoders, los Adversarial Autoencoders seremos capaces de generar nuevas imágenes que no teníamos

inicialmente. Es decir, podríamos pensar en poder tener un AE que sea capaz de reconstruir imágenes de caras, pero que además con toda la información aprendida fuera capaz de generar nuevas caras que realmente no existen. Diseño del modelo de AE Transformación de datos Cuando se trabaja con redes neuronales y en particular con AEs, necesitamos representar los valores de las variables de entrada en forma numérica. En una red neuronal todos los datos son siempre numéricos. Esto significa que todas aquellas variables que sean categóricas necesitamos convertirlas en numéricas. Además es muy conveniente normalizar los datos para poder trabajar con valores entre 0 y 1, que van a ayudar a que sea más fácil que se pueda converger a la solución. Como ya sabemos normalmente nos encontramos que en una red neuronal las variables de salida son:

- un número (regresión)
- una serie de números (regresión múltiple)
- un dato binario (clasificación binaria)
- un número que representa una categoría (clasificación múltiple)

En el caso de los AE puede que tengamos una gran parte de las veces valores de series de números, ya que necesitamos volver a representar los datos de entrada. Esto significa que tendremos que conseguir en la capa de salida esos datos numéricos que teníamos inicialmente, como si se tuviera una regresión múltiple. Arquitectura de red Como ya se ha comentado en las redes neuronales, algunos de los hiperparámetros más importantes en un AE son los relacionados con la arquitectura de la red neuronal. Para la construcción de un AE vamos a elegir una topología simétrica del encoder y el decoder. Durante el diseño del AE necesitaremos ir probando y adaptando todos estos hiperparámetros de la ANN para conseguir que sea lo más eficiente posible:

- Número de capas ocultas y neuronas en cada una
- Función de coste y pérdida
- Optimizador
- Función de activación en capas ocultas
- Función de activación en salida

Número de capas ocultas y neuronas en cada una La selección del número de capas ocultas y la cantidad de neuronas en cada una va a ser un procedimiento de prueba y error en el que se pueden probar muchas combinaciones. Es cierto que en el caso de trabajar con imágenes y CNN ya hay muchas arquitecturas definidas y probadas que consiguen muy buenos resultados. Por otro lado para tipos de datos estructurados será muy dependiente de esos datos, de forma que será necesario realizar diferentes pruebas para conseguir un buen resultado.

Función de coste y pérdida En este caso no hay ninguna recomendación especial para las funciones de costes/pérdida y dependerá al igual que en las redes neuronales de la naturaleza de los datos de salida con los que vamos a trabajar.

Optimizador Se recomienda usar el optimizador ADAM (Diederik P. Kingma 2017) que es el que mejores resultados ha dado en las pruebas según (Walia 2017), consiguiendo una convergencia más rápida que con el resto de optimizadores.

Función de activación en capas ocultas En un AE las funciones de activación en las capas ocultas van a conseguir establecer las restricciones no lineales al pasar de una capa a la siguiente, normalmente se evita usar la función de activación lineal en las capas intermedias ya que queremos conseguir transformaciones no lineales. Se recomienda usar la función de activación ReLu en las capas ocultas, ya que parece ser que es la que mejores resultados da en la convergencia de la solución y además menor coste computacional tiene a la hora de realizar los cálculos.

Función de activación en salida En la capa de salida tenemos que tener en cuenta cual es el tipo de datos final que queremos obtener, que en el caso de un AE es el mismo que el tipo de dato de entrada. Normalmente las funciones de activación que se usarán en la última capa serán:

- Lineal con multiples unidades, para regresión de varios datos numéricos
- Sigmoid para valores entre 0 y 1

Tipos de Autoencoders Una vez entendido el funcionamiento de los AE, veamos algunos de los AE que se pueden construir para diversas tareas.

- Simple
- Multicapa o Profundo
- Sparse
- Convolucional
- Denoising
- Variational

En la descripción de los tipos de AE vamos usar código en R y en python y el framework keras con el backend Tensorflow. Todo el código se proporciona aparte. Usaremos como dataset a MINIST, que contiene 60.000/10.000 (entrenamiento/validación) imágenes de los números del 0 al 9, escritos a mano. Cada imagen tiene un tamaño de $28 \times 28 = 784$ pixels, en escala de grises, con lo que para cada pixel tendremos un valor entre 0 y 255 para definir cuál es su intensidad de gris.

Autoencoder Simple Vamos a describir como construir un autoencoder Simple usando una red neuronal densamente conectada en lugar de usar una red neuronal convolucional, para que sea más sencillo comprender el ejemplo. Es decir, vamos a tratar los datos de entrada como si fueran unos datos numéricos que queremos reproducir y no vamos a utilizar ninguna de las técnicas asociadas a las redes convolucionales. Hay que recordar que las redes convolucionales permiten mediante un tratamiento de las imágenes (convolución, pooling, etc) conseguir mejores resultados que si lo

hiciéramos directamente con redes densamente conectadas. En este caso tendremos una capa de entrada con 784 neuronas (correspondientes a los pixels de cada imagen), una capa intermedia de 32 neuronas, y una capa de salida de nuevo de las 784 neuronas para poder volver a obtener de nuevo los datos originales. En nuestro ejemplo vamos a tener los siguientes elementos.

- 1 capa de entrada (784 datos), 1 capa oculta (32 datos) y una capa de salida (784 datos)
- La función de coste/pérdida va a ser la Entropía
- Usaremos el optimizador Adam
- Como función activación intermedia usaremos ReLu
- Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1)

Autoencoder Sparse Ya hemos comentado que una forma de conseguir que un autoencoder aprenda estructuras o correlaciones es la reducción del número de neuronas, pero parte de este trabajo también se puede conseguir mediante técnicas de sparsing (escasez). Este tipo de técnicas se usan normalmente en las ANN para evitar el sobreajuste de nuestro modelo, de forma que en cada actualización de los pesos de la red no se tienen en cuenta todas las neuronas de la capa. Es decir, vamos a conseguir que en las capas que decidamos no todas las neuronas van a estar activadas, de esta manera además de ayudar a evitar el sobreajuste, también conseguiremos crear esas correlaciones que ayudan a construir el autoencoder. Existen dos métodos básicos para generar el sparse que son:

- Regularización L1
- Regularización L2 Básicamente los dos métodos tratan de hacer que los pesos de las neuronas tengan valores muy pequeños consiguiendo una distribución de pesos más regular. Esto lo consiguen al añadir a la función de pérdida un coste asociado a tener pesos grandes en las neuronas. Este peso se puede construir o bien con la norma L1 (proporcional al valor absoluto) o con la norma L2 (proporcional al cuadrado de los coeficientes de los pesos).

Básicamente trabajaremos con un AE Simple, con una red densamente conectada, al que le aplicaremos la regularización en su capa oculta. En nuestro ejemplo vamos a tener los siguientes elementos.

- 1 capa de entrada, 1 capa oculta y una capa de salida - Las capas ocultas tendrán aplicada la Regularización L2
- La función de coste/pérdida va a ser la Entropía
- Usaremos el optimizador Adam
- Como función activación intermedia usaremos ReLu
- Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1)

Autoencoder Multicapa o profundo Vamos a pasar ahora a una versión del autoencoder donde habilitamos más capas ocultas y hacemos que el descenso del número de neuronas sea más gradual hasta llegar a nuestro valor deseado, para luego volver a reconstruirlo. En este caso seguimos con redes densamente conectadas y aplicamos varias capas intermedias reduciendo el número de neuronas en cada una hasta llegar a la capa donde acaba el encoder para volver a ir creciendo en las sucesivas capas hasta llegar a la de salida. En nuestro ejemplo vamos a tener los siguientes elementos.

- 1 capa de entrada (784 datos), 5 capas ocultas (32 datos intermedia) y una capa de salida (784 datos)
- La función de coste/pérdida va a ser la Entropía
- Usaremos el optimizador Adam
- Como función activación intermedia usaremos ReLu
- Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1)

Autoencoder Convolucional En nuestro ejemplo al estar trabajando con imágenes podemos pasar a trabajar con Redes Convolucionales (CNN) de forma que en lugar de usar las capas densamente conectadas que hemos usado hasta ahora, vamos a pasar a usar las capacidades de las redes convolucionales. Al trabajar con redes convolucionales necesitaremos trabajar con capas de convolución o pooling para llegar a la capa donde acaba el encoder para volver a ir creciendo aplicando operaciones de convolución y upsampling (contrario al pooling). En nuestro ejemplo vamos a tener los siguientes elementos.

- 1 capa de entrada, 5 capas ocultas y una capa de salida
- La función de coste/pérdida va a ser la Entropía
- Usaremos el optimizador Adam
- Como función activación intermedia usaremos ReLu
- Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1)

Autoencoder Denoising Vamos a usar ahora un autoencoder para hacer limpieza en imagen, es decir, conseguir a partir de una imagen que tiene ruido otra imagen sin ese ruido. Entrenaremos al autoencoder para que limpie “ruido” que hay en la imagen y lo reconstruya sin ello. El ruido lo vamos a generar mediante una distribución normal y modificaremos el valor de los pixels de las imágenes. Usaremos estas imágenes con ruido para que sea capaz de reconstruir la imagen original sin ruido con el AE. Para realizar este proceso lo que haremos será

- Crear nuevas imágenes con ruido
- Entrenar el autoencoder con estas nuevas imágenes
- Calcular el error de reconstrucción respecto a las imágenes originales

Al estar trabajando con imágenes vamos a partir del Autoencoder de Convolución para poder aplicar el denosing. En nuestro ejemplo vamos a tener los siguientes elementos.

- 1 capa de entrada, 5 capas ocultas y una capa de salida
- La función de coste/pérdida va a ser la Entropía
- Usaremos el optimizador Adam
- Como

función activación intermedia usaremos ReLu - Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Variational Los Variational Autoencoder son un tipo de modelo que se denomina generativo, ya que va a permitir construir nuevas imágenes que no existían a partir de otras imágenes con las que se ha entrenado a la red neuronal. En realidad, es un autoencoder que durante el entrenamiento se le regulariza para evitar un sobreajuste y asegurar que en el espacio latente (intermedio) tenga buenas propiedades que permitan un buen proceso generativo. El proceso de construir este tipo de AE es muy parecido a los que ya hemos visto, con una pequeña diferencia en el paso entre el proceso de encoder y el posterior decoder. Hasta ahora lo que teníamos era que lo que obteníamos del encoder se lo pasábamos directamente al decoder, en este caso, el resultado del encoder no va a ser realmente un dato, sino una distribución de datos. De forma que al decoder no se le pasa directamente lo que ha salido del encoder, si no otro elemento cogido de la distribución generada. En este caso el proceso sería:

- Se codifica la entrada con el encoder no como un dato concreto, sino como una distribución normal (media y desviación)
- Se toma una muestra de un punto del espacio latente a partir de la distribución
- Se decodifica el punto de muestra con el decoder
- Se calcula la función de pérdida con el error de reconstrucción y la parte de regularización
- Se usa el backpropagation a través de la red neuronal para ajustar los pesos Figura nº 90: Autoencoder Mixto (Incompleto y Extra dimensionado)

Fuente: <https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73> Una vez que tenemos entrenado nuestro autoencoder seremos capaces de construir nuevas imágenes partiendo de puntos que estén en la distribución del espacio latente, de forma que esas pequeñas variaciones van a dar lugar a imágenes finales diferentes.

1.7.2 Casos de uso

1.8 Arquitecturas preentrenadas

Las **arquitecturas pre-entrenadas** en Deep Learning son modelos de redes neuronales que han sido previamente entrenados en grandes conjuntos de datos para realizar tareas específicas como clasificación de imágenes, generación de texto o reconocimiento de voz. Durante el entrenamiento, estos modelos aprenden patrones complejos de los datos, lo que les permite realizar tareas relacionadas con alta precisión y generalización.

El funcionamiento de las arquitecturas pre-entrenadas se basa en el concepto de **transferencia de aprendizaje**. La transferencia de aprendizaje consiste en aprovechar el conocimiento aprendido en una tarea para aplicarlo a otra tarea diferente; es decir, se utiliza como base para realizar tareas específicas con mayor facilidad.

Las arquitecturas pre-entrenadas suelen ser diseñadas por investigadores y equipos de desarrollo en instituciones académicas, laboratorios de investigación y empresas de tecnología. Estos expertos desarrollan las arquitecturas, definen el proceso de entrenamiento y seleccionan los conjuntos de datos adecuados para cada tarea. Algunas de las organizaciones e instituciones que lideran el diseño de arquitecturas pre-entrenadas son:

- *Google AI*: ha desarrollado arquitecturas pre-entrenadas como BERT y Vision Transformers, las cuales han tenido un gran impacto en el procesamiento del lenguaje natural y la visión artificial, respectivamente. Además, en los últimos años, han publicado Gema y Gemini, que permiten el aprendizaje automático en general, ofreciendo flexibilidad, escalabilidad y alto rendimiento en diversas tareas

- *Facebook AI Research*: ha contribuido con arquitecturas pre-entrenadas como FAIRSEQ y Detectron2, que han impulsado el rendimiento en tareas de procesamiento del lenguaje natural y detección de objetos, respectivamente. Además, también ha desarrollado Llama, Llama2 y Llama3, modelos que han destacado por su versatilidad y precisión en aplicaciones de reconocimiento de voz y procesamiento de texto.
- *OpenAI*: conocido por sus arquitecturas pre-entrenadas líderes como GPT-3 y Whisper, ha lanzado recientemente GPT-4. Este modelo promete mejorar la comprensión contextual y la generación de texto, lo que podría tener un gran impacto en aplicaciones de procesamiento del lenguaje natural. GPT-4 representa un avance significativo en la investigación de inteligencia artificial y abre nuevas posibilidades para sistemas más avanzados.

Estos modelos son entrenados en grandes clústeres de servidores con hardware especializado, como GPUs y TPUs, utilizando conjuntos de datos masivos y técnicas de optimización avanzadas. A día de hoy se han convertido en un elemento capital para los desarrolladores de aprendizaje automático puesto que nos permite aprovechar modelos entrenados previamente que, en condiciones normales, sería casi imposible realizar por centros no especializados. Así, podríamos destacar las siguientes ventajas:

- Ahorro de tiempo y recursos: al aprovechar el conocimiento pre-entrenado, nos permiten a los desarrolladores ahorrar tiempo y recursos computacionales en comparación con entrenar un modelo desde cero
- Mayor rendimiento: suelen ofrecer un rendimiento superior a los modelos entrenados desde cero, especialmente para tareas complejas
- Facilidad de uso: las librerías y entornos facilitan el acceso y la utilización de arquitecturas pre-entrenadas, incluso para usuarios con poca experiencia en Deep Learning

Como se ha dicho, cada vez más se emplean este tipo de arquitecturas para realizar tareas que, con otro tipo de diseños de aprendizaje automático serían más complejas y costosas de abordar. En este aspecto, destacar que a la hora de elegir qué tipo de arquitectura pre-entrenada es adecuada para un caso de uso es necesario tener en cuenta la tarea específica a abordar, el conjunto de datos con los que ha sido entrenada y la similaridad con los nuestros propios así como si disponemos de ciertas limitaciones en cuanto a recursos computacionales.

Por último, destacar que existen diversas librerías y entornos que facilitan el trabajo con arquitecturas pre-entrenadas. Algunas de las opciones más interesantes son:

- TensorFlow Hub: es un repositorio de módulos pre-entrenados para TensorFlow, que incluye una amplia gama de arquitecturas pre-entrenadas para diversas tareas.
- Hugging Face Hub: es una plataforma similar a TensorFlow Hub, pero que ofrece soporte para múltiples frameworks de Deep Learning, incluyendo TensorFlow, PyTorch y JAX.

1.8.1 Paquetes específicos en Python

1.8.1.1 Transformers

Transformers es una biblioteca de código abierto en Python desarrollada por Hugging Face que proporciona un conjunto de herramientas para trabajar con modelos de lenguaje basados en redes neuronales transformadoras.

Esta librería se caracteriza por su gran variedad de modelos pre-entrenados y capacidad de hacer fine-tuning de modelos. Así, entre sus principales ventajas podemos citar:

- La facilidad de uso
- La flexibilidad y la escalabilidad
- La comunidad activa de desarrolladores

Finalmente, indicamos proporcionamos una serie de recursos adicionales:

- Sitio web de Transformers: <https://huggingface.co/docs/transformers/en/index>
- Documentación de Transformers: <https://huggingface.co/docs>
- Tutoriales de Transformers: <https://www.youtube.com/watch?v=QEaBAZQCtwE>

1.8.2 Tensorflow-Hub

Tensorflow-Hub alberga una gran colección de módulos de aprendizaje automático pre-entrenados y reutilizables, creados por Google y la comunidad de TensorFlow.

Esta librería se caracteriza por disponer de una gran cantidad de modelos pre-entrenados para diferentes tareas relacionadas con el aprendizaje profundo. Así, entre sus principales ventajas podemos citar: - El acceso a modelos pre-entrenados de última generación - La flexibilidad y personalización - La facilidad de

Finalmente, indicamos proporcionamos una serie de recursos adicionales:

- Sitio web de TensorFlow Hub: <https://www.tensorflow.org/hub>
- Documentación de TensorFlow Hub: <https://www.tensorflow.org/hub>
- Tutoriales de TensorFlow Hub: <https://www.tensorflow.org/hub/tutorials>

1.8.3 Arquitecturas Zero-shot

Las **arquitecturas Zero-Shot**, también conocidas como *modelos de Aprendizaje por Analogía* son capaces de realizar tareas de clasificación sin necesidad de entrenamiento específico para cada categoría. En su lugar, estas arquitecturas aprenden representaciones vectoriales de conceptos a partir de datos no etiquetados, lo que les permite generalizar a nuevas categorías sin haberlas visto nunca antes.

1.8.3.1 Uso en imágenes

En el ámbito de la visión artificial, las arquitecturas Zero-Shot para imágenes han demostrado ser particularmente útiles para tareas como la clasificación de imágenes de escenas naturales, la identificación de animales y la detección de objetos. Entre los modelos más destacados en esta área se encuentran: - *ImageNet-pretrained CLIP*: basado en la arquitectura CLIP (Contrastive Language-Image Pre-training), este modelo utiliza representaciones de imágenes y texto para realizar clasificación Zero-Shot de imágenes con gran precisión - *ResNet-pretrained ZSL*: Este modelo combina la arquitectura ResNet, conocida por su rendimiento en tareas de clasificación de imágenes, con un enfoque Zero-Shot basado en la distancia entre representaciones

MATERIAL COMPLEMENTARIO - NOTEBOOK: ejemplo en Zero-shot: [claseificación animales](#)

1.8.3.2 Uso en texto

En el procesamiento del lenguaje natural, las arquitecturas Zero-Shot para texto han ganado popularidad para tareas como la clasificación de documentos, la extracción de información y la categorización de textos. Algunos modelos representativos en este campo son:

- *BERT-pretrained ZSL*: basado en la arquitectura BERT (Bidirectional Encoder Representations from Transformers), este modelo utiliza representaciones contextuales de palabras para realizar clasificación Zero-Shot de texto con gran precisión
- *Siamese Networks with Word Embeddings*: este enfoque utiliza redes siamesas, un tipo de red neuronal que aprende a comparar pares de entradas, para clasificar texto Zero-Shot utilizando representaciones de palabras pre-entrenadas.

1.8.4 Detección de objetos

La **detección de objetos** es una tarea fundamental en el ámbito de la visión artificial, que consiste en identificar y localizar objetos dentro de una imagen o video. Las arquitecturas pre-entrenadas para la detección de objetos han revolucionado este campo, ofreciendo modelos de alta precisión y eficiencia. Entre los modelos más utilizados se encuentran:

- *Faster R-CNN*: Este modelo combina la arquitectura de redes convolucionales profundas (CNN) con una región de propuesta de regiones (RPN) para detectar y localizar objetos con gran precisión
- *YOLOv5*: Este modelo destaca por su velocidad y eficiencia, utilizando una arquitectura basada en convoluciones y bloques de atención para detectar objetos en tiempo real

MATERIAL COMPLEMENTARIO - VIDEOTUTORIAL: ejemplo en [Arquitecturas Tensorflow-Hub](#)

1.8.5 Conversión de voz a texto

La **conversión de voz a texto**, también llamado como **Speech-to-Text** es una tarea crucial para la interacción hombre-máquina. Las arquitecturas pre-entrenadas para este tipo de casos han impulsado el desarrollo de sistemas de reconocimiento de voz de alta precisión, capaces de transcribir audio en texto con gran fluidez. Uno de los principales modelos para esta tarea es *Whisper* el cual fue desarrollado por OpenAI. Este modelo se basa en la arquitectura *Transformer* y utiliza aprendizaje supervisado y multitarea para lograr una precisión y robustez excepcionales en una amplia gama de condiciones acústicas.

MATERIAL COMPLEMENTARIO - VIDEOTUTORIAL: ejemplo en [Speech2Text](#)

1.9 Aprendizaje por Refuerzo

1.9.1 Introducción

Hasta ahora hemos visto como el Deep Learning se usa para el **aprendizaje supervisado** y el **aprendizaje no supervisado**, pero vamos a dar un paso más, en el que veremos como usar Deep Learning en otro tipo de aprendizaje llamado **aprendizaje por refuerzo**.

El **Aprendizaje por Refuerzo (RL)** por sus siglas en inglés, **Reinforcement Learning**) trata de conseguir que el sistema aprenda mediante recompensa/castigo, en función de si los pasos que da son buenos o malos. De esta manera, cuanta mayor recompensa se tenga es que nuestro sistema se ha acercado a la solución buena. Se trata de aprender mediante la interacción y la retroalimentación de lo que ocurra.

Partiremos de dos elementos clave **agente** (es el que aprende y toma decisiones), y el **entorno** (donde el agente aprende y decide que acciones tomar). Tendremos que el agente podrá realizar **acciones** que

normalmente provocarán un cambio de **estado** y a la vez se tendrá una **recompensa** (positiva o negativa) en función de la acción tomada en el entorno en ese momento.

Es decir, nos encontraremos un agente que realizará una acción a_t en el tiempo t , esta acción afectará al entorno que estará en un estado S_t y mediante esta acción cambiará a un estado S_{t+1} y además dará una recompensa r_{t+1} en función de los malo o bueno que haya sido este paso.

El agente volverá a examinar el nuevo estado del entorno S_{t+1} y la nueva recompensa recibida r_{t+1} y volverá a tomar la decisión de realizar una nueva acción a_{t+1} .

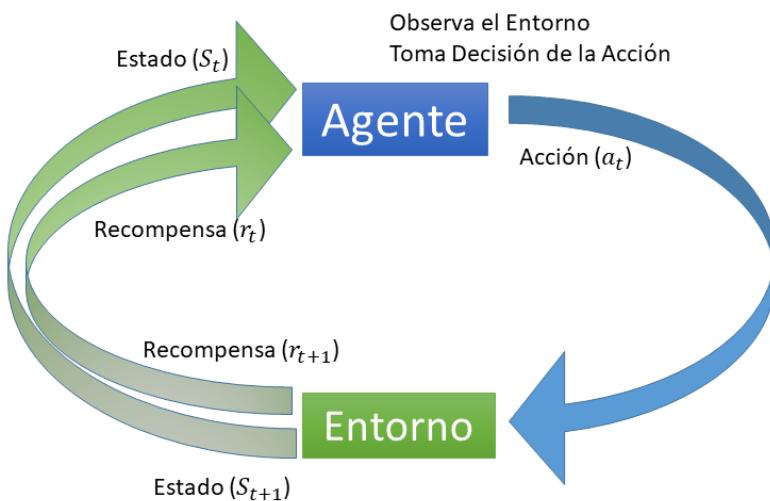


Imagen 1.21: Esquema Aprendizaje por Refuerzo - Fuente: Propia

1.9.2 Formalismo Matemático

El formalismo matemático para el Aprendizaje por Refuerzo está basado en los **Procesos de Decisión de Markov** (MDP por sus siglas en inglés). (CS229 Lecture notes).

1.9.2.1 Propiedad de Markov

Si tenemos una secuencia de estados s_1, s_2, \dots, s_t y tenemos la probabilidad de pasar a otro estado s_{t+1} , diremos que se cumple la **Propiedad de Markov** si el **futuro** es independiente del **pasado** y sólo se ve afectado por el **presente**, es decir:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_t, s_{t-1}, \dots, S_2, S_1]$$

Tendremos una **Matriz de Probabilidades de Transición** a una matriz con las probabilidades de todos los posibles cambios de estado que se puedan producir,

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s]$$

1.9.2.2 Proceso de Markov

Así llamaremos **Proceso de Markov** a un proceso aleatorio sin memoria, es decir, una secuencia de estados S_1, S_2, \dots con la propiedad de Markov.

Un Proceso de Markov está formado por una dupla $\langle \mathcal{S}, \mathcal{P} \rangle$:

- \mathcal{S} conjunto finito de Estados
- \mathcal{P} matriz de probabilidades de transición

1.9.2.3 Proceso de Recompensa de Markov

LLamaremos **Proceso de Recompensa de Markov (MRP)**, por sus siglas en inglés) a una cuádrupla $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, formada por:

- \mathcal{S} conjunto finito de Estados
- \mathcal{P} matriz de probabilidades de transición
- \mathcal{R} Función de recompensa definida como: $\mathcal{R}_s = E[R_{t+1} | S_t = s]$, donde R_{t+1} es la recompensa obtenida de pasar al estado S_{t+1} desde el estado S_t
- γ Factor de descuento, con $\gamma \in [0, 1]$

En este contexto llamaremos **Saldo (G_t)** a la suma de todas las recompensas conseguidas a partir del estado s_t con el factor de descuento aplicado.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

El hecho de usar γ (**factor descuento**), nos permite dar grandes recompensas lo antes posible, y no dar tanto valor a futuras recompensas lejanas. También puede haber otras interpretaciones por ejemplo a nivel económico, si la recompensa está basado en un dato monetario real, tendría sentido que el dinero a futuro tendría menos valor. También nos permite asegurar que este valor de G_t es finito ya que produce que la serie sea convergente.

Cuando los valores del factor descuento se acercan a **0** podríamos decir que nos fijamos sólo en los valores más cercanos de la recompensa. En cambio cuando los valores se acercan a **1** entonces les daremos más peso a los valores más lejanos de la recompensa.

Una vez definido el Saldo podemos definir la **Función Valor de Estado** como la función que nos da el **Saldo Esperado** comenzando por el estado s . Es decir:

$$V(s) = \mathbb{E}[G_t | S_t = s]$$

Esta función nos dice cómo de bueno es partir de este estado y continuar.

1.9.2.4 Proceso de Decisión de Markov

Un **Proceso de Decisión de Markov** (MDP por sus siglas en inglés) es un tupla $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ donde:

- \mathcal{S} es el conjunto de posibles **estados**.
- \mathcal{A} es el conjunto de posibles **acciones**.
- \mathcal{P} son las **probabilidades de transición** de un estado a otro en función de la acción realizada. Por cada estado y acción hay una distribución de probabilidad para pasar a otro estado.

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- γ es el conocido como **factor de descuento** y tendrá un valor entre $[0, 1]$ y nos proporciona cuánto descontamos en las recompensas a futuro.
- \mathcal{R} es la **Función de recompensa** definida como: $\mathcal{R}_s^a = E[R_{t+1} | S_t = s, A_t = a]$, donde R_{t+1} es la recompensa obtenida de pasar al estado S_{t+1} desde el estado

Además tenemos que este **proceso estocástico** cumple la **propiedad de Markov** que dice que el futuro es independiente del pasado dado el presente. En términos de nuestro problema, podría decir que pasar de un estado s_t al siguiente s_{t+1} sólo depende de s_t y no de los anteriores estados

$$\mathbb{P}(s_{t+1} | s_t) = \mathbb{P}(s_{t+1} | s_1, s_2, \dots, s_t)$$

Veamos cuál es la **dinámica** de un MDP:

- Empezamos con un estado $s_0 \in \mathcal{S}$
- Elegimos una acción $a_0 \in \mathcal{A}$ (la política será la que la elija)
- Obtenemos una recompensa $R_1 = R(s_0) = R(s_0, a_0)$
- Elegimos una acción $a_1 \in \mathcal{A}$ (la política será la que la elija)
- Se transiciona aleatoriamente a un estado s_1 en función del valor de $P_{s_0 s_1}^{a_1}$
- Obtenemos una recompensa $R_2 = R(s_1) = R(s_1, a_1)$
- Se transiciona a aleatoriamente a un estado s_2 en función del valor de $P_{s_1 s_2}^{a_2}$
- ...
- Repetimos de forma iterativa este proceso

La meta en RL es elegir las **acciones** adecuadas en el tiempo para **maximizar**:

$$\mathbb{E}[G_t] = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \dots]$$

Que es conocido como la **hipótesis de la recompensa**.

Vamos a introducir el término de **política** como una función $\pi : \mathcal{S} \rightarrow \mathcal{A}$ que mapea los estados a las acciones. Es decir, es la que decide qué **acción** hay que **ejecutar** en función de **cual** es el **estado** en el que estamos. Una política podría ser determinística o estocástica.

$$a = \pi(s)a = \pi(a|s) = \mathbb{P}[A = a | S = s]$$

Una política define cuál va a ser el comportamiento de un **agente**. En un MDP las políticas dependen del estado actual, y no de la historia de los estados pasados.

Diremos que estamos **ejecutando una política** π si cuando estamos en un estado s aplicamos la acción $a = \pi(s)$

Definiremos:

$$\mathcal{P}_{s,s}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{s,s}^a \mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a$$

También definiremos la **Función Valor de Estado** para una **política** π a la función que nos predice la recompensa a futuro (el **saldo esperado**):

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R(s_t) + \gamma R(s_{t+1}) + \gamma^2 R(s_{t+2}) + \gamma^3 R(s_{t+3}) + \dots | s_t = s]$$

Es decir, la esperanza de la suma de las recompensas con factor descuento suponiendo el comienzo en $s_t = s$ y tomando las acciones bajo la política π . Nos permite decir cómo de buenos o malos son los estados.

Añadiremos el concepto de la **Función Valor de Acción**, también llamada **Función de Calidad** (por eso se usa la Q (Quality), para una **política** π a la función que nos predice la recompensa a futuro (el saldo esperado), suponiendo que se parte de una acción a .

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi[R(s_t) + \gamma R(s_{t+1}) + \gamma^2 R(s_{t+2}) + \gamma^3 R(s_{t+3}) + \dots | s_t = s, A_t = a]$$

La función de **Valor de Estado** puede ser descompuesta en la **recompensa inmediata** y el resto de la recompensa:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s]$$

y del mismo modo se puede descomponer la función **Valor de Acción**:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

Luego tenemos

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s, a)$$

y

$$Q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^\pi(s')$$

Llegando a

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^\pi(s'))$$

y

$$Q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s) Q^\pi(s', a')$$

Dada una política π su **función valor de estado** asociada $V^\pi(s)$ cumple la **Ecuación de Bellman**:

$$V^\pi(s) = R_s + \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V^\pi(s')$$

Lo que nos dice que la **función valor** está separada en **dos términos**:

- La recompensa inmediata $R(s)$
- La suma de recompensas a futuro con el factor de descuento.

Igualmente su **función valor de acción asociada** $Q^\pi(s, a)$ cumple la **Ecuación de Bellman**:

$$Q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s', a)$$

Las **Ecuaciones de Bellman** permiten garantizar una **solución óptima** del problema de forma que dada una **política óptima** (π^*), además se cumple:

$$V^{\pi^*}(s) = V^*(s) = \max_\pi V^\pi(s) Q^{\pi^*}(s, a) = Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

Es decir, que las funciones de valor de estado y de acción óptimas son las mismas que se general con la **política óptima**.

Como la meta del RL es encontrar una **política óptima** π^* la cual maximize el valor del **saldo esperado total (desde el inicio)** $G_0 = \sum_{t=0}^{\infty}$, es decir, podríamos definir la política óptima como:

$$\pi^*(a|s) = \begin{cases} 1 & \text{si } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^*(s, a) \\ 0 & \text{si cualquier otro caso} \end{cases} \quad (1.1)$$

Luego si conocemos $Q^*(s, a)$ inmediatamente tenemos una **política óptima**.

1.9.2.5 Resolución de las Ecuaciones de Bellman

Las ecuaciones de Bellman pueden ser usadas para resolver de forma eficiente V^π , especialmente en un **MDP** de un número finito de estado, escribiendo una ecuación $V^\pi(s)$ por cada estado.

La mayoría de los algoritmos de RL usan las Ecuaciones de Bellman para resolver el problema. La forma básica de resolverlo es usando **programación dinámica** (PD por sus siglas en inglés), aunque nos encontramos con muchos problemas para resolverla cuando el número de acciones/estados aumenta. También se usan otras técnicas como los **métodos de montecarlo** (MMC, por sus siglas en inglés) o los métodos de **diferencia temporal** (TD, por sus siglas en inglés).

Pasemos a ver una clasificación de los tipos de algoritmos para resolver los problemas de RL.

1.9.3 Taxonomía de Algoritmos

Desde OpenAI (https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below) obtenemos la siguiente taxonomía de algoritmos de RL que nos servirá como guía para entender como clasificar los algoritmos:

La primera gran separación se hace sobre si los algoritmos siguen un modelo definido (model-based) o no (model-free).

Model-free

Por otro lado los **model-free** usan la experiencia para aprender o una o ambas de dos cantidades más simples (valores estado/acción o políticas).

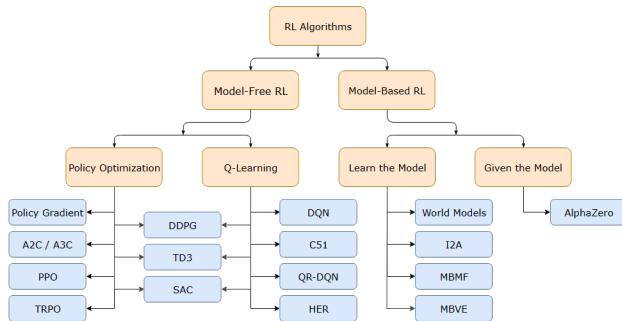


Imagen 1.22: Modelos Reinforcement Learning

Las aproximaciones de estos algoritmos son de tres tipos:

- **Policy Optimization**

El agente aprende directamente la función política que mapea el estado a una acción. Nos podemos encontrar con dos tipos de políticas, las **políticas determinísticas** (no hay incertidumbre en el mapeo) y las **políticas estocásticas** (tenemos una distribución de probabilidad en las acciones). En este último caso diremos que tenemos un Proceso de Decisión de Markov Parcialmente Observable (POMDP, por sus siglas en inglés).

- **Q-Learning**

En este caso el agente aprende una función valor de acción $Q(s, a)$ que nos dirá cómo de bueno es tomar una acción dependiendo del estado.

- **Híbridos**

Estos métodos combinan la fortaleza de los dos métodos anteriores, aprendiendo tanto la función política como la función valor de acción.

Model-based

Los algoritmos **model-based** usan la experiencia para construir un modelo interno de transiciones y resultados inmediatos en el entorno. Las acciones son elegidas mediante búsqueda o planificación en este modelo construido.

Las aproximaciones de estos algoritmos son de dos tipos:

- Aprender el Modelo

Para aprender el modelo se ejecuta una política base,

- Aprender dado el Modelo

Nos centraremos en los algoritmos de tipo **Model-Free** que son los más utilizados ya que no requieren del modelo. Si se quieren profundizar en los diferentes algoritmos, se puede consultar la documentación en: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#links-to-algorithms-in-taxonomy.

Vamos a ver 2 de los algoritmos de tipo **Model-free** que nos van a permitir el paso de un algoritmo sin **Deep Learning** y otro en el que se aplica **Deep Learning** para obtener el objetivo final de tener un **agente** capaz de aprender por sí solo a realizar las tareas específicas que se tengan que realizar.

1.9.4 Q-Learning (value)

Q-Learning es un método basado en valor y que usa el **sistema TD** (actualización su función valor en cada paso) para el entrenamiento y su función de valor de estado.

El nombre de **Q** viene de **Quality** (calidad), por que nos da la calidad de la acción en un determinado estado. Lo que tenemos es que vamos a tener una **función de valor de acción (Q-función)** que nos da un valor numérico de cómo de buena es a partir de un estado **s** y una acción **a**.

En este caso tenemos que internamente nuestra **Q-función ($Q(s, a)$)** es una **Q-tabla**, de forma que cada fila corresponde a un estado, y cada columna a una de las posibles acciones.

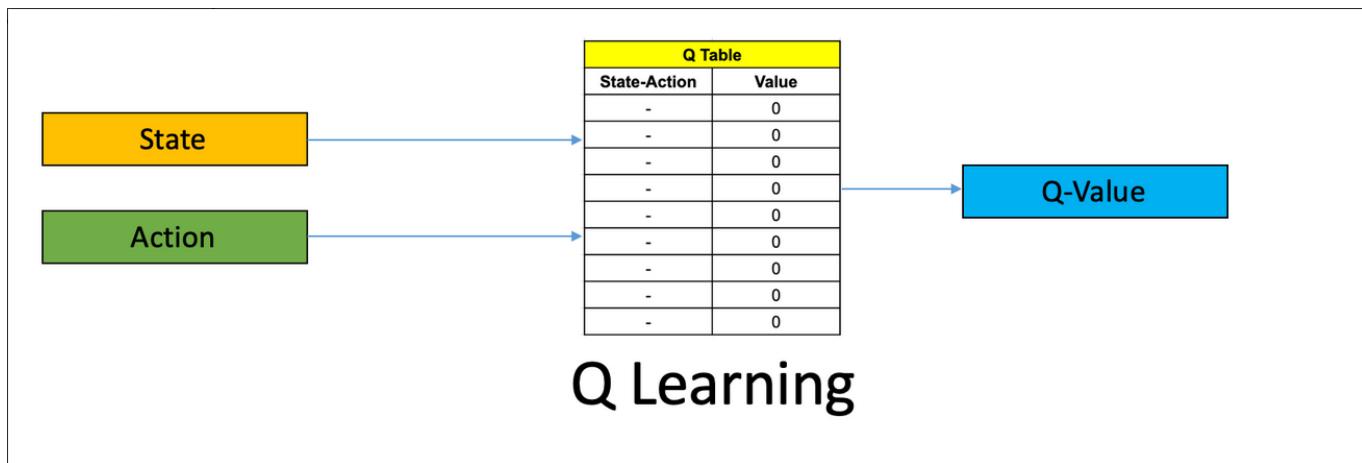


Imagen 1.23: Q-Learning - Fuente: <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python>

Es decir, esta tabla va a contener la información de **recompensa total esperada** para cada valor de estado y acción. Cuando nosotros realizamos el **entrenamiento** de la Q-función, nosotros conseguimos una función que **optimice** esta **Q-tabla**.

Si nosotros tenemos una Q-función óptima ($Q^*(s, a)$), entonces podemos obtener la **política óptima** a partir de ella:

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a)$$

Veamos cuales serían los pasos que deberíamos dar:

Inicializamos nuestra **Q-Tabla** con valores a 0. Conforme avance nuestro entrenamiento estos valores irán cambiando en función de los datos que se obtengan al **porbar** a realizar **acciones** y obtener las **recompensas** correspondientes.

El siguiente elemento que necesitamos es una **política de entrenamiento** (función que nos permita elegir que acción tomar en función del estado en el que estemos), en este caso nuestra política estará basada en los valores de la **Q-tabla**, es lo que llamaremos **explotación** (explotamos la información que tenemos cogiendo la acción con mejor valor Q) o elegiremos otra acción, es lo que llamaremos **exploración** (exploramos nuevos caminos cogiendo una acción de forma aleatoria).

Esto es lo que se llama una política ϵ -greedy, ya que se usa un parámetro ϵ , valor entre 0 y 1, que nos permite decidir si elegimos **explorar** o si queremos **explotar** los datos que ya tenemos.

XXXXX Imagen del gráfico epsilon (epsilon respecto al número de episodios)

Tendremos que:

- con probabilidad $1-\epsilon$ nosotros haremos **explotación** y
- con probabilidad ϵ nosotros haremos **exploración**.

Es decir, inicialmente le damos valor 1 a ϵ de forma que empezaremos haciendo **exploración** e iremos bajando este valor de epsilon conforme avance el entrenamiento para que cada vez usemos más la **explotación**.

La idea base es que al principio del entrenamiento, lo prioritario es **explorar**, es decir, seleccionar una acción al azar y obtener su recompensa, ya que nuestra **Q-Tabla** está inicializada a 0. Conforme avance el entrenamiento nos tendremos que ir fiando más de los datos que ya tenemos y tendrá que primar la **explotación** de nuestros datos de la **Q-Tabla**. Para hacer ésto de una forma efectiva, usaremos un parámetro **decay_epsilon** que conforme avancemos en entrenamiento se encargará de ir reduciendo el valor de ϵ para conseguir este efecto.

Una vez que tenemos nuestros elementos base, pasaremos al **entrenamiento**, de forma que para todos los **episodios** (iteraciones de partidas) que definamos haremos lo siguiente:

- Partimos de un **estado inicial**, y obtenemos una **acción** a partir de nuestra **política de entrenamiento**
- Actualizmos ϵ con el nuevo valor en este episodio
- Iteramos para un número máximo de pasos dentro de este episodio
- Obtenemos el nuevo estado, así como la recompensa obtenida
- Actulizamos el valor de la **Q-Tabla** correspondiente según la fórmula basada en los métodos de **TD** (Diferencias temporales)

$$Q(s, a) = Q(s, a) + \alpha(R(s, a) + \gamma \operatorname{argmax}_a Q(s', a) - Q(s, a)) \text{ donde } s \text{ es el estado actual y } s' \text{ es el nuevo estado}$$

- Verificamos si se ha llegado al final del juego para salir de este episodio si es el caso
- Cambiamos el **estado** como el **nuevo estado**

Una vez acabemos nuestro entrenamiento, obtendremos nuestra **política óptima** como:

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a)$$

Pseudo código Q-Learning

1.9.5 DQN (Deep Q-Learning)

<https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>

Hemos visto el algoritmo de **Q-Learning** en el que usábamos una **Q-Tabla**, es decir una tabla donde guardábamos todos los valores de la función $Q(s, a)$ y que entrenando el agente, éramos capaces de conseguir aproximar a la función **Q óptima**, con lo cual teníamos una **Política Óptima**.

Este tipo de algoritmos son válidos cuando nos encontramos con un número “limitado” de estados y acciones, de forma que la tabla es relativamente manejable y somos capaces de entrenarla. Si nos encontramos ante un problema en el que tenemos miles o cientos de miles de estados no va a ser efectivo construir una tabla y entrenarla para todas las posibles combinaciones **estado-acción**. Para abordar este tipo de problemas, la

Algoritmo 1: Q-Learning

Input: Política entrenamiento, $numero_episodios$, $numero_pasos$, ϵ , $decay_epsilon$

Output: Valores de la función Q optimizada

```

 $Q \leftarrow 0;$ 
for  $i \leftarrow 1$  to  $numero\_episodios$  do
     $\epsilon \leftarrow nuevo\_epsilon(calculado con decay\_epsilon);$ 
     $s_0;$ 
     $t \leftarrow 0;$ 
    for  $t \leftarrow 1$  to  $numero\_pasos \& Not\ finished$  do
        Elegimos una acción  $A_t$  con política entrenamiento;
        Aplicamos la acción  $A_t$  obteniendo  $R_{t+1}, S_{t+1}$ ;
        Actualizamos  $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma argmax_a Q(S_{t+1}, a) - Q(S_t, A_t))$ ;
    return  $Q$ ;

```

Imagen 1.24: Algoritmo Q-Learning - Fuente: Propia

mejor solución es buscar un **aproximador** de la función $Q(s, a)$, que nos permita obtener la mejor solución sin necesidad de entrenar todas las posibles combinaciones.

Para realizar este trabajo una de las posibles opciones es usar **redes neuronales** como función aproximadora y que nos abrirá la posibilidad de trabajar con problemas en los que existan grandes cantidades de estados/acciones.

Fue el equipo de **Deepmind** en 2013 (<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>), en su artículo **“Human-level control through deep reinforcement learning”**, los primeros que decidieron atacar los problemas de alta dimensionalidad de **estados/acciones** mediante el uso de **Redes Neuronales Profundas**. La forma de probar su código fue mediante la implementación de **agentes** que fueran capaces de aprender a jugar a los clásicos **juegos de Atari 2600**. De forma que el agente, recibiendo la información de entrada de los pixels que hay en cada momento en pantalla y el marcador del juego, eran capaces de sobrepasar el rendimiento de algoritmos actuales que hacían ese trabajo. En este caso usaron la misma red neuronal, con la misma arquitectura e hiperparámetros para los 49 juegos con los que se probaron.

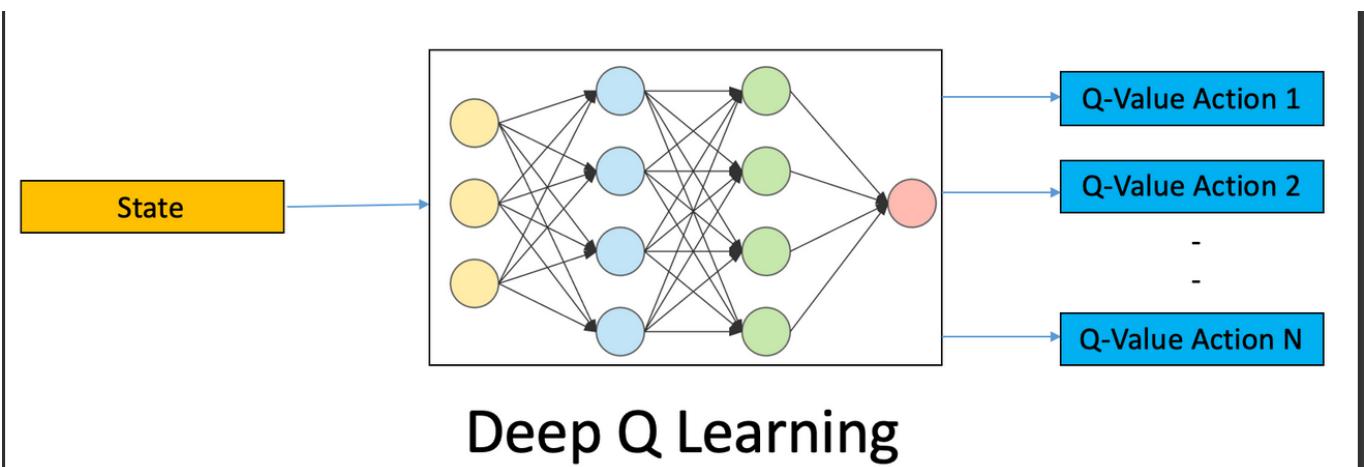


Imagen 1.25: Deep Q-Learning - Fuente: <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>

Con nuestro algoritmo de **Q-Learning** teníamos una función $Q(s, a)$ que implementábamos con un tabla

y nos daba para cada **estado** y cada **acción** cual era el valor de **Q** (Quality) de la recompensa esperada. Ahora, con **Deep Q-Learning** nos encontramos que vamos a tener una red neuronal que será la encargada de para cada **estado** obtener el valor de **Q** para cada posible **acción**.

DQN (Deep Q-Network) Arquitectura

Para poder implementar nuestro trabajo con redes neuronales nos vamos a encontrar con el problema de entrenar la red neuronal (obtener los pesos) que permitan alcanzar nuestra función **Q-Óptima** que nos daría la **Política Óptima** que es lo que realmente buscamos.

Básicamente para realizar el trabajo usaremos 2 redes neuronales que tendrán la misma arquitectura de forma que el entrenamiento sea estable.

- **DQN** que será la red de predicción, y que será la que entrenaremos para minimizar el valor del error $(R + \gamma \text{argmax}_a' Q(s', a', w') - Q(s, a, w))^2$
- **DQN_Target** que será la red que calculará $R + \gamma \text{argmax}_a' Q(s', a', w')$

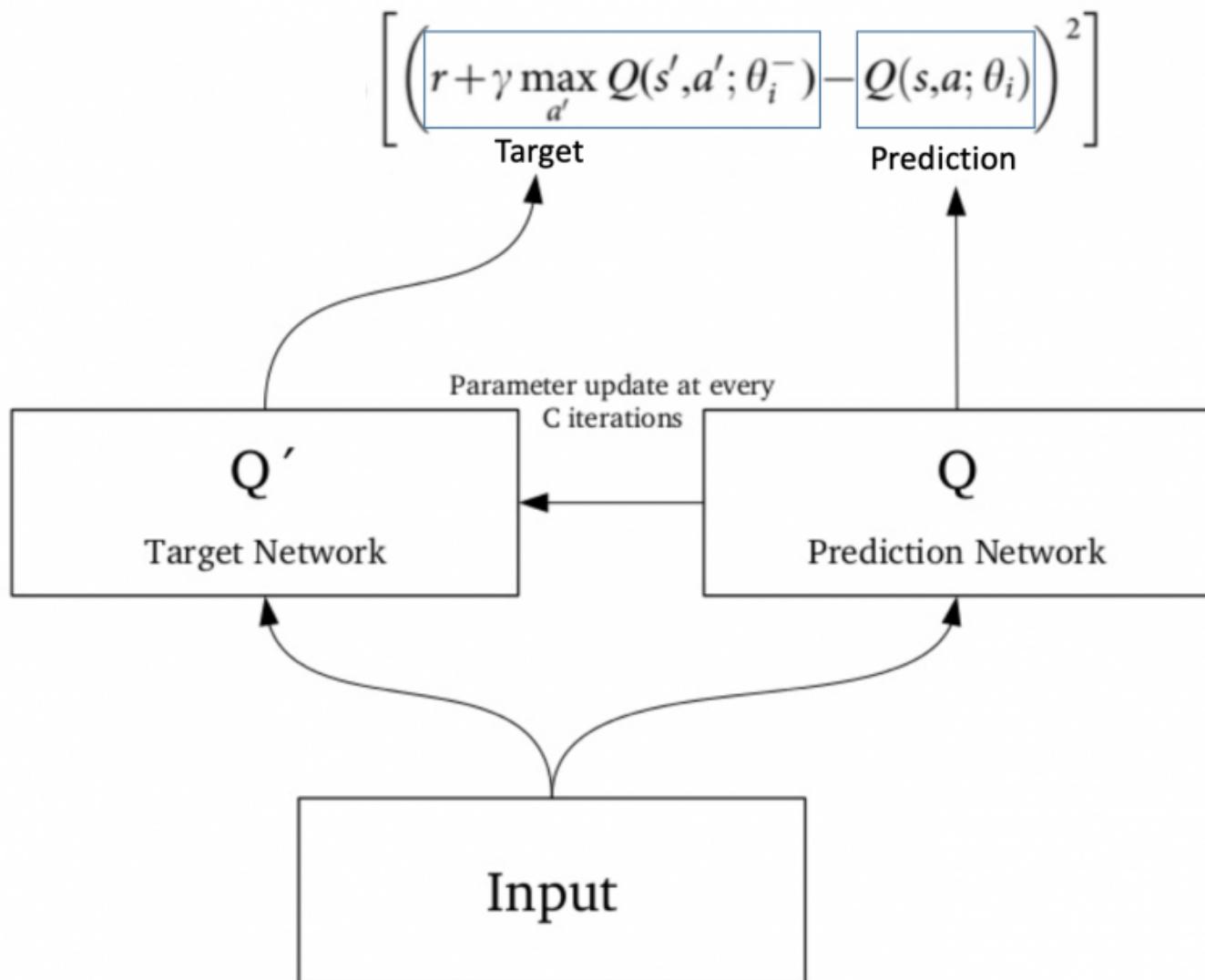


Imagen 1.26: Arquitectura Redes DQN - Fuente: <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>

Experience Replay

El mecanismo del **Experience Replay** nos va a permitir entrenar nuestra red **DQN** con minibatchs que vamos a extraer de forma aleatoria de la memoria en la que vamos a ir guardando los resultados que vamos obteniendo $\langle s, a, r, s' \rangle$.

Ésto nos va a permitir por un lado **entrenar** nuestra **red de predicción** y además va a servirnos para evitar **correlaciones** de secuencias consecutivas que pudieran producir un sesgo en nuestros resultados. De esta manera, al elegir al azar los elementos que vamos a usar para entrenar la red, no tendrán ninguna relación con los datos consecutivos que se van produciendo en los pasos de los episodios.

Algoritmo Deep Q-Learning

- Obtenemos los datos de entrada, que es el estado.
- Seleccionamos la acción usando nuestra política de entrenamiento epsilon-greedy
- Ejecutamos la acción y obtenemos el siguiente estado así como la recompensa obtenida
- Almacenamos en memoria $\langle s, a, r, s' \rangle$
- Si tenemos bastantes elementos en la memoria
 - Hacemos un minibatch aleatorio y enteramos la red siendo $R + \gamma \text{argmax}_{a'} Q(s', a', w')$ el **target de la red** y $Q(s, a, w)$ el valor predicho.
 - La función de pérdida será la de Diferencia de Cuadrados $L = (R + \gamma \text{argmax}'_{a'} Q(s', a', w') - Q(s, a, w))^2$
- Después de cada C iteraciones, copiaremos los pesos de la red DQN a la DQN_Target
- Repetiremos estos pasos durante M episodios

Pseudo-código Deep Q-Learning

Variantes de Deep Q-Learning

- Double Deep Q Network (DDQN) – 2015
- Deep Recurrent Q Network (DRQN) – 2015
- Dueling Q Network – 2015
- Persistent Advantage Learning (PAL) – 2015
- Bootstrapped Deep Q Network – 2016
- Normalized Advantage Functions (NAF) = Continuous DQN – 2016
- N-Step Q Learning – 2016
- Noisy Deep Q Network (NoisyNet DQN) – 2017
- Deep Q Learning for Demonstration (DqfD) – 2017
- Categorical Deep Q Network = Distributed Deep Q Network = C51 – 2017
 - Rainbow – 2017
- Quantile Regression Deep Q Network (QR-DQN) – 2017
- Implicit Quantile Network – 2018

1.9.6 Listado Algoritmos

1. Model-Free

Value-based

[Q-learning = SARSA max](#) – 1992

Algoritmo 2: Deep Q-Learning (DQN)

Input: Política entrenamiento epsilon-greedy, $numero_episodios$, $numero_pasos$, ϵ , $decay_epsilon$, $size_replay_memory$, $size_minibatch$, C

Output: Valores de la función Q optimizada

$DQNetwork \leftarrow Initialize$;

$DQNetwork_target \leftarrow DQNetwork$;

$Replay_Memory$ Initializer con $size_replay_memory$ elementos;

$C \leftarrow 0$;

for $i \leftarrow 1$ **to** $numero_episodios$ **do**

$\epsilon \leftarrow nuevo_epsilon$ (calculado con $decay_epsilon$);

s_0 ;

$t \leftarrow 0$;

for $t \leftarrow 1$ **to** $numero_pasos$ & *Not finished* **do**

Elegimos una acción A_t con política entrenamiento epsilon-greedy;

Aplicamos la acción A_t obteniendo R_{t+1}, S_{t+1} ;

Almacenamos $< S_t, A_t, R_{t+1}, S_{t+1} >$ en $Replay_Memory$;

Obtenemos de forma aleatoria un minibatch desde $Replay_Memory$;

for $t \leftarrow 1$ **to** $size_minibatch$ & *Not finished* **do**

Entrenamos la red $DQNetwork$ con función de pérdida

$L = (R + \gamma argmax'_a Q(s', a', w') - Q(s, a, w))^2$;

Donde $R + \gamma argmax'_a Q(s', a', w')$ lo calculamos con $DQNerwork_Target$ y $Q(s, a, w)$ con $DQNetwork$

if $i \% C = 0$ **then**

Actualizamos la red $DQNetwork_Target$ con $DQNerwork$

$DQNetwork_target \leftarrow DQNetwork$;

return $DQNetwork$;

Imagen 1.27: Algoritmo Deep Q-Learning - Fuente: Propia

State Action Reward State-Action (SARSA) – 1994

Deep Q Network (DQN) – 2013

Double Deep Q Network (DDQN) – 2015

Deep Recurrent Q Network (DRQN) – 2015

Dueling Q Network – 2015

Persistent Advantage Learning (PAL) – 2015

Bootstrapped Deep Q Network – 2016

Normalized Advantage Functions (NAF) = Continuous DQN – 2016

N-Step Q Learning – 2016

Noisy Deep Q Network (NoisyNet DQN) – 2017

Deep Q Learning for Demonstration (DqfD) – 2017

Categorical Deep Q Network = Distributed Deep Q Network = C51 – 2017

- Rainbow – 2017

Quantile Regression Deep Q Network (QR-DQN) – 2017

Implicit Quantile Network – 2018

Mixed Monte Carlo (MMC) – 2017

Neural Episodic Control (NEC) – 2017

Policy-based

Cross-Entropy Method (CEM) – 1999

Policy Gradient

- REINFORCE = Vanilla Policy Gradient (VPG) – 1992
- Policy gradient softmax
- Natural Policy Gradient (Optimisation) (NPG) / (NPO) – 2002
- Truncated Natural Policy Gradient (TNPG) – 2016

Actor-Critic

Advantage Actor Critic (A2C) – 2016

Asynchronous Advantage Actor-Critic (A3C) – 2016

Generalized Advantage Estimation (GAE) – 2015

Trust Region Policy Optimization (TRPO) – 2015

Deterministic Policy Gradient (DPG) – 2014

Deep Deterministic Policy Gradients (DDPG) – 2015

- Distributed Distributional Deterministic Policy Gradients (D4PG) – 2018
- Twin Delayed Deep Deterministic Policy Gradient (TD3) – 2018

[Actor-Critic with Experience Replay \(ACER\)](#) – 2016

[Actor Critic using Kronecker-Factored Trust Region \(ACKTR\)](#) – 2017

[Proximal Policy Optimization \(PPO\)](#) – 2017

- [Distributed PPO \(DPPO\)](#) – 2017
- [Clipped PPO \(CPPO\)](#) – 2017
- [Decentralized Distributed PPO \(DD-PPO\)](#) – 2019

[Soft Actor-Critic \(SAC\)](#) – 2018

General Agents

- [Covariance Matrix Adaptation Evolution Strategy \(CMA-ES\)](#) – 1996
- [Episodic Reward-Weighted Regression \(ERWR\)](#) – 2009
- [Relative Entropy Policy Search \(REPS\)](#) – 2010
- [Direct Future Prediction \(DFP\)](#) – 2016

Imitation Learning Agents

[Behavioral Cloning \(BC\)](#)

[Dataset Aggregation \(Dagger\)](#) (i.e. query the expert) – 2011

[Adversarial Reinforcement Learning](#)

- [Generative Adversarial Imitation Learning \(GAIL\)](#) – 2016
- [Adversarial Inverse Reinforcement Learning \(AIRL\)](#) – 2017

[Conditional Imitation Learning](#) – 2017

[Soft Q-Imitation Learning \(SQIL\)](#) – 2019

Hierarchical Reinforcement Learning Agents

- [Hierarchical Actor Critic \(HAC\)](#) – 2017

Memory Types

- [Prioritized Experience Replay \(PER\)](#) – 2015
- [Hindsight Experience Replay \(HER\)](#) – 2017

Exploration Techniques

- E-Greedy
- Boltzmann
- Ornstein–Uhlenbeck process
- Normal Noise
- Truncated Normal Noise
- Bootstrapped Deep Q Network
- UCB Exploration via Q-Ensembles (UCB)
- Noisy Networks for Exploration
- Intrinsic Curiosity Module (ICM) – 2017

Meta Learning

- Model-agnostic meta-learning (**MAML**) – 2017
- Improving Generalization in Meta Reinforcement Learning using Learned Objectives (MetaGenRLis) – 2020

2. Model-Based

Dyna-Style Algorithms / Model-based data generation

- Dynamic Programming (DP) = **DYNA-Q** – 1990
- Embed to Control (**E2C**) – 2015
- Model-Ensemble Trust-Region Policy Optimization (**ME-TRPO**) – 2018
- Stochastic Lower Bound Optimization (**SLBO**) – 2018
- Model-Based Meta-Policy-Optimization (**MB-MPO**) (meta learning) – 2018
- Stochastic Ensemble Value Expansion (**STEVE**) – 2018
- Model-based Value Expansion (**MVE**) – 2018
- Simulated Policy Learning (**SimPLe**) – 2019
- Model Based Policy Optimization (**MBPO**) – 2019

Policy Search with Backpropagation through Time / Analytic gradient computation

- Differential Dynamic Programming (**DDP**) – 1970
- Linear Dynamical Systems and Quadratic Cost (**LQR**) – 1989
- Iterative Linear Quadratic Regulator (**ILQR**) – 2004
- Probabilistic Inference for Learning Control (**PILCO**) – 2011
- Iterative Linear Quadratic-Gaussian (**iLQG**) – 2012
- Approximate iterative LQR with Gaussian Processes (**AGP-iLQR**) – 2014
- Guided Policy Search (**GPS**) – 2013
- Stochastic Value Gradients (**SVG**) – 2015
- Policy search with Gaussian Process – 2019

Shooting Algorithms / sampling-based planning

Random Shooting (**RS**) – 2017

Cross-Entropy Method (**CEM**) – 2013

- Deep Planning Network (**DPN**) – 2018
- Probabilistic Ensembles with Trajectory Sampling (**PETS-RS** and **PETS-CEM**) – 2018
- Visual Foresight – 2016

Model Predictive Path Integral (**MPPI**) – 2015

- Planning with Deep Dynamics Models (**PDDM**) – 2019

Monte-Carlo Tree Search (**MCTS**) – 2006

- AlphaZero – 2017

2 Modelos Gráficos Probabilísticos y Análisis Causal

2.1 Redes Bayesianas

Al contrario de la Estadística tradicional, el aprendizaje bajo la Estadística Bayesiana tiene un enfoque probabilístico. Así, el razonamiento bayesiano supone que:

- Las hipótesis están gobernadas por una distribución de probabilidad
- Las decisiones son tomadas de forma “óptima” a partir de las observaciones y dichas probabilidades
En este proceso de aprendizaje, las instancias de entrenamiento pueden modificar la probabilidad de una hipótesis, de forma que su planteamiento es mucho menos restrictivo que las técnicas tradicionales (cumplimiento de hipótesis más deterministas). Por tanto, el conocimiento a priori es combinado con las observaciones de los datos con el fin de mejorar la eficiencia de las estimaciones.

Como veremos más adelante, los modelos bayesianos son muy utilizados en todo tipo de investigaciones debido a que proporcionan muy buenos resultados tanto para problemas descriptivos como predictivos:

- Método descriptivo: permite descubrir las relaciones de dependencia/independencia entre las diferentes variables
- Método predictivo: son utilizadas como métodos de clasificación. Entre las características de este tipo de técnicas se pueden citar:
 - Permite realizar inferencias sobre los datos, lo que conlleva a inducir modelos probabilísticos
 - Facilitar la interpretación de otros métodos en términos probabilísticos
 - Se necesita conocer un elevado número de probabilidades
 - Elevado coste computacional al realizar la actualización de las probabilidades

Antes de entrar en detalle en la estructura de los métodos bayesianos definir algunos conceptos:

- Arco: es un par ordenado (X, Y) . En la representación gráfica, un arco (X, Y) viene dado por una flecha desde X hasta Y .
- Grafo dirigido: es un par $G = (N, A)$ donde N es un conjunto de nodos y A un conjunto de arcos definidos sobre los nodos.
- Grafo no dirigido. Es un par $G = (N, A)$ donde N es un conjunto de nodos y A un conjunto de arcos no orientados (es decir, pares noordenados (X, Y)) definidos sobre los nodos. Ciclo: es un camino no dirigido que empieza y termina en el mismo nodo X .
- Grafo acíclico: es un grafo que no contiene ciclos.
- Padre. X es un parente de Y si y sólo si existe un arco $X \rightarrow Y$. Se dice también que Y es hijo de X . Al conjunto de los padres de X se representa como $pa(X)$, y al de los hijos de X por $S(X)$.
- Antepasado o ascendiente. X es un antepasado o ascendiente de Z si y sólo si existe un camino dirigido de X a Z .
- Descendiente. Z es un descendiente de X si y sólo si X es un antepasado de Z . Al conjunto de los descendientes de X lo denotaremos por $de(X)$. - Variable proposicional es una variable aleatoria que toma un conjunto exhaustivo y excluyente de valores. La denotaremos con letras mayúsculas, por ejemplo X , y a un valor cualquiera de la variable con la misma letra en minúscula, x .

- Dos variables X e Y son independientes si se tiene que $P(X/Y) = P(X)$. De esta definición se tiene una caracterización de la independencia que se puede utilizar como definición alternativa: X e Y son independientes sí y sólo sí $P(X,Y) = P(X) \cdot P(Y)$.
- Dos variables X e Y son independientes dado una tercera variable Z si se tiene que $P(X/Y,Z) = P(X/Y)$. De esta definición se tiene una caracterización de la independencia que se puede utilizar como definición alternativa: X e Y son independientes dado Z sí y sólo sí $P(X,Y/Z) = P(X/Z) \cdot P(Y/Z)$. También se dice que Z separa condicionalmente a X e Y.

2.1.1 Modelo Naive Bayes: Hipótesis Map y Teorema de Bayes

La inferencia bayesiana es el eje central de los métodos bayesianos. Bajo ella, las hipótesis son expresadas a partir de distribuciones de probabilidad formuladas según los datos observados, $p(\theta)$, donde θ son magnitudes desconocidas. La función verosimilitud, $p(y/\theta)$, contiene la información disponible en los datos en relación a los parámetros y es ésta la que se usa para actualizar la distribución a priori, $p(\theta)$). Finalmente, para llevar a cabo dicha actualización se emplea el Teorema de Bayes.

Para entender el del **Teorema de Bayes** es necesario definir los siguientes conceptos:

- $P(h)$ es la probabilidad a priori de la hipótesis h. Esta probabilidad contiene la información de que dicha hipótesis sea cierta
- $P(D)$ es la probabilidad a priori de D. Esta es la probabilidad de observar los datos D (sin tener en cuenta la hipótesis que ha de ser cumplida)
- $P(h/D)$ es la probabilidad a posteriori de D, es decir, es la probabilidad de que la hipótesis h una vez los datos D son observados.
- $P(D/h)$ es la probabilidad a posteriori de D, es decir, es la probabilidad de que los datos D sean observados una vez la hipótesis h sea correcta.

Sabiendo que la probabilidad conjunta de un evento dado el otro es proporcional a la probabilidad conjunta de ambos ponderada por la probabilidad del evento condicionante, se tiene:

$$P(h \cap D) = P(h) \cdot P(D | h)$$

$$P(h \cap D) = P(D) \cdot P(h | D)$$

Igualando ambas ecuaciones y manipulando los términos se llega el Teorema de Bayes:

$$P(h | D) = \frac{P(h) \cdot P(D | h)}{P(D)}$$

De forma que la probabilidad a posteriori se puede determinar a partir de la probabilidad a priori y un factor de corrección.

Para una mejora interpretación del Teorema de Bayes se muestra un ejemplo: >**En la sala de Pediatría de un determinado hospital el 60% de los pacientes son niñas. De los niños, se conoce que el 35% tienen menos de 24 meses, mientras que para las niñas el 20% son menores de 24 meses. Un médico selecciona una criatura al azar. Si la criatura tiene menos de 24, ¿cuál es la probabilidad de que sea niña? La tabla siguiente muestra la información que se deduce del enunciado:**

La tabla siguiente muestra la información que se deduce del enunciado:

Probabilidad	Valor
P(niño)	0.40
P(niña)	0.60
P(<24m / niño)	0.35
P(<24m / niña)	0.20

Se obtiene la probabilidad total de que la criatura tenga menos de 24 meses

$$P(< 24m) = P(\text{niño}) \cdot P(< 24m | \text{niño}) + P(\text{niña}) \cdot P(< 24m | \text{niña}) = 0.4 \cdot 0.35 + 0.6 \cdot 0.2 = 0.26$$

Aplicando el teorema de Bayes:

$$P(\text{niña} | < 24m) = \frac{P(\text{niña}) \cdot P(< 24m | \text{niña})}{P(< 24m)} = \frac{0.6 \cdot 0.2}{0.26} = 0.46$$

Por tanto, se tiene un 46% de posibilidades de que el médico haya seleccionado a una niña.

A partir de la *probabilidad a posteriori* obtenida mediante la aplicación del Teorema de Bayes, se está en disposición de maximizar tal expresión; es decir, obtener la hipótesis más probable conocida como **hipótesis MAP (o máximo a posteriori)**:

$$h_{MAP} = \arg \max_h P(h | D) = \arg \max_h [P(h) \cdot P(D | h)]$$

Donde se ha tenido en cuenta que $P(D)$ toma el mismo valor en todas las hipótesis.

Supongamos que estamos tratando de predecir si un estudiante aprueba un examen basándonos en dos características: horas de estudio y nivel de preparación. Nuestras hipótesis son:

- H_1 : el estudiante aprueba el examen
- H_2 : el estudiante no aprueba el examen

Tenemos los siguientes datos:

- $H_1 = 0.7$: probabilidad de que el estudiante apruebe el examen
- $H_2 = 0.3$: probabilidad de que el estudiante NO apruebe el examen
- $P(E | H_1) = 0.8$: probabilidad de que el estudiante estudie suficiente si aprueba
- $P(E | H_2) = 0.8$: probabilidad de que el estudiante estudie suficiente si NO aprueba

Ahora supongamos que un estudiante estudia durante 4 horas y está muy bien preparado. Queremos calcular las probabilidades a posteriori de que el estudiante apruebe o no apruebe el examen, y determinar la hipótesis MAP.

1. Calculamos la probabilidad marginal de observar las evidencias E :

$$P(E) = P(E | H_1) \times P(H_1) + P(E | H_2) \times P(H_2)$$

$$P(E) = (0.8 \times 0.7) + (0.3 \times 0.3) = 0.56 + 0.09 = 0.65$$

2. Calculamos la probabilidad a posteriori de que el estudiante apruebe el examen (H_1) dado que las evidencias E se observan:

$$P(H_1 | E) = \frac{P(E | H_1) \times P(H_1)}{P(E)}$$

$$P(H_1 | E) = \frac{0.8 \times 0.7}{0.65} = \frac{0.56}{0.65} \approx 0.861$$

3. Calculamos la probabilidad a posteriori de que el estudiante no apruebe el examen (H_2) dado que las evidencias E se observan:

$$P(H_2 | E) = \frac{P(E | H_2) \times P(H_2)}{P(E)}$$

$$P(H_2 | E) = \frac{0.3 \times 0.3}{0.65} = \frac{0.09}{0.65} \approx 0.138$$

Por tanto, la hipótesis más probable es que el estudiante apruebe el examen dado que ha estudiado durante 4 horas y está bien preparado.

Nota: Dado que $P(E)$ es constante para ambas hipótesis, se podría haber comparado directamente $P(H_1 | E)$ y $P(H_2 | E)$ para determinar la hipótesis MAP.

El uso de la *hipótesis MAP* puede ser aplicado para resolver problemas de clasificación.

Como sabemos, en dichas investigaciones se tiene una variable independiente conocida como clase o target y un conjunto de variables predictoras o atributos. Así, el Teorema de Bayes se puede reescribir como:

$$P(C | (A_1, A_2, \dots, A_N)) = \frac{P(C) \cdot P((A_1, A_2, \dots, A_N) | C)}{P(A_1, A_2, \dots, A_N)}$$

Donde C denota el target o clase y A_i el conjunto de variables explicativas.

Haciendo máxima la probabilidad de C dado los atributos se tiene:

$$c_{MAP} = \arg \max_{c \in \Delta} P(C | (A_1, A_2, \dots, A_N)) = P(C) \cdot P((A_1, A_2, \dots, A_N) | C)$$

siendo Δ el conjunto de valores que puede tomar la variable objetivo (target del problema).

Como puede verse, el enfoque planteado es bastante sencillo pero también muy costoso desde el punto de vista computacional ya que es necesario conocer las distribuciones de probabilidad de las variables implicadas en la investigación.

2.1.2 Modelo Naïve-Bayes

El clasificador Naïve-Bayes es una versión simplificada del proceso de modelización anterior. Este método supone que todos los atributos son independientes conocido el valor de la variable clase de forma que la función de probabilidad conjunta queda como:

$$P(C | (A_1, A_2, \dots, A_N)) = P(C) \cdot \prod_{i=1}^N P(A_i | C)$$

Como es de esperar, el supuesto que subyace este clasificador no es muy realista; si bien, alcanza muy buenos resultados por lo que su uso está muy extendido en la comunidad de científico de datos.

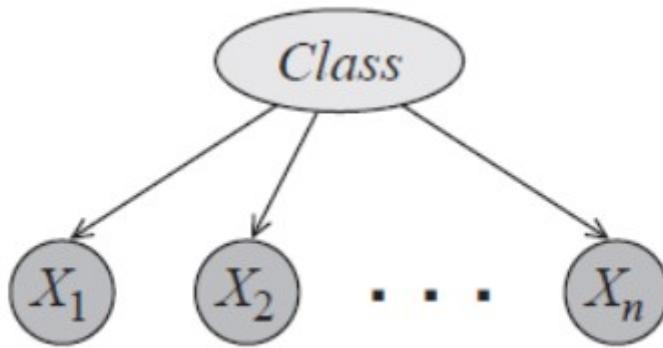


Imagen 2.1: Naive bayes

Como en el caso anterior, se obtiene la hipótesis que maximiza la probabilidad del valor de la clase.

$$c_{MAP} = \arg \max_{c \in \Delta} \left(P(C) \cdot \prod_{i=1}^N P(A_i | C) \right)$$

El clasificador Naïve-Bayes puede emplearse tanto con variables explicativas discretas como numéricas.

Cuando las variables explicativas son discretas, la probabilidad condicional es obtenida a partir de la frecuencia de los datos muestrales; de forma que ésta se define como el número de casos favorables entre el número de casos posibles. Matemáticamente, se tiene:

$$P(x_i | \text{pa}(x_i)) = \frac{n(x_i, \text{pa}(x_i))}{n(\text{pa}(x_i))}$$

Donde $n(x_i, \text{Pa}(x_i))$ denota el número de registros de la muestra en el que la variable X_i toma el valor x_i y $\text{pa}(x_i)$ los padres de X_i . Notar que el padre de cada variable explicativa es la variable independiente, la cual se ha denominado target o clase.

En el caso en que el tamaño de la muestra de trabajo sea pequeño, el uso de las frecuencias puede ocasionar estimaciones poco fiables por lo que se emplean estimadores basados en suavizados. Uno de los más empleados es el estimador de Laplace en el que la probabilidad viene expresada por el número de casos favorables + 1 dividida por el de casos totales más el número de alternativas.

$$P(x_i | \text{Pa}(x_i)) = \frac{n(x_i, \text{pa}(x_i)) + 1}{n(\text{pa}(x_i)) + \alpha}$$

Por su parte, si se dispone de variables numéricas el estimador Naïve-Bayes supone que dichas variables siguen una distribución normal donde la media y la desviación típica son estimadas a partir de los datos de la muestra. Sin embargo, en la mayor parte de las ocasionales, las variables continuas no suelen seguir una distribución de probabilidad normal es posible que las estimaciones sean poco eficientes por lo que se recomienda transformar dichas variables en cualitativas (por ejemplo: empleando los intervalos que se obtienen al tomar los cuantiles de su distribución).

```

1 import os
2 import numpy as np
3 import pandas as pd
4
5 datos = pd.read_csv("../datos/credit_g.csv")
6
7 datos.info()

```

```

1 # Pasamos las variables a categóricas
2 datos['checking_status'] = datos['checking_status'].astype('category')
3 datos['credit_history'] = datos['credit_history'].astype('category')
4 datos['purpose'] = datos['purpose'].astype('category')
5 datos['savings_status'] = datos['savings_status'].astype('category')
6 datos['employment'] = datos['employment'].astype('category')
7 datos['personal_status'] = datos['personal_status'].astype('category')
8 datos['other_parties'] = datos['other_parties'].astype('category')
9 datos['property_magnitude'] = datos['property_magnitude'].astype('category')
10 datos['other_payment_plans'] = datos['other_payment_plans'].astype('category')
11 datos['housing'] = datos['housing'].astype('category')
12 datos['job'] = datos['job'].astype('category')
13 datos['property_magnitude'] = datos['property_magnitude'].astype('category')
14 datos['own_telephone'] = datos['own_telephone'].astype('category')
15 datos['foreign_worker'] = datos['foreign_worker'].astype('category')
16 datos['class'] = datos['class'].astype('category')

```

```

1 # La variable class es una variable reservada en diferentes módulos de Python ->
2 # reemplazar por por target
3 datos.rename(columns={'class': 'target'}, inplace=True)
4 datos['target']=np.where(datos['target']=='good', 0, 1) # cambio en la codificación por
# sencillez en el preprocesado

```

```

1 # Definición de la muestra de trabajo
2 datos_entrada = datos.drop('target', axis=1) # Datos de entrada
3 datos_entrada = pd.get_dummies(datos_entrada, drop_first=True, dtype=int) #conversión a
# variables dummy
4
5 target = datos["target"] # muestra del target

```

```

1 from sklearn.preprocessing import StandardScaler
2 from sklearn.model_selection import train_test_split, RepeatedStratifiedKFold,
   ↵ GridSearchCV
3
4 # Partición de la muestra
5
6 test_size = 0.3 # muestra para el test
7 seed = 222 # semilla
8
9 X_train, X_test, y_train, y_test = train_test_split(
10     datos_entrada, target, test_size=test_size, random_state=seed, stratify=target
11 )
12
13 # Estandarización de la muestra
14 esc = StandardScaler().fit(X_train) # valores media y std de los datos de train
15
16 # aplicación a los datos de train y test
17 X_train_esc = esc.transform(X_train)
18 X_test_esc = esc.transform(X_test)
19
20 # Validación cruzada
21 cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=2, random_state=seed)

```

2.1.2.1 Bernoulli Naive Bayes

```

1 from sklearn.naive_bayes import BernoulliNB
2 bernoulli_nb=BernoulliNB(force_alpha=False)
3
4 grid=[{'alpha': list(np.arange(0.05, 1, 0.1)), 'binarize': [0.3, 0.1, 0.0]}]
5
6 # Definición del modelo con hiperparámetros
7 gs_bernoulli_nb = GridSearchCV(
8     estimator=bernoulli_nb, param_grid=grid, scoring='accuracy', cv=cv, n_jobs=1,
9     ↵ return_train_score=False
10 )
11 gs_bernoulli_nb = gs_bernoulli_nb.fit(X_train, y_train)
12
13 print(f'Naive-Bayes (Bernoulli) (parámetros): {gs_bernoulli_nb.best_params_}') #
14     ↵ parámetros del modelo final
15
16 bernoulli_nb = gs_bernoulli_nb.best_estimator_ # modelo final
17
18 # Resultados importantes de estos algoritmos (acceso dentro del objeto del modelo)
19 print(bernoulli_nb.class_log_prior_) # logaritmo de la probabilidad de cada clase
20 print(bernoulli_nb.class_log_prior_) # logaritmo de la probabilidad de cada clase
21 bernoulli_nb.feature_log_prob_ # logaritmo de la probabilidad de la variable dada la
   ↵ clase ( $P(X_i|Y)$ )

```

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3
4 from sklearn.metrics import accuracy_score, roc_curve, auc, confusion_matrix
5
6 import warnings
7 # Suprimir todas las advertencias
8 warnings.simplefilter("ignore")
9
10
11 # Predicciones muestra entrenamiento y test
12
13 preds_train = bernoulli_nb.predict(X_train)
14 preds_test = bernoulli_nb.predict(X_test)
15
16 # Cálculo métricas bondad de ajuste
17 print('Accuracy')
18 print('-----')
19 print(f'Entrenamiento (cv): {round(gs_bernoulli_nb.best_score_,5)}')
20 accuracy_test = accuracy_score(y_test, preds_test)
21 print(f'Test: {round(accuracy_test,5)}')
22
23 # AUC - test y curva roc (final)
24 y_pred_test = bernoulli_nb.predict_proba(X_test)
25 fp_rate_test, tp_rate_test, thresholds = roc_curve(y_test, y_pred_test[:,1])
26 auc_test = auc(fp_rate_test, tp_rate_test)
27
28 # Bondad de ajuste: matriz de confusión y curva roc para los datos de test
29
30 f, axes = plt.subplots(1, 2, figsize=(10,5))
31
32 sns.heatmap(confusion_matrix(preds_test, y_test), annot=True, cmap=plt.cm.Reds,
   ↵ fmt='.\u00b2f', ax=axes[0]) # matriz de confusión
33 sns.lineplot(x=fp_rate_test, y=tp_rate_test, color='skyblue', label='AUC = %0.2f' %
   ↵ auc_test, ax=axes[1]) # curva roc
34
35 plt.legend(loc="lower right")
36 plt.show()

```

2.1.2.2 Gaussian Naive Bayes

```

1 from sklearn.naive_bayes import GaussianNB
2
3 gaussian_nb = GaussianNB()
4 grid=[{'var_smoothing': list(np.arange(0,0.1, 0.02))}]

```

```
5 # Definición del modelo con hiperparámetros
6 gs_gaussian_nb=GridSearchCV(
7     estimator=gaussian_nb, param_grid=grid, scoring='accuracy', cv=cv, n_jobs=1,
8     ↵ return_train_score=False
9 )
10
11 gs_gaussian_nb = gs_gaussian_nb.fit(X_train, y_train)
12 print('Naive-Bayes (Bernoulli) (parámetros):', gs_gaussian_nb.best_params_)
13
14 #parámetros del modelo final
15 gaussian_nb = gs_gaussian_nb.best_estimator_ #modelo final
16
17 # predicciones muestra entrenamiento y test
18
19 preds_train = gaussian_nb.predict(X_train)
20 preds_test = gaussian_nb.predict(X_test)
21
22 # Cálculo métricas bondad de ajuste
23
24 print('Accuracy')
25 print('-----')
26 print(f'Entrenamiento (cv):, {round(gs_gaussian_nb.best_score_,5)}')
27 accuracy_test = accuracy_score(y_test, preds_test)
28 print('Test:', round(accuracy_test,5))
29
30 #AUC - test y curva roc (final)
31
32 y_pred_test = gaussian_nb.predict_proba(X_test)
33 fp_rate_test, tp_rate_test, thresholds = roc_curve(y_test, y_pred_test[:,1])
34 auc_test = auc(fp_rate_test, tp_rate_test)
35
36 # Bondad de ajuste: matriz de confusión y curva roc para los datos de test
37
38 f, axes = plt.subplots(1, 2, figsize=(10,5))
39
40 sns.heatmap(confusion_matrix(preds_test, y_test), annot = True, cmap = plt.cm.Reds,
41     ↵ fmt='%.0f', ax=axes[0]) # matriz de confusión
42 sns.lineplot(x=fp_rate_test, y=tp_rate_test, color='skyblue', label='AUC = %.2f' %
43     ↵ auc_test, ax=axes[1]) # curva roc
44
45 plt.legend(loc="lower right")
46 plt.show()
```

2.2 Modelos Bayesianos

Las redes bayesianas son métodos estadísticos que representan la incertidumbre a través de las relaciones de independencia condicional que se establecen entre ellas. Por tanto, permiten modelar un fenómeno a partir de dichas relaciones y hacer inferencia.

Este tipo de métodos son una representación gráfica de dependencias para razonamiento probabilístico, en las que los nodos representan variables aleatorias y los arcos las relaciones de dependencia directa entre las variables.

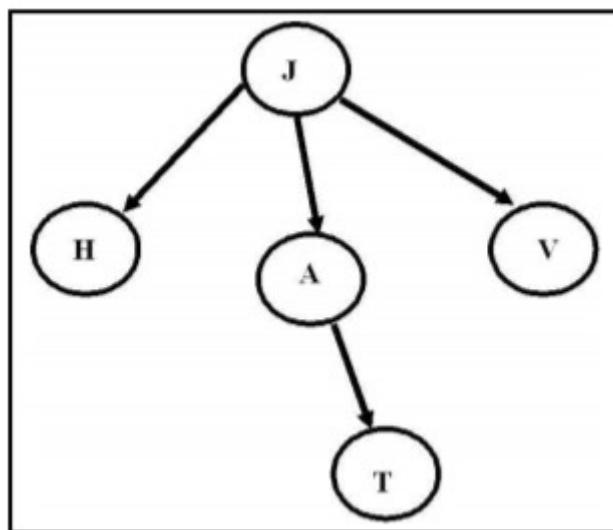


Imagen 2.2: Topología Bayesiana

La ventaja de las redes bayesianas frente a otros métodos es la posibilidad de codificar las dependencias/independencias relevantes considerando no sólo las dependencias marginales sino también las dependencias condicionales entre un conjunto de variables.

En definitiva, las redes bayesianas modelan las relaciones entre las variables tanto de forma cualitativa como cuantitativa. La fuerza de dichas relaciones viene dada en las distribuciones de probabilidad como una medida de la creencia que tenemos sobre esas relaciones en el modelo.

2.2.1 Formulación general

Una red bayesiana queda especificada formalmente por una dupla $B=(G,\Theta)$ donde G es un grafo dirigido acíclico (DAG, por las siglas en inglés) y Θ es el conjunto de distribuciones de probabilidad. Definimos un grafo como un par $G = (V, E)$, donde V es un conjunto finito de vértices, nodos o variables, y E es un subconjunto del producto cartesiano $V \times V$ de pares ordenados de nodos que llamamos enlaces o aristas. Por tanto, puede decirse que las redes bayesianas representan el conocimiento cualitativo del modelo mediante el grafo dirigido acíclico.

Supongamos una red bayesiana que contiene un padre A y 3 hijos (B , C y D), siendo C también padre de B . El DAG que definido sería:

```

1 import bnlearn as bn
2 import matplotlib.pyplot as plt
3
4 edges = [('A', 'B'), ('A', 'C'), ('A', 'D'), ('C', 'B')]
5 DAG = bn.make_DAG(edges, methodtype="bayes")
6
7 bn.plot(DAG, interactive=False)
8 plt.show()
9
10 # print(DAG["adjmat"]) # podemos ver el dag en formato tabla (no visual cuando existen
    ↵ muchos nodos)

```

El grafo define un modelo probabilístico mediante el producto de varias funciones de probabilidad condicionada:

$$P(x_1, \dots, x_n) = \prod_{i=1}^N P(x_i | \text{pa}(x_i))$$

Con $\text{pa}(x_i)$ las variables inmediatamente predecesoras de la variable X_i . En este sentido, los valores de probabilidades $P(x_i / \text{pa}(x_i))$ son “almacenados” en el nodo que precede a la variable X_i .

Es importante resaltar que de no existir la expresión anterior, la red debiese ser descrita a partir de la probabilidad conjunta, lo que obligaría a trabajar con un número de parámetros mucho más elevado (creciente de forma exponencial en el número de nodos).

2.2.2 Independencia condicional e inferencia de la red

Como se ha comentado anteriormente, una variable X es condicionalmente independiente de otra variable Y dada una tercera Z si, el hecho de que se tenga conocimiento Z, hace que Y no tenga influencia en X.

$$P(X|Y, Z) = P(X|Z)$$

Por tanto, la hipótesis de **independencia condicional** establece que cada nodo debe ser independiente de los otros nodos de la red (salvo sus descendientes) dados sus padres. Dicho de otro modo, si se conocen los padres de una variable, ésta se vuelve independiente del resto de sus predecesores.

Veamos un ejemplo para facilitar la comprensión de la independencia condicional.

Partiendo de la red bayesiana de la imagen anterior, la probabilidad conjunta se define como:

$$P(X_1, X_2, \dots, X_9) = P(X_1) \cdot P(X_2) \cdot P(X_3 | X_2, X_1) \cdot P(X_4 | X_3, X_2, X_1) \quad (2.1)$$

$$\cdot P(X_5 | X_4, X_3, X_2, X_1) \cdot P(X_6 | X_5, X_4, X_3, X_2, X_1) \quad (2.2)$$

$$\cdot P(X_7 | X_6, X_5, X_4, X_3, X_2, X_1) \quad (2.3)$$

$$\cdot P(X_8 | X_7, X_6, X_5, X_4, X_3, X_2, X_1) \quad (2.4)$$

$$\cdot P(X_9 | X_8, X_7, X_6, X_5, X_4, X_3, X_2, X_1) \quad (2.5)$$

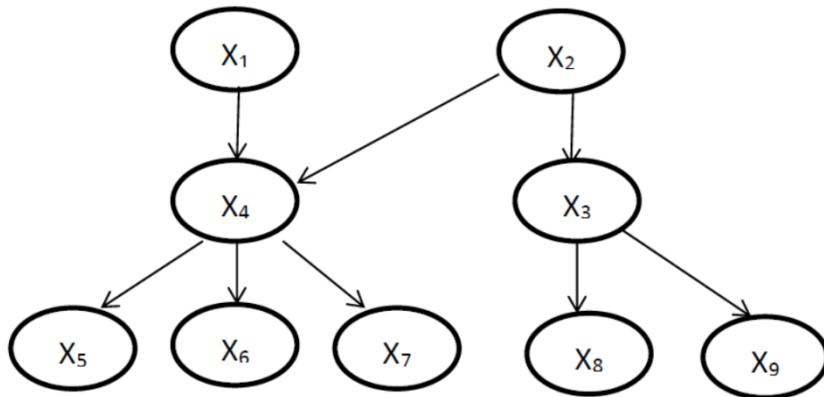


Imagen 2.3: Topología Bayesiana

En cambio, como las probabilidades condicionales solo dependen de sus padres (teorema anterior), la probabilidad conjunta toma la siguiente forma:

$$P(X_1, X_2, \dots, X_9) = P(X_1) \cdot P(X_2) \cdot P(X_3 | X_2) \cdot P(X_4 | X_2, X_1) \quad (2.6)$$

$$\cdot P(X_5 | X_4) \cdot P(X_6 | X_4) \cdot P(X_7 | X_4) \quad (2.7)$$

$$\cdot P(X_8 | X_3) \cdot P(X_9 | X_3) \quad (2.8)$$

Por tanto, *la propiedad de independencia de las redes bayesianas hace que se reduzca en gran medida los cálculos**.

En una red bayesiana, se conoce como **inferencia probabilística** a la propagación del conocimiento a través de la misma una vez se tienen nuevos datos. Este proceso se lleva a cabo actualizando las probabilidades a posteriori en toda la estructura de la red mediante el Teorema de Bayes.

Como es de imaginar, el proceso de inferencia es muy costoso computacionalmente de forma que, dependiendo de las necesidades, se emplean algoritmos exactos o aproximados:

- Exactos: cuando puede calcularse la inferencia de forma exacta. El coste computacional necesario para la actualización de las probabilidades es viable
- Aproximados: se usan técnicas de muestreo que permita calcular de forma aproximada la inferencia. Usado cuando no es viable obtener la propagación exacta en un tiempo razonable

2.2.3 Aprendizaje de las redes bayesianas

Como se ha visto, para determinar una red bayesiana es necesario especificar su estructura gráfica y una función de probabilidad conjunta. Dicho proceso es bastante laborioso debido a que, en muchos casos, se desconoce ambas especificaciones. Para paliar esta circunstancia, se han desarrollado diferentes métodos de aprendizaje. Así, el proceso de aprendizaje de una red bayesiana puede dividirse en dos etapas:

- Estructural (o dimensión cualitativa): búsqueda en el espacio de posibles redes
- Paramétrico (o dimensión cuantitativa): aprende la distribución de probabilidad a partir de los datos, dada la red

El *aprendizaje paramétrico* consiste en hallar los parámetros asociados a la estructura de la red. Estos parámetros están constituidos por las probabilidades de los nodos raíz y las probabilidades condicionales de las demás variables dados sus padres. Las probabilidades previas se corresponden con las marginales de los nodos raíz y las condicionales se obtienen de las distribuciones de cada nodo con sus padres.

En el *aprendizaje estructural* es donde se establecen las relaciones de dependencia que existen entre las variables del conjunto de datos para obtener el mejor grafo que represente estas relaciones. Este problema se hace prácticamente intratable desde el punto de vista computacional cuando el número de variables es grande. Por ello, suelen emplearse algoritmos de búsqueda para aprender la estructura de la red.

A continuación, se presentan algunos algoritmos de búsqueda para establecer la estructura de una red bayesiana.

Algoritmo K2

El algoritmo K2 es considerado el predecesor de otros algoritmos de búsqueda más sofisticados. basado en búsqueda y optimización de una métrica bayesiana es considerado como el predecesor y fuente de inspiración para las generaciones posteriores. El proceso de búsqueda de este algoritmo está dividido en las siguientes etapas: - Ordenación de los nodos (variables de entrada) de forma que los posibles padres de una variable aparezcan siempre antes de ella para evitar la generación de ciclos. Esta restricción provoca que el algoritmo busque los padres posibles entre las variables predecesoras (ventaja computacional) - Partiendo de este orden establecido, se calcula la ganancia que se produce en la medida al introducir una variable como padre

Finalmente, el proceso se repite para cada nodo mientras el incremento de calidad supere un cierto umbral preestablecido.

Algoritmo B

Este algoritmo elimina la dependencia de la ordenación previa de los nodos de forma que su coste de computación es superior al algoritmo K2. complejidad computacional es mayor. Como en el caso anterior, el proceso es iniciado con padres vacíos con padres vacíos y en cada etapa se añade aquel enlace que maximice el incremento de calidad eliminando aquellos que producen ciclos. El proceso es detenido cuando una vez la inclusión de un arco no represente ninguna ganancia.

Algoritmo Hill Climbing

El algoritmo Hill Climbing (HC) es un procedimiento de búsqueda que parte de una solución inicial y, a partir de ésta, mediante técnicas heurística se calcula el nuevo valor utilizando todas las soluciones vecinas a la solución actual, seleccionando el vecino que mejor solución presenta. Por tanto, este algoritmo finaliza cuando no existe ningún vecino que pueda mejorar la solución vecina.

Una variante muy útil y muy empleada consiste en considerar todos los posibles movimientos a partir del estado actual y elegir el mejor de ellos como nuevo estado. A este método se le denomina ascensión por la máxima pendiente o búsqueda del gradiente.

Vamos a mostrar un ejemplo de **aprendizaje de la estructura** en python:

```

1 import pandas as pd
2
3 datos = pd.read_csv("../datos/bayesian_data.csv", sep=";", index_col="Unnamed: 0")
4 datos = datos.rename(columns={'class': 'target'}) # target con 4 categorías
5
6
7 # Modelo de estructura
8 structure_model = bn.structure_learning.fit(datos, methodtype='tan', root_node="doors",
    ↴ class_node="target") # uso de hill-climbing

```

9

10 # nota: en este caso no estamos definiendo un padre para obtener la estructura bayesian

1 structure_model["adjmat"]

```
1 import matplotlib.pyplot as plt
2 bn.plot(model)
3 plt.show()
```

Tanto del cuadro como del grafo, podemos ver que:

- target es padre de: safety, lug_boot y person
- target es hijo de: buying y maint

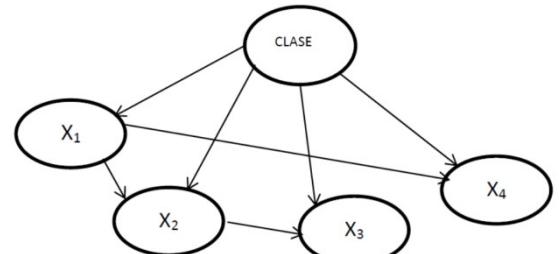
2.2.4 Clasificadores

Como determinar la estructura de la red bayesiana es una tarea realmente compleja, la mayor parte de los modelos de clasificación basados en redes bayesianas suelen ser modificaciones del clasificador Naïve-Bayes.

A día de hoy, existen muchos clasificadores de forma que se exponen brevemente tres de los más utilizados.

Tan: Tree Augmented Naïve Bayes

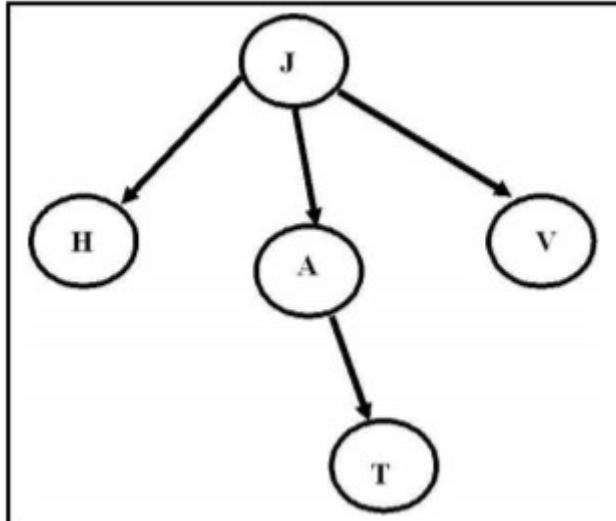
En el modelo TAN todos los atributos tienen como padre a otro atributo como mucho, además de la clase en sí,



de forma que cada atributo obtiene un arco aumentado apuntando a él.

Ban: Naïve Bayes aumentado

En este modelo se incorporan nuevos arcos entre todas las variables con la limitación de que no formen ciclos. Destacar la relevancia de este clasificador ya que su estructura es capaz de representar cualquier forma de

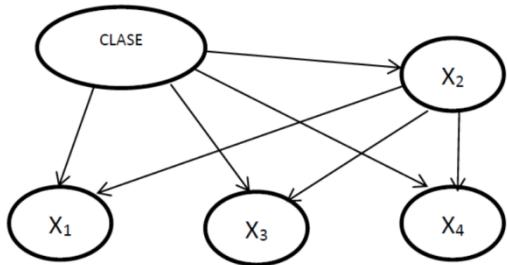


red bayesiana.

AODE: Average One-Dependence Estimators

Al igual que el algoritmo TAN, cada variable tiene como padre a la variable clase y como máximo a otro atributo. Sin embargo, la principal diferencia respecto al modelo anterior tiene lugar en la forma de obtener la predicción definitiva del modelo. Dicha predicción consiste en:

- El algoritmo establece posibles estructuras de red compatibles con el problema y, en función de ésta, hace una predicción de la clase
- La predicción final se obtiene como la media ponderada de las predicciones anteriores



Una vez visto la parte teórica entramos en detalle a nivel práctico.

```

1 structure_tan_model = bn.structure_learning.fit(
2     datos,
3     methodtype='tan',
4     root_node="doors", # hay que tener en cuenta algún hijo que no tenga más padre que el
5     ↵ target
6     class_node="target" # en el modelo tan hay que tener una clase/padre)
7
8 parameter_model = bn.parameter_learning.fit(structure_tan_model, datos,
9     ↵ methodtype='bayes', verbose=0)
10
11 structure_tan_model["model_edges"] # bordes y nodos. También podría pintarse como en el
12 ↵ caso anterior
  
```

- Obención de las probabilidades condicionadas

```

1 # Probabilidades condicionadas
2
3 CPDs = bn.print_CPD(parameter_model, verbose=0) # esto es un diccionario de dataframes
   ↵ (clave cada columna del df

```

- Para doors:

```

1 CPDs["doors"] [CPDs["doors"] ["target"] == 0]

```

- Para maint (y primera clase del target):

```

1 CPDs["maint"] [CPDs["maint"] ["target"] == 0]

```

Obtención de las Predicciones sobre la muestra

```

1 feats = list(datos.columns)
2 feats.remove("target")
3
4 # dado las evidencias de dos variables, calculamos la probabilidad de la clase
5 query = bn.inference.fit(parameter_model, variables=["target"], evidence={'doors':2,
   ↵ 'lug_boot': 'small'}, verbose=0)
6
7 query.df

```

Por último, presentamos un ejemplo de uso de clasificador bayesiano empleando la librería **pyAgrum**. Esta librería es que es un contenedor de Python para la biblioteca aGrUM de C++. Proporciona una interfaz de alto nivel a la parte de aGrUM que permite crear, modelar, aprender, usar, calcular e integrar redes bayesianas y otros modelos gráficos probabilísticos como las redes de Markov o los modelos relacionales probabilísticos.

La librería se integra adecuadamente con *scikit-learn* por lo que se recomienda su uso para desarrollar clasificadores bayesianos.

```

1 import os
2
3 import pandas as pd
4 import numpy as np
5
6 import pyAgrum.skbn as skbn
7 import pyAgrum.lib.notebook as gnb
8
9 datos = pd.read_csv("../datos/credit_g.csv")
10
11 datos.info()

```

```

1 # Pasamos las variables a categóricas
2 datos['checking_status'] = datos['checking_status'].astype('category')
3 datos['credit_history'] = datos['credit_history'].astype('category')
4 datos['purpose'] = datos['purpose'].astype('category')
5 datos['savings_status'] = datos['savings_status'].astype('category')
6 datos['employment'] = datos['employment'].astype('category')
7 datos['personal_status'] = datos['personal_status'].astype('category')
8 datos['other_parties'] = datos['other_parties'].astype('category')
9 datos['property_magnitude'] = datos['property_magnitude'].astype('category')
10 datos['other_payment_plans'] = datos['other_payment_plans'].astype('category')
11 datos['housing'] = datos['housing'].astype('category')
12 datos['job'] = datos['job'].astype('category')
13 datos['property_magnitude'] = datos['property_magnitude'].astype('category')
14 datos['own_telephone'] = datos['own_telephone'].astype('category')
15 datos['foreign_worker'] = datos['foreign_worker'].astype('category')
16 datos['class'] = datos['class'].astype('category')

17
18 # La variable class es una variable reservada en diferentes módulos de Python ->
19 # reemplazar por por target
20 datos.rename(columns={'class': 'target'}, inplace=True)
21 datos['target']=np.where(datos['target']=='good', 0, 1) # cambio en la codificación por
22 # sencillez en el preprocesado

23 # Definición de la muestra de trabajo
24 datos_entrada = datos.drop('target', axis=1) # Datos de entrada
25 datos_entrada = pd.get_dummies(datos_entrada, drop_first=True, dtype=int) #conversión a
26 # variables dummy

27 target = datos["target"] # muestra del target

28 from sklearn.preprocessing import StandardScaler
29 from sklearn.model_selection import train_test_split, RepeatedStratifiedKFold,
30 # GridSearchCV

31 # Partición de la muestra

32 test_size = 0.3 # muestra para el test
33 seed = 222 # semilla

34 X_train, X_test, y_train, y_test = train_test_split(
35     datos_entrada, target, test_size=test_size, random_state=seed, stratify=target
36 )
37

38 # Estandarización de la muestra
39 esc = StandardScaler().fit(X_train) # valores media y std de los datos de train

40 # aplicación a los datos de train y test
41 X_train_esc = esc.transform(X_train)
42 X_test_esc = esc.transform(X_test)
43

```

```

1 # Creación del clasificador TAN en python
2 bayesian_network = skbn.BNClassifier(
3     learningMethod='TAN',
4     prior='Smoothing',
5     scoringType='BIC',
6     priorWeight=0.5,
7     discretizationStrategy='quantile',
8     usePR=True,
9     significant_digit = 6
10)
11
12 bayesian_network.fit(X_train, y_train) # ajuste del modelo

1 from sklearn.metrics import accuracy_score
2
3 # predicciones para la muestra de train y test
4
5 train_probs = bn.predict_proba(X_train)
6 test_probs = bn.predict_proba(X_test)
7
8 # predict-proba proporciona las probabilidades
9
10 def preds_ones(probs, threshold = 0.5):
11     return np.where(probs[:, 0] > threshold, 0, 1)
12
13 y_train_pred = preds_ones(train_probs)
14 y_test_pred = preds_ones(tests_probs)
15
16 print(f'Accuracy (train) {round(accuracy_score(y_train, y_train_pred),2)}')
17 print(f'Accuracy (test) {round(accuracy_score(y_test, y_test_pred), 2)}')

```

2.3 Modelos Ocultos de Markov

2.3.1 Cadenas de Markov

Una cadena de Markov es un sistema matemático que experimenta transiciones de un estado a otro de acuerdo con un conjunto dado de reglas probabilísticas. La siguiente imagen presenta una representación gráfica de una cadena de Markov.

Como puede verse, una cadena de Markov puede ser planteada como un gráfico dirigido en el que los nodos son los estados y los arcos contienen la probabilidad de pasar de un estado a otro.

Las cadenas de Markov son procesos estocásticos pero se diferencian en que carecen de memoria. Así, en un proceso de Markov la probabilidad del siguiente estado del sistema depende solamente del estado actual del sistema y no de ningún estado anterior.

$$P(x_i | x_0 \dots x_{i-1}) = P(x_i | x_{i-1})$$

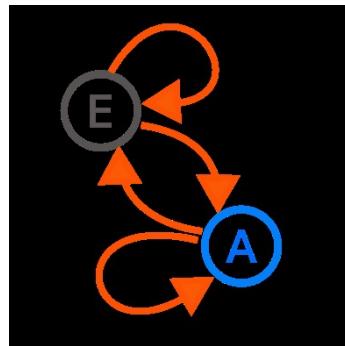


Imagen 2.4: Cadena Markow

La expresión anterior se conoce como **propiedad de Markov**.

Es importante destacar que una cadena de Markov puede ser vista como una red bayesiana en la que cada nodo tiene una tabla de probabilidad correspondiente a $P(x_t | x_{t-1})$ y es la misma para todos los nodos salvo para el instante inicial.

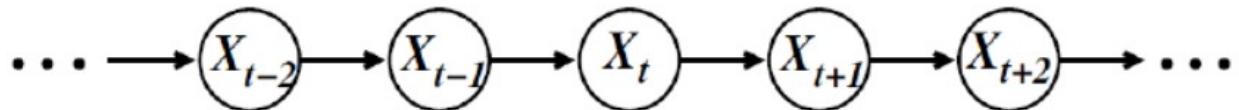


Imagen 2.5: Cadena Markov

En toda cadena de Markov es necesario definir una matriz de transición, T , la cual contiene la información sobre la probabilidad de transición entre los diferentes estados del sistema. Como hecho relevante, cada fila de la matriz debe ser un vector de probabilidad y la suma de todos sus términos debe ser igual a la unidad.

Asimismo, las matrices de transición tienen la propiedad de que el producto de las matrices posteriores puede describir las probabilidades de transición a lo largo de un intervalo de tiempo. Esta característica permite modelar la probabilidad de estar en un determinado estado después de n pasos como:

$$p^n = p^0 * T^n$$

Veamos un ejemplo con el que facilitar la comprensión del funcionamiento de una cadena de Markov.

Un grupo farmacéutico ha sacado al mercado tres pomadas hace pocas semanas. Con el fin de conocer su acogida así como el comportamiento futuro de los potenciales clientes ante las tres variantes del producto ha realizado un estudio de mercado. De dicho estudio se conocen las probabilidades de cambio de un tipo de pomada a otra.

La matriz de transición para T es:

$$T = \begin{pmatrix} 0.80 & 0.10 & 0.10 \\ 0.03 & 0.95 & 0.02 \\ 0.20 & 0.05 & 0.75 \end{pmatrix}$$

Sabiendo que actualmente, la participación en el mercado de las tres pomadas es:

$$p = \begin{pmatrix} 0.30 \\ 0.45 \\ 0.25 \end{pmatrix}$$

¿Cuáles serán las participaciones de mercado de cada marca en dos meses más?

La matriz de transición para T^2 es:

$$T^2 = \begin{pmatrix} 0.663 & 0.180 & 0.155 \\ 0.057 & 0.907 & 0.037 \\ 0.312 & 0.105 & 0.584 \end{pmatrix}$$

De forma que usando la fórmula anterior, se tiene:

$$p^2 = p^0 \cdot T^2 = (0.30 \ 0.45 \ 0.25) \begin{pmatrix} 0.663 & 0.180 & 0.155 \\ 0.057 & 0.907 & 0.037 \\ 0.312 & 0.105 & 0.584 \end{pmatrix} = (0.302 \ 0.488 \ 0.209)$$

En vista de los resultados, la cuota de mercado de cada tipo de pomada variará en los dos meses siguientes en: - Pomada 1: de un 30% a 30,2% (estable) - Pomada 2: de un 45% a un 48,8% (leve aumento) - Pomada 3: de un 25% a un 20,9% (ligera caída)

2.3.2 Cadena de Markov absorbente

Una **cadena de Markov absorbente** es una cadena de Markov en la que para algunos estados una vez ingresados, no es posible salir. Sin embargo, este es solo uno de los requisitos previos para que una cadena de Markov sea una cadena de Markov absorbente. Para que sea una cadena de Markov absorbente, todos los demás estados transitorios deben poder alcanzar el estado absorbente con una probabilidad de 1.

Con el fin de ayudar al entendimiento del comportamiento de una **cadena de Markov absorbente**, se plantea una simulación en python sobre la calidad crediticia de n individuos y su comportamiento durante un año (12 pagos).

Suponiendo un modelo de impago bancario con los siguientes tres estados: - Pago al día - Pago con retraso - Impago (estado absorbente)

Así, la matriz de transición para esta cadena de Markov es:

$$T = \begin{pmatrix} 0.8 & 0.1 & 0.0 \\ 0.2 & 0.4 & 0.4 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Esto significa que hay un 80% de probabilidad de que un individuo que paga al día continúe pagando al día, un 20% de probabilidad de que pase a un estado de pago con retraso, y un 0% de probabilidad de que entre en estado de impago (para pasar a impago debe pasar previamente por pago con retraso). Además, hay un 20% de probabilidad de que un individuo en estado de pago con retraso vuelva al estado de pago al día, un 40% de probabilidad de que permanezca en estado de pago con retraso y un 20% de probabilidad de que entre en estado de impago. Por último, el estado de impago es absorbente, lo que significa que una vez que un individuo entra en estado de impago, permanece allí indefinidamente.

```

1 import numpy as np
2
3 np.random.seed(123)
4
5 # Matriz de transición completa
6 transition_matrix = np.array([[0.8, 0.2, 0.0],   # De pago al día a pago con retraso o
7                             ↵ impago
8                             [0.45, 0.4, 0.15],   # De pago con retraso a pago al día o
9                             ↵ impago
10                            [0.0, 0.0, 1.0]]) # De impago a impago (estado de
11                             ↵ absorción)
12
13 # Muestra de individuos + número de pagos
14 n_samples = 10
15 n_pagos = 12
16
17 y = np.zeros(n_samples, dtype=int) # Todos los individuos comienzan en estado de pago al
18                             ↵ día
19
20 muestra_dict = {} # Diccionario para recoger los pagos de cada muestra
21 for i in range(n_samples):
22     # Generar transiciones de estado basadas en la matriz de transición completa
23     current_state = 0 # Estado inicial: pago al día
24     pagos_muestra_list = [] # Obtener secuencia en cada mes de pago
25     for _ in range(n_pagos): # Realizar los 12 pagos
26         if current_state == 0: # Si estamos en el estado de pago al día
27             # solo nos quedamos con las posibles transiciones (no es posible ir al impago
28             ↵ sin tener retraso en pago)
29             next_state = np.random.choice([0, 1],
30                                         ↵ p=transition_matrix[current_state][0:2])
31         elif current_state == 1: # Si estamos en el estado de pago con retraso
32             # una vez estamos en retraso pago podemos volver a regular pagos (pago al
33             ↵ día) o ir a impago
34             next_state = np.random.choice([0, 1, 2], p=transition_matrix[current_state])
35         else: # Si estamos en el estado de impago
36             y[i] = 1 # estado absorbente
37             break
38         current_state = next_state
39         pagos_muestra_list.append(current_state)
40     muestra_dict[f"Individuo_{i}"] = pagos_muestra_list

```

En el diccionario `muestra_dict` se ha guardado el comportamiento de cada individuo a lo largo de los 12 pagos posteriores al punto inicial.

```
1 muestra_dict
```

Como puede verse, la mayor parte de individuos no llegan al estado de impago y esto es consecuencia de las probabilidades existentes en la matriz de transición de partida.

La secuencia de pagos del *Individuo_5* hace que sea de interés focalizarse en él para detallar el impacto que tienen las cadenas de markov. Como puede verse, al inicio de pago se empieza a retrasar hasta volver a regularizar sus pagos a mediados del segundo trimestre. Tras esta regularización, meses después vuelve a caer de estado.

Las **cadenas de Markov** absorbentes tienen algunas propiedades específicas que las diferencian de las cadenas de Markov más simples. La más destacada es la referida a la forma en que la matriz de transición puede ser escrita. Sea una cadena con t estados transitorios y r estados absorbentes, la matriz de transición T puede escribirse en su forma canónica como:

$$T = \begin{pmatrix} Q & R \\ 0 & I_t \end{pmatrix}$$

Donde Q es una matriz de txt , R es una matriz de txr , 0 es una matriz de ceros de rxt e I_t es la matriz identidad de txt .

En particular, la descomposición de la matriz de transición en la matriz fundamental permite ciertos cálculos, como el *número esperado de pasos hasta la absorción de cada estado*. La matriz fundamental N se calcula de la siguiente manera:

$$N = (I_t - Q)^{-1}$$

Siendo I_t es la matriz identidad de txt . Así, para obtener el *número esperado de pasos* se calcula como:

$$n = N * 1$$

Donde 1 denota un vector columna de valor uno y longitud igual al número estados transitorios.

Por último, la probabilidad de que un estado transitorio sea absorbido es calculada como:

$$p_{trans \rightarrow abs} = N * R$$

Veamos un ejemplo de Cadena de Markov absorbente con el que podamos ver en detalle estos cálculos matriciales:

Imaginemos un cliente en un casino. Por cada apuesta gana 1€ con probabilidad de 0.3 o pierde 1€ con probabilidad de 0.7. Sabiendo que la apuesta ha sido iniciada con 2 € y que el cliente se retirará se retirará si pierde todo el dinero o bien lo duplica. Se pide:

- **Cuestión 1: Escribir la matriz de transición de una cadena de Markov**
- **Cuestión 2: Determinar el promedio de apuestas hasta que el juego termina**
- **Cuestión 3: Determinar la probabilidad de terminar el juego con 4€ o de marcharse de vacío**

Cuestión 1: Del enunciado se conoce que se tienen 5 posibles estados (0, 1, 2, 3, 4) siendo los estados 0 y 4 absorbentes (pierde todo o duplica la apuesta, respectivamente). Teniendo en cuenta los posibles movimientos y las probabilidades asociadas se tiene:

$$T = \begin{pmatrix} t_{00} & t_{01} & t_{02} & t_{03} & t_{04} \\ t_{10} & t_{11} & t_{12} & t_{13} & t_{14} \\ t_{20} & t_{21} & t_{22} & t_{23} & t_{24} \\ t_{30} & t_{31} & t_{32} & t_{33} & t_{34} \\ t_{40} & t_{41} & t_{42} & t_{43} & t_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.7 & 0 & 0.3 & 0 & 0 \\ 0 & 0.7 & 0 & 0.3 & 0 \\ 0 & 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Cuestión 2: Se escribe la matriz T en su forma canónica. Notar que para ello es necesario reorganizar los estados (ahora, los estados absorbentes están en las últimas filas de la matriz T).

$$T = \begin{pmatrix} Q & R \\ 0 & I_t \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & t_{13} & t_{10} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{20} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{30} & t_{34} \\ t_{01} & t_{02} & t_{03} & t_{00} & t_{04} \\ t_{41} & t_{42} & t_{43} & t_{40} & t_{44} \end{pmatrix} = \begin{pmatrix} 0 & 0.3 & 0 & 0.7 & 0 \\ 0.7 & 0 & 0.3 & 0 & 0 \\ 0 & 0.7 & 0 & 0 & 0.3 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

De forma que Q y R son:

$$Q = \begin{pmatrix} 0.0 & 0.3 & 0.0 \\ 0.7 & 0.0 & 0.3 \\ 0.0 & 0.7 & 0.0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.7 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.3 \end{pmatrix}$$

El número de apuestas hasta terminar el juego es:

$$N = (I_t - Q)^{-1} * 1 = \begin{pmatrix} 0.0 & 0.3 & 0.0 \\ 0.7 & 0.0 & 0.3 \\ 0.0 & 0.7 & 0.0 \end{pmatrix}^{-1} * \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.362 & 0.517 & 0.155 \\ 1.207 & 1.724 & 0.517 \\ 0.845 & 1.207 & 1.362 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2.034 \\ 3.448 \\ 3.414 \end{pmatrix}$$

Teniendo en cuenta que el cliente empezó su apuesta con 2€, el número de apuestas esperadas hasta que el juego acabe son 3.448€.

Cuestión 3: En este caso, se sabe que la probabilidad de llegar a un estado absorbente desde uno transitorio sigue la siguiente expresión:

$$p_{trans \rightarrow abs} = N * R = (I_t - Q)^{-1} * R = \begin{pmatrix} 1.362 & 0.517 & 0.155 \\ 1.207 & 1.724 & 0.517 \\ 0.845 & 1.207 & 1.362 \end{pmatrix} * \begin{pmatrix} 0.7 & 0 \\ 0 & 0 \\ 0 & 0.3 \end{pmatrix} = \begin{pmatrix} 0.953 & 0.046 \\ 0.845 & 0.155 \\ 0.591 & 0.409 \end{pmatrix}$$

Así, la probabilidad de que el cliente acabe con 4€ es de 15.5%. Por su parte, se tiene un 84.5% de posibilidades de que se vaya de vacío.

2.3.3 Modelos Ocultos de Markov

Los **Modelos Ocultos de Markov**, HMMs (por sus siglas en inglés) son una extensión de las cadenas de Markov y sirven para tratar tanto eventos observables (presentes en la cadena de entrada) como eventos ocultos que consideramos causales del modelo probabilístico. Los Modelos Ocultos de Markov son utilizados cuando se conocen las evidencias sobre un sistema pero no los estados tienen lugar de forma que buscan establecer la relación existente entre los estados visibles y los ocultos. Algunos ejemplos de uso de este tipo de modelos:

- Separación de secuencias de nucleótidos por sus características biológicas (exón-intrón)
- Relacionar proteínas con sus funcionalidades
- Localización de genes en las células eucariotas
- Reconocimiento del habla
- Etiquetado de texto y traducción automática

En un HMM, para cada instante de tiempo o posición t en una secuencia se tiene:

- Una variable aleatoria X_t , con posibles estados s_1, \dots, s_n (no observables directamente)
- Otra variable aleatoria E_t , con posibles estados v_1, \dots, v_m (observaciones)

Para un buen funcionamiento de este tipo de modelos se asume dos propiedades:

- Propiedad de Markov: en cada posición, el estado solo depende del estado en la posición inmediatamente anterior: $P(X_t | Y, X_{t-1}) = P(X_t | X_{t-1})$
- Independencia de las observaciones: en cada posición, la observación solo depende del estado en esa posición: $P(E_t | Y, X_t) = P(E_t | X_t)$

De forma análoga a las cadenas de Markov, un HMM también puede ser expresado según una red bayesiana:

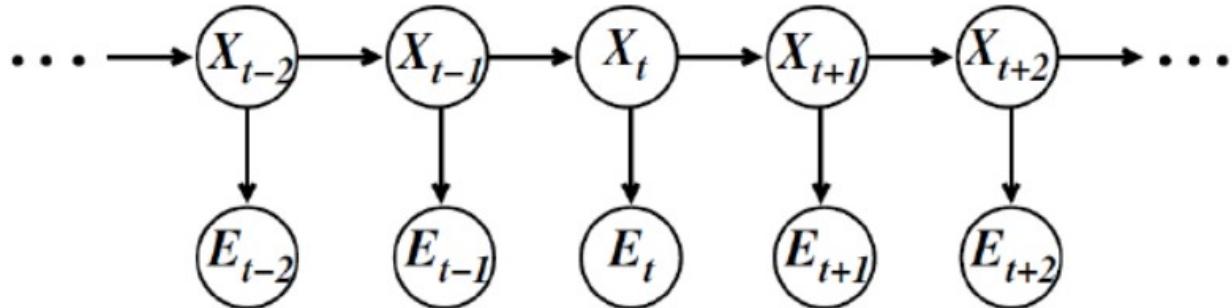


Imagen 2.6: HMM

Así, cada nodo X_t tiene la misma tabla de probabilidad correspondiente a $P(X_t | X_{t-1})$ salvo en el instante anterior. Por el contrario, cada nodo E_t tiene una única tabla de probabilidad correspondiente a $P(E_t | X_t)$.

Además de los estados ocultos y observables comentados anteriormente, un Modelo Oculto de Markov consta también de otros elementos que son citados a continuación:

- Respecto a los estados ocultos:

- La matriz de probabilidades entre los estados, A , denominada matriz de transición. Así, $a_{ij} = P(X_t = s_j | X_{t-1} = s_i)$ es la probabilidad de pasar del estado si al estado s_j . Es importante destacar que el modelo probabilístico que describe la manera de transitar entre una posición y la siguiente no cambia a lo largo de la secuencia.
- El vector de probabilidades a priori de cada estado, π , con $\pi_i = P(x_1 = s_i)$ - Respecto a las observaciones: - La matriz de probabilidades de los observables, B , conocida como matriz de observación. Así, $b_{ij} = P(E_t = v_j | X_t = s_i)$ es la probabilidad de observar v_j cuando el estado es s_i

Es importante destacar que el modelo probabilístico que describe la emisión de la observación en cada estado no cambia a lo largo de la secuencia.

Por tanto, un HMM está formado por la combinación de dos tipos de modelos: - El transicional el cual responde a los estados ocultos - El modelo de evidencias que tiene en cuenta la información disponible de las observaciones

Un ejemplo básico sobre el uso de *Modelos Ocultos de Markov* en **bioinformática** se plantea a continuación. En este ejemplo, se parte de una secuencia de ADN ficticia (observaciones) y se hace uso de un HHM para predecir la probabilidad de los estados ocultos (“codificación de genes” y “regiones no codificantes”) en la secuencia de ADN.

```

1 import numpy as np
2 from hmmlearn import hmm
3
4 np.random.seed(444)
5
6 dna_sequence = "TCGAATCGAAGTATCGGCATTGGCTCGAGCGATCGATGCTAGCA"
7 states = ["Gene", "Non-Gene"]
8
9 # Conversión de la secuencia de ADN a números para que el modelo HMM pueda procesarla
10 # Por ejemplo, A=0, C=1, G=2, T=3
11 dna_encoded = np.array([[0 if base == "A" else 1 if base == "C" else 2 if base == "G"
   ↵ else 3 for base in dna_sequence]]).T

```

```

1 # Definir y entrenar el modelo
2 model = hmm.CategoricalHMM(n_components=2, n_iter=100) # las componentes son los estados
3 model.fit(dna_encoded)

```

```

1 model.predict_proba(dna_encoded)[0:20] # probabilidades de decodificación

```

```

1 # Decodificar los estados ocultos (genes vs no genes) utilizando el modelo entrenado
2 decoded_states = model.predict(dna_encoded) # predict asume un threshold de 0.5
3
4 # Decodificar los estados ocultos a sus etiquetas originales
5 decoded_states_labels = [states[state] for state in decoded_states]
6
7 print(f"Secuencia de ADN: {dna_sequence}")
8 print(f"Estados ocultos predichos: {decoded_states_labels}")

```

Dado una secuencia de observaciones $o_1 o_2 \dots o_t$, mediante un **Modelo Oculto de Markov** se pueden responder a distintos tipos de problemas como:

- Filtrado: permite conocer la probabilidad de que $X_t = q$
- Explicación más verosímil: también conocida como decodificación, permite conocer la secuencia de estados más probable.

A continuación, se presenta un ejemplo para explicar en detalle el proceso de obtención del **filtrado** y de la **explicación más verosímil en un Modelo Oculto de Markov**.

Suponga un trabajador en una plataforma de petróleo que no tiene contacto con el exterior en todo un año. Debido a su profesión, desconoce la situación meteorológica de cada día (si llueve o no), pero todas las mañanas siempre ve llegar al gerente a su oficina. El gerente unos días viene con paraguas y otros no. Imagine entonces que un sistema formado por dos estados ocultos (lluvia, no lluvia) y dos observaciones (paraguas, no paraguas) es utilizado para pronosticar el tiempo por el trabajador. La siguiente imagen muestra la estructura de un Modelo Oculto de Markov en formato de red.

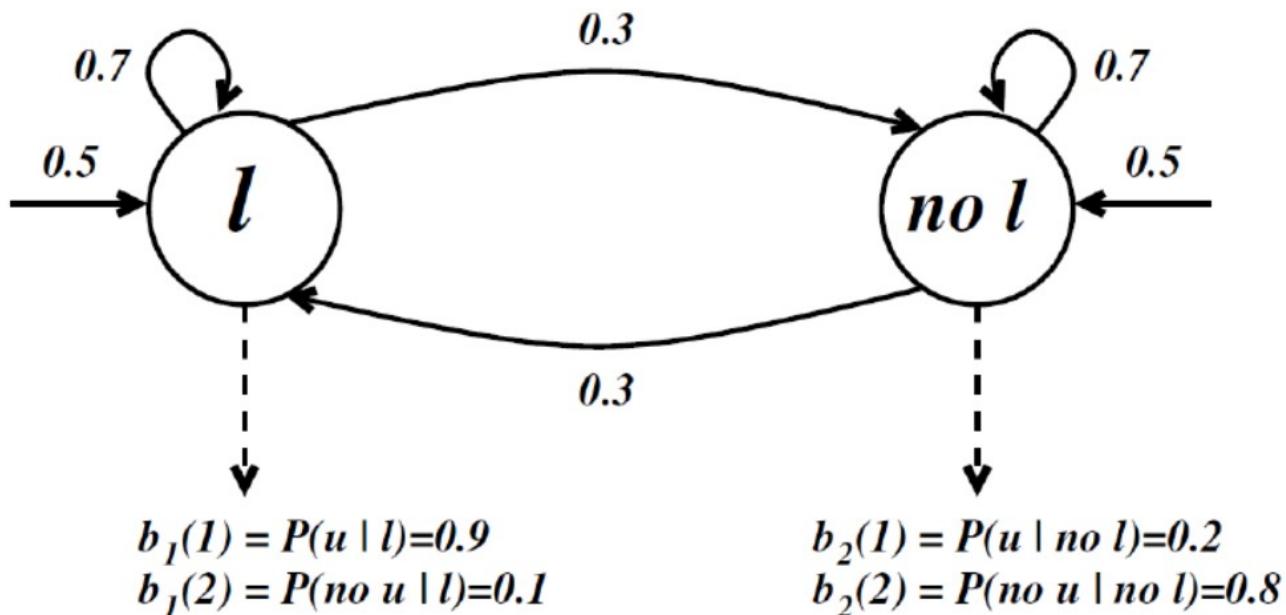


Imagen 2.7: Ejemplo HHM

El ejemplo es detallado tanto siguiendo los cálculos “manualmente” como a partir de una implementación en python.

Los vectores de información a priori como las matrices de probabilidad entre estados y las matrices de probabilidad de observables se obtienen directamente del enunciado:

```

1 import numpy as np
2
3 # Definir parámetros del modelo HMM como listas y diccionarios
4
5 states = ('lluvia', 'no_lluvia')
6 observations = ('paraguas', 'no_paraguas')
7

```

```

8 start_probability = {'lluvia': 0.5, 'no_lluvia': 0.5} # Vector de información a priori
9
10 # Matrices de probabilidad entre estados
11 transition_probability = {
12     'lluvia': {'lluvia': 0.7, 'no_lluvia': 0.3},
13     'no_lluvia': {'lluvia': 0.3, 'no_lluvia': 0.7},
14 }
15
16 # Matriz de probabilidad de observables
17 emission_probability = {
18     'lluvia': {'paraguas': 0.9, 'no_parguas': 0.1},
19     'no_lluvia': {'paraguas': 0.2, 'no_parguas': 0.8},
20 }

```

2.3.3.1 Filtrado

2.3.3.1.1 Implementación del Algoritmo Forward

Se define la función para calcular la probabilidad conjunta de una secuencia de observaciones y estados usando el algoritmo de avance (forward).

```

1 def forward(obs, states, start_p, trans_p, emit_p):
2     alpha = np.zeros((len(obs), len(states)))
3
4     # Inicializar primer paso
5     for i, state in enumerate(states):
6         alpha[0][i] = start_p[state] * emit_p[state][obs[0]]
7
8     # Recorrer el resto de la secuencia de observaciones
9     for t in range(1, len(obs)):
10        for i, current_state in enumerate(states):
11            alpha[t][i] = sum(alpha[t-1][j] * trans_p[states[j]][current_state] *
12                             emit_p[current_state][obs[t]] for j in range(len(states)))
13
14     return alpha

```

```

1 # Secuencia de observaciones y estados de los tres primeros días
2 observations_sequence = ['paraguas', 'paraguas', 'no_parguas']
3
4 # Calcula la probabilidad conjunta de la secuencia de observaciones y estados usando el
4 # algoritmo de avance
5 alpha = forward(observations_sequence, states, start_probability, transition_probability,
5 #                emission_probability)
6 alpha
7
8 # Suma de las probabilidades en el último paso para obtener la probabilidad total de la
8 # secuencia de observaciones
9 probability_sequence = np.sum(alpha[-1])

```

```
1 alpha[-1] / probability_sequence # Probabilidad normalizada en el último paso (día 3)
```

Así, la probabilidad de que el día 3 sea lluvia es del 19%

2.3.3.2 Explicación más verosimil

2.3.3.2.1 Implementación del algoritmo Viterbi

Función para calcular la secuencia de estados más probable utilizando el algoritmo Viterbi

```
1 def viterbi(obs, states, start_p, trans_p, emit_p):
2     V = []
3     path = []
4
5     # Inicializar primer paso
6     for state in states:
7         V[0][state] = start_p[state] * emit_p[state][obs[0]]
8         path[state] = [state]
9
10    # Recorrer el resto de la secuencia de observaciones
11    for t in range(1, len(obs)):
12        V.append([])
13        new_path = []
14
15        for current_state in states:
16            (prob, state) = max(
17                (V[t - 1][previous_state] * trans_p[previous_state][current_state] *
18                 emit_p[current_state][obs[t]], previous_state)
19                for previous_state in states
20            )
21            V[t][current_state] = prob
22            new_path[current_state] = path[state] + [current_state]
23
24        path = new_path
25
26    # Encontrar el estado final con la mayor probabilidad
27    (prob, state) = max((V[len(obs) - 1][final_state], final_state) for final_state in
28                         states)
29
30    return (prob, path[state])
```

Se aplica la función y se obtiene tanto la secuencia de estados ocultos más probable como la probabilidad de ésta.

```

1 prob, path = viterbi(observations_sequence, states, start_probability,
2   ↵ transition_probability, emission_probability)
3 print(f"Secuencia de estados ocultos más probable: {path}")
3 print(f"Probabilidad de la secuencia más probable: {prob}")

```

2.3.3.3 Aplicación de un HMM: Post-tagging

El **post-tagging** es una tarea fundamental en el procesamiento del lenguaje natural (NLP por sus siglas en inglés) que consiste en asignar etiquetas gramaticales a cada palabra en una oración después de haber sido segmentada en palabras individuales. Esta tarea es crucial para comprender el significado y la estructura de las oraciones, ya que las etiquetas gramaticales proporcionan información sobre la función sintáctica de cada palabra.

En el contexto del post-tagging, los estados del HMM representan las etiquetas gramaticales de las palabras, las transiciones representan la dependencia entre las etiquetas gramaticales de las palabras consecutivas y las emisiones representan la probabilidad de que una palabra dada se observe en un estado determinado.

Para realizar el post-tagging con un HMM, se sigue el siguiente procedimiento:

- *Entrenamiento del modelo*: se entrena con un conjunto de datos de oraciones etiquetadas, aprendiendo las probabilidades de transición y emisión
- *Predicción de etiquetas*: para una nueva oración sin etiquetar, el modelo predice la secuencia de etiquetas gramaticales más probable para la oración, utilizando el algoritmo de Viterbi

Ventajas

- *Flexibilidad*: pueden modelar secuencias de palabras con diferentes patrones gramaticales

Interpretabilidad: Los estados del HMM pueden interpretarse como diferentes tipos de palabras o estructuras gramaticales.

Robustez: Los HMMs son robustos a errores de segmentación de palabras y a palabras desconocidas.

Limitaciones

- *Dependencia de datos*: el rendimiento del modelo depende de la calidad y cantidad de datos de entrenamiento disponibles
- *Ambigüedad gramatical*: pueden no ser capaces de resolver ambigüedades gramaticales en oraciones complejas
- *Necesidad de preprocessamiento*: requiere preprocessamiento previo de las oraciones, como la segmentación de palabras.

```

1 import warnings
2
3 import nltk
4 import numpy as np
5 from hmmlearn import hmm
6
7 warnings.filterwarnings("ignore")
8

```

```

9 from nltk.corpus import brown # corpus con etiquetado
10
11 # Cargar las sentencias etiquetadas del corpus brown
12 tagged_sentences = brown.tagged_sents(tagset='english')
13
14 # Crear un diccionario de palabras y un diccionario de etiquetas
15 word2idx = {}
16 tag2idx = {}
17
18 # Iterar sobre las sentencias etiquetadas para construir los diccionarios
19 for sentence in tagged_sentences:
20     for word, tag in sentence:
21         if word.lower() not in word2idx:
22             word2idx[word.lower()] = len(word2idx)
23         if tag not in tag2idx:
24             tag2idx[tag] = len(tag2idx)
25
26 # Estos diccionarios serán útiles para convertir palabras y tags en índices numéricos que
27 # nuestro modelo HMM pueda entender.
28
29 # Conjunto de entrenamiento
30 words_train = [] # Lista de palabras (en minúsculas por lower)
31 tags_train = [] # Lista de etiquetas
32 for sentence in tagged_sentences:
33     words, tags = zip(*sentence)
34     words_train.append([word.lower() for word in words])
35     tags_train.append(tags)

```

```

1 # Creación y entrenamiento del modelo HMM
2 model = hmm.MultinomialHMM(n_components=len(tag2idx), init_params="ste") # estados
3 # ocultos como número de etiquetas
4 model.fit(
5     X=np.array([word2idx[word] for words in words_train for word in words]).reshape(-1,
6     1),
7     lengths=[len(words) for words in words_train]
8 ) # El entrenamiento se hace conviriendo a índices las palabras

```

```

1 # Función para realizar post-tagging en una nueva sentencia en castellano
2 def post_tag(model, sentence, word2idx, tag2idx):
3
4     # Convertir las palabras de la sentencia a índices
5     word_idxs = [word2idx[word.lower()] for word in sentence if word.lower() in word2idx]
6
7     # Si no hay palabras conocidas, devolver None
8     if len(word_idxs) == 0:
9         return None
10
11     # Realizar post-tagging utilizando el modelo HMM

```

```
12     predicted_tags = model.predict(np.array(word_idxs).reshape(-1, 1))
13
14     # Convertir los índices de etiquetas a etiquetas POS
15     predicted_tags = [list(tag2idx.keys())[list(tag2idx.values()).index(tag)] for tag in
16     ↵ predicted_tags]
17
18     return list(zip(sentence, predicted_tags))
```

```
1 sentence = "I love Python"
2 predicted_tags = post_tag(model, sentence.split(), word2idx, tag2idx)
3 print(f"Post-tagging de la oración: {predicted_tags}")
```

3 Algoritmos Genéticos

Buenas referencias <https://repository.urosario.edu.co/server/api/core/bitstreams/7ae959ec-81de-435b-aca4-e9bb365e4894/content>

Buena documentación (graficos)

https://www.cs.us.es/~fsancho/Blog/posts/Algoritmos_Geneticos.md.html

<http://www.roboLabo.etsit.upm.es/asignaturas/irin/transparencias/AG.pdf>

<http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf>

<https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Bioinformatica/Tema%2006%20-%20AGs%20I.pdf>

Libros de Algoritmos Genéticos

Individuo o Cromosoma Compuesto de Genes Codificación de genes (Binaria, Entera, Real)

3.1 Introducción

Los **algoritmos genéticos**, o también llamados **algoritmos evolutivos** es un método bastante común en minería de datos. Se inspiran en el **proceso natural de selección y evolución** tal y como se describe por la teoría evolucionista de la selección natural postulada por **Darwin** [1].

Los **principios** sobre los que se asientan los algoritmos genéticos son:

- Los individuos **mejor adaptados** al entorno son aquellos que tienen una probabilidad mayor de **sobrevivir** y, por ende, de **reproducirse**.
- Los descendientes **heredan** características de sus progenitores.
- De forma esporádica y natural se producen **mutaciones** en el material genético de algunos individuos, provocando cambios permanentes.

Los algoritmos genéticos se empezaron a estudiar sobre los años 60 a partir del trabajo de Fogel [2] (donde los organismos eran máquinas de esados finitos), siguiendo con los trabajos de Rechenberg [6] (se establecen estrategias de selección) y principalmente de Holland [3] (se estableció el nombre de **Algoritmos Genéticos**).

Los algoritmos genéticos son adecuados para obtener buenas aproximaciones en **problemas de búsqueda, aprendizaje y optimización** [5].

De forma esquemática un algoritmo genético es una **función matemática** que tomando como entrada unos individuos iniciales (**población origen**) selecciona aquellos **ejemplares** (también llamados individuos o cromosomas) que **recombinándose** por algún método generarán como resultado la **siguiente generación**. Esta función se aplicará de forma **iterativa** hasta verificar alguna condición de parada, bien pueda ser un número máximo de iteraciones o bien la obtención de un individuo que cumpla unas restricciones iniciales.

Condiciones para la aplicación de los Algoritmos Genéticos

No es posible la aplicación en toda clase de problemas de los algoritmos genéticos. Para que estos puedan aplicarse, los problemas deben cumplir las siguientes condiciones:

- El **espacio de búsqueda** [Recordemos que cualquier método de Data Mining se puede asimilar como una búsqueda en el espacio solución, es decir, el espacio formado por todas las posibles soluciones de un problema] debe estar acotado, por tanto ser **finito**.
- Es necesario poseer una **función** de aptitud, que denominaremos **fitness**, que evalúe cada solución (individuo) indicándonos de forma cuantitativa cuán buena o mala es una solución concreta.
- Las **soluciones** deben ser **codificables** en un lenguaje comprensible para un **ordenador**, y si es posible de la forma más **compacta** y abreviada posible.

Habitualmente, la segunda condición es la más complicada de conseguir, para ciertos problemas es trivial la función de fitness (por ejemplo, en el caso de la búsqueda del máximo de una función) no obstante, en la vida real a veces es muy complicada de obtener y, habitualmente, se realizan conjeturas evaluándose los algoritmos con varias funciones de fitness.

Ventajas e inconvenientes

Ventajas

- No necesitan ningún conocimiento particular del problema sobre el que trabajan, únicamente cada ejemplar debe representar una posible solución al problema.
- Es un algoritmo admisible, es decir, con un número de iteraciones suficiente son capaces de obtener la solución óptima en problemas de optimización.
- Los algoritmos genéticos son bastante robustos frente a falsas soluciones ya que al realizar una inspección del espacio solución de forma no lineal (por ejemplo, si quisieramos obtener el máximo absoluto de una función) el algoritmo no recorre la función de forma consecutiva por lo que no se ve afectada por máximos locales.
- Altamente paralelizable, es decir, ya que el cálculo no es lineal podemos utilizar varias máquinas para ejecutar el programa y evaluar así un mayor número de casos.
- Pueden ser incrustables en muchos algoritmos de data mining para formar modelos híbridos. Por ejemplo para seleccionar el número óptimo de neuronas en un modelo de Perceptrón Multicapa.

Inconvenientes

- Su coste computacional puede llegar a ser muy elevado, si el espacio de trabajo es muy grande.
- En el caso de que no se haga un correcto ajuste de los parámetros pueden llegar a caer en una situación de dominación en la que se produce un bucle infinito ya que unos individuos dominan sobre los demás impidiendo la evolución de la población y por tanto inhiben la diversidad biológica.
- Puede llegar a ser muy complicado encontrar una función de evaluación de cada uno de los individuos para seleccionar los mejores de los peores.

3.2 Fundamentos teóricos

A continuación, se explican los conceptos básicos de los algoritmos genéticos.

3.2.1 Codificación de los datos

Cada **individuo o cromosoma** está formado por unos cuantos **genes**. Para nuestro caso vamos a establecer que los individuos tienen un único cromosoma con una cierta cantidad de genes. Estos genes los consideramos como la cantidad mínima de información que se puede transferir. Los genes se pueden agrupar en **características o rasgos** que nos podrían ayudar en la resolución de ciertos problemas.

Estos individuos con sus genes los tenemos que representar de forma que podamos codificar esa información.

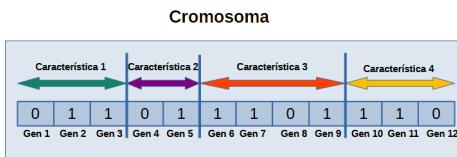


Imagen 3.1: Representación de un cromosoma

Los principales métodos de representación son:

- **Binaria:** Los individuos/cromosomas están representados por una serie de genes que son bits (valores 0 ó 1).
- **Entera:** Los individuos/cromosomas están representados por una serie de genes que son números enteros.
- **Real:** Los individuos/cromosomas están representados por una serie de genes que son números reales en coma flotante.
- **Permutacional:** Los individuos/cromosomas están representados por una serie de genes que son permutaciones de un conjunto de elementos. Se usan en aquellos problemas en los que la secuencia u orden es importante.
- **Basada en árboles:** Los individuos/cromosomas están representados por una serie de genes que son estructuras jerárquicas.

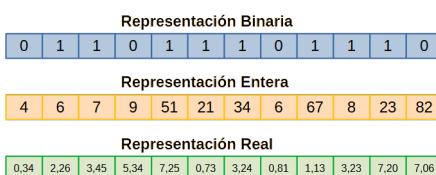


Imagen 3.2: Diferentes representaciones

El primer paso para conseguir que un ordenador procese unos **datos** es conseguir **representarlos** de una forma apropiada. En primer término, para codificar los datos, es necesario separar las posibles configuraciones posibles del dominio del problema en un **conjunto de estados finito**.

Una vez obtenida esta clasificación el objetivo es representar cada **estado** de **forma única** con una cadena (compuesta en la mayoría de casos por unos y ceros).

A pesar de que cada estado puede codificarse con alfabetos de diferente cardinalidad[^{La longitud de las cadenas que representen los posibles estados no es necesario que sea fija, representaciones como la de Kitano para representar operaciones matemáticas son un ejemplo de esto]}, uno de los resultados fundamentales de la teoría de algoritmos genéticos es el **Teorema del Esquema** de Holland [3], que afirma que la codificación

óptima es aquella en la que los algoritmos tienen un alfabeto de cardinalidad, es decir el uso del **alfabeto binario**.

El enunciado del **Teorema del Esquema** es el siguiente: *Esquemas cortos, de bajo orden y aptitud superior al promedio reciben un incremento exponencial de representantes en generaciones subsecuentes de un Algoritmo Genético.*

Una de las ventajas de usar un alfabeto binario para la construcción de configuraciones de estados es la sencillez de los operadores utilizados para la modificación de estas. En el caso de que el alfabeto sea binario, los operadores se denominan, lógicamente, **operadores binarios**. Es importante destacar que variables que estén próximas en el espacio del problema deben preferiblemente estarlo en la codificación ya que la proximidad entre ellas condiciona un elemento determinante en la mutación y reproducibilidad de éstas. Es decir, dos estados que en nuestro espacio de estados del universo del problema que están consecutivos deberían estarlo en la representación de los datos, esto es útil para que cuando haya mutaciones los saltos se den entre estados consecutivos. En términos generales cumplir esta premisa mejora experimentalmente los resultados obtenidos con algoritmos genéticos.

En la práctica el factor que condiciona en mayor grado el fracaso o el **éxito** de la aplicación de algoritmos genéticos a un problema dado es una **codificación acorde** con los **datos**.

Otra opción muy común es establecer a cada uno de los posibles casos un **número natural** y luego codificar ese número en binario natural, de esta forma minimizamos el problema que surge al concatenar múltiples variables independientes en el que su representación binaria diera lugar a numerosos huecos que produjeran soluciones no válidas.

3.2.2 Algoritmo

Un algoritmo genético implementado en **pseudo código** podría ser el siguiente:

Algoritmo 1 Quicksort

```

1: procedure QUICKSORT( $A, p, r$ )
2:   if  $p < r$  then
3:      $q = \text{PARTITION}(A, p, r)$ 
4:     QUICKSORT( $A, p, q - 1$ )
5:     QUICKSORT( $A, q + 1, r$ )
6:   end if
7: end procedure
8: procedure PARTITION( $A, p, r$ )
9:    $i = i + 1$ 
10:  exchange
11: end procedure

```

Un posible diagrama de flujo que puede representar una posible implementación de algoritmos genéticos se muestra en la figura 3.3 .

A continuación, en los siguientes apartados, se hará una descripción de las fases anteriormente expuestas:

Inicializar Población

Como ya se ha explicado antes, el primer paso es inicializar la población origen. Habitualmente la inicialización se hace de forma **aleatoria** procurando una **distribución homogénea** en los casos iniciales de

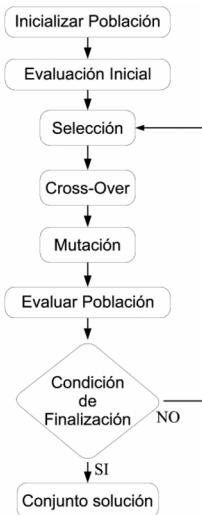


Imagen 3.3: Esquema de implementación de un algoritmo genético

prueba. No obstante, si se tiene un conocimiento más profundo del problema es posible obtener mejores resultados inicializando la población de una forma apropiada a la clase de soluciones que se esperan obtener.

Evaluar Población

Durante cada **iteración** (generación) cada individuo/cromosoma se decodifica convirtiéndose en un grupo de parámetros del problema y se evalúa el problema con esos datos.

Pongamos por **ejemplo** que queremos evaluar el máximo de la función $f(x) = x^2$ en el intervalo $[0, 1]$ y supongamos que construimos cada individuo con **6 dígitos** ($2^6 = 64$), por lo que interpretando el número obtenido en binario natural y dividiéndolo entre 64 obtendremos el punto de la función que corresponde al individuo. Evaluando dicho punto en la función que queremos evaluar ($f(x) = x^2$) obtenemos lo que en nuestro caso sería el **fitness**, en este caso cuanto mayor fitness tenga un individuo, mejor valorado está y más probable es que prospere su descendencia en el futuro. No en todas las implementaciones de algoritmos genéticos se realiza una fase de evaluación de la población tal y como aquí está descrita, en ciertas ocasiones se omite y no se genera ningún fitness asociado a cada estado evaluado. La fase de selección elige los individuos a reproducirse en la próxima generación, esta selección puede realizarse por muy distintos métodos.

En el algoritmo mostrado en pseudo código anteriormente el **método de selección** usado depende del fitness de cada individuo. A continuación, se describen los más comunes:

Selección elitista: Se seleccionan los individuos con mayor fitness de cada generación. La mayoría de los algoritmos genéticos no aplican un elitismo puro, sino que en cada generación evalúan el fitness de cada uno de los individuos, en el caso de que los mejores de la anterior generación sean mejores que los de la actual éstos se copian sin recombinación a la siguiente generación.

Selección proporcional a la aptitud: los individuos más aptos tienen más probabilidad de ser seleccionados, asignándoles una probabilidad de selección más alta. Una vez seleccionadas las probabilidades de selección a cada uno de los individuos se genera una nueva población teniendo en cuenta éstas.

Selección por rueda de ruleta: Es un método conceptualmente similar al anterior. Se le asigna una probabilidad absoluta de aparición de cada individuo de acuerdo al fitness de forma que ocupe un tramo del intervalo total de probabilidad (de 0 a 1) de forma acorde a su fitness. Una vez completado el tramo total se generan números aleatorios de 0 a 1 de forma que se seleccionen los individuos que serán el caldo de cultivo de la siguiente generación.

Selección por torneo: se eligen subgrupos de individuos de la población, y los miembros de cada subgrupo compiten entre ellos. Sólo se elige a un individuo de cada subgrupo para la reproducción.

Selección por rango: a cada individuo de la población se le asigna un rango numérico basado en su fitness, y la selección se basa en este ranking, en lugar de las diferencias absolutas en el fitness. La ventaja de este método es que puede evitar que individuos muy aptos ganen dominancia al principio a expensas de los menos aptos, lo que reduciría la diversidad genética de la población y podría obstaculizar la búsqueda de una solución aceptable. Un ejemplo de esto podría ser que al intentar maximizar una función el algoritmo genético convergiera hacia un máximo local que posee un fitness mucho mejor que el de sus congéneres de población lo que haría que hubiera una dominancia clara con la consecuente desaparición de los individuos menos aptos (con peor fitness).

Selección generacional: la descendencia de los individuos seleccionados en cada generación se convierte en la siguiente generación. No se conservan individuos entre las generaciones.

Selección por estado estacionario: la descendencia de los individuos seleccionados en cada generación vuelve al acervo genético preexistente, reemplazando a algunos de los miembros menos aptos de la siguiente generación. Se conservan algunos individuos entre generaciones.

Búsqueda del estado estacionario: Ordenamos todos los genes por su fitness en orden decreciente y eliminamos los últimos m genes, que se sustituyen por otros m descendientes de los demás. Este método tiende a estabilizarse y converger.

Selección jerárquica: los individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método es que reduce el tiempo total de cálculo al utilizar una evaluación más rápida y menos selectiva para eliminar a la mayoría de los individuos que se muestran poco o nada prometedores, y sometiendo a una evaluación de aptitud más rigurosa y computacionalmente más costosa sólo a los que sobreviven a esta prueba inicial.

Recombinación.

Recombinación también llamada **Cross-over o reproducción**. La recombinación es el operador genético más utilizado y consiste en el **intercambio de material genético** entre **dos individuos** al azar (pueden ser incluso entre el mismo elemento). El material genético se intercambia entre **bloques**. Gracias a la presión selectiva[[↑] Presión Selectiva es la fuerza a la que se ven sometido naturalmente los genes con el paso del tiempo. Con el sucesivo paso de las generaciones los genes menos útiles estarán sometidos a una mayor presión selectiva produciéndose la paulatina desaparición de estos] irán predominando los mejores bloques génicos.

Existen diversos **tipos** de **cross-over**:

Cross-over de 1 punto. Los cromosomas se cortan por 1 punto y se intercambian los dos bloques de genes.

Cross-over de n-puntos. Los cromosomas se cortan por n puntos y el resultado se intercambia.

Cross-over uniforme. Se genera un patrón aleatorio en binario, y en los elementos que haya un 1 se realiza intercambio genético.

Cross-over especializados. En ocasiones, el espacio de soluciones no es continuo y hay soluciones que a pesar de que sean factibles de producirse en el gen no lo son en la realidad, por lo que hay que incluir restricciones al realizar la recombinación que impidan la aparición de algunas combinaciones.

Mutación.

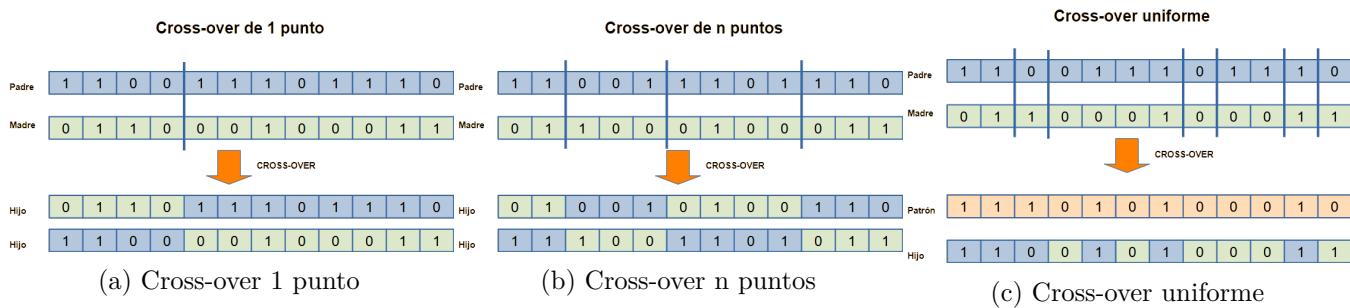


Imagen 3.4: Cross-Over

Este fenómeno, generalmente muy raro en la naturaleza, se modela de la siguiente forma: cuando se genera un hijo se examinan uno a uno los genes del mismo y se genera un coeficiente aleatorio para cada uno. En el caso de que algún coeficiente supere un cierto umbral se modifica dicho gen. Modificando el umbral podemos variar la probabilidad de la mutación. Las mutaciones son un mecanismo muy interesante por el cual es posible generar nuevos individuos con rasgos distintos a sus predecesores.

Los **tipos de mutación** más conocidos son:

- **Mutación de gen:** existe una única probabilidad de que se produzca una mutación de algún gen. De producirse, el algoritmo toma aleatoriamente un gen, y lo invierte.
- **Mutación multigen:** cada gen tiene una probabilidad de mutarse o no, que es calculada en cada pasada del operador de mutación multigen.
- **Mutación de intercambio:** Se intercambia el contenido de dos genes aleatoriamente.
- **Mutación de barajado:** existe una probabilidad de que se produzca una mutación. De producirse, toma dos genes aleatoriamente y baraja de forma aleatoria los genes, según hubiéramos escogido, comprendidos entre los dos.

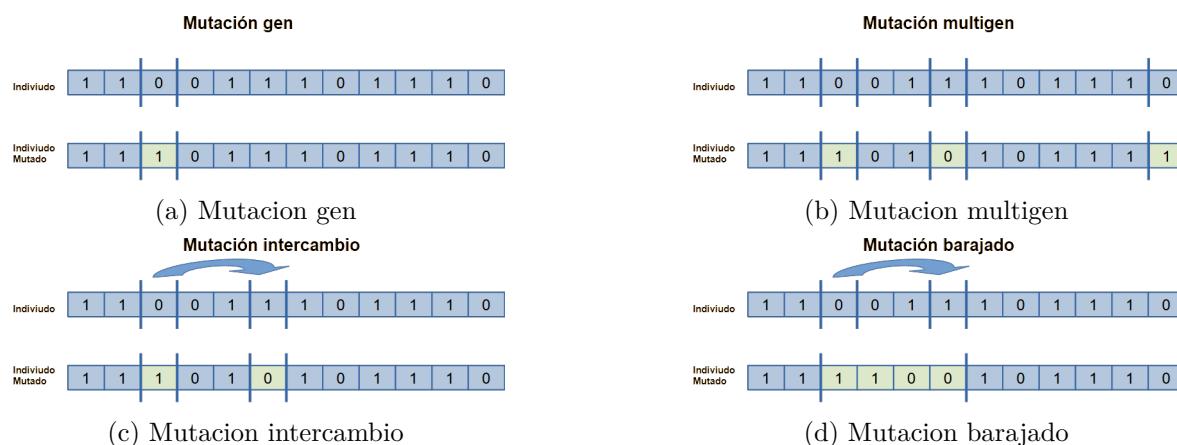


Imagen 3.5: Mutación

Estos ejemplos de **mutaciones** han sido ilustradas usando una representación binaria de los datos, en caso de tener una representación **entera** al hacer la mutación no cambiaríamos de 0 a 1 o viceversa, sino que se elegiría al azar un entero de los posibles valores que tenemos para ese gen. En el caso de una representación **real** se podría pensar en al mutación de un gen como la selección de un número real entre unos valores dados mediante una **distribución uniforme** o incluso una **distribución normal**. Para profundizar en operadores sobre distintos tipos de representaciones puedes consultar [@ Eiben2015]

Condición de finalización

Una vez que se ha generado la nueva población se evalúa la misma y se selecciona a aquel individuo o aquellos que por su fitness se consideran los más aptos. Podemos tener definido un umbral del valor de fitness que queremos alcanzar o simplemente definir el número de iteraciones que queremos que se realicen.

3.2.3 Otros Operadores

Los operadores descritos anteriormente suelen ser operadores **generalistas** (aplicables y de hecho aplicados a todo tipo de problemas), sin embargo, para ciertos contextos suele ser más recomendable el uso de operadores específicos para realizar un recorrido por el espacio de solución más acorde a la solución buscada.

Modificadores de la longitud de los individuos. En ocasiones las soluciones no son una combinación de todas las variables de entrada, en estas ocasiones los individuos deberán tener una longitud variable[En muchas ocasiones, se realizan estudios de minería de datos sobre todos los datos existentes, encontrándose en ellos variables espúreas, es decir, variables que no aportan nada de información para el problema evaluado]. Lógicamente, en este tipo de casos, es necesario modificar la longitud de los individuos, para ello haremos uso de los operadores añadir y quitar, que añadirán o quitarán a un individuo un trozo de su carga génica (es decir, un trozo de información).

3.2.4 Parámetros necesarios al aplicar Algoritmos Genéticos

Cualquier algoritmo genético necesita ciertos parámetros que deben fijarse antes de cada ejecución, como:

Tamaño de la población: Determina el tamaño máximo de la población a obtener. En la práctica debe ser de un valor lo suficientemente grande para permitir diversidad de soluciones e intentar llegar a una buena solución, pero siendo un número que sea computable en un tiempo razonable.

Condición de terminación: Es la condición de parada del algoritmo. Habitualmente es la convergencia de la solución (si es que la hay), un número prefijado de generaciones o una aproximación a la solución con un cierto margen de error.

Individuos que intervienen en la reproducción de cada generación: se especifica el porcentaje de individuos de la población total que formarán parte del acervo de padres de la siguiente generación. Esta proporción es denominada proporción de cruces.

Probabilidad de ocurrencia de una mutación: En toda ejecución de un algoritmo genético hay que decidir con qué frecuencia se va a aplicar la mutación. Se debe de añadir algún parámetro adicional que indique con qué frecuencia se va a aplicar dentro de cada gen del cromosoma. La frecuencia de aplicación de cada operador estará en función del problema; teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suelen aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.

Cada implementación de algoritmo tendrá sus propios parámetros que permitirán personalizar la ejecución de nuestro problema concreto.

! Recordad

Los algoritmos genéticos es uno de los **enfoques más originales** en data mining. Su sencillez, combinada con su flexibilidad les proporciona una **robustez** que les hace adecuados a infinidad de problemas. No obstante, su **simplicidad** y sobre todo independencia del problema hace que sean algoritmos poco

específicos. Recorriendo este capítulo hemos visto los numerosos parámetros y métodos aplicables a los algoritmos genéticos que nos ayudan a realizar una adaptación de los algoritmos genéticos más concreta a un problema. En definitiva, la implementación de esquemas evolutivos tal y como se describen en biología podemos afirmar que funciona.

3.3 Casos de uso

Para los **Algoritmos genéticos** tenemos 3 grandes grupos de casos de uso:

- **Optimización de Funciones**
 - Podemos buscar máximo o mínimos de funciones.
- **Optimización Cominatorial**
 - TSP (Travel Salesman Problem) Problema del viajante
 - VRP (Vehicle Routing Problem) Problema de rutas de vehículos
- **Optimización Machine Learning**
 - Hiperparámetros
 - Selección de variables
 - Network Architecture

Vamos a ver cómo se podrían abordar algunos de estos casos de uso:

3.3.1 Selección de Variable

El objetivo del ejemplo es ver cómo podemos usar un algoritmo genético para hacer una **selección de variables**, quedándonos sólo con unas pocas.

Supongamos que tenemos un **dataset** que es un problema de **clasificación** con **3 clases**, cuenta con **1500 muestras** y **14 variables** explicativas.

Tendremos que, para el algoritmo genético, nuestro cromosoma o individuo será un **vector** de tamaño 14 (**14 genes**), que representa las **14 variables** del dataset que hemos preparado.

En la imagen Figure 3.6 mostramos la población de 100 individuos en una iteración N.

Función Fitness (Evaluación)

Cuando estamos trabajando con selección de variables, el objetivo es conseguir el conjunto de variables que mejor modelo construyan según nuestro dataset. En este caso, al ser un problema de **clasificación**, veremos cuál es la combinación de variables que nos da **menos errores al clasificar**.

Nuestra función fitness deberá seguir estos **pasos**:

- Recibe una **variable** que tiene el tamaño del número de variables (el tamaño del cromosoma) que hay en el dataframe (en nuestro caso 14) de datos.
- Los valores son **1** si esa variable se va a **usar** y **0** si **no** se va a **usar**.

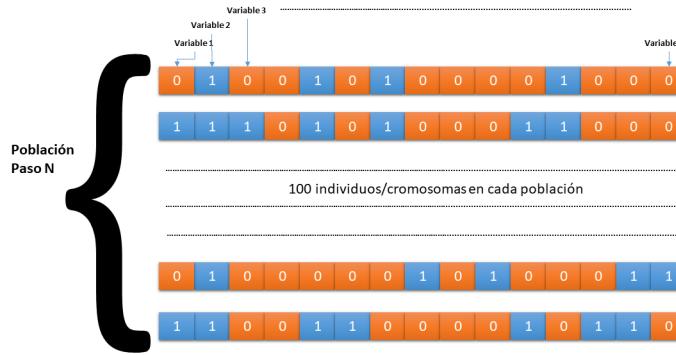


Imagen 3.6: Población en iteración N

- Se construye un **modelo**, en este caso usamos **LDA** (Análisis Discriminante Lineal) con las **variables** que tienen **valor 1**.
- Calculamos el **error** que queremos minimizar (número de fallos)
- Para este caso del LDA cogemos los valores **\$posterior** que nos dan la probabilidad de cada clase para cada entrada de la muestra
- Calculamos cual es el **máximo** y así le asignamos esa **clase** como su solución. También podríamos coger directamente el valor de **\$class** con la clase dada como predicción.
- Verificamos cuantos hemos fallado y lo dividimos por el número de muestras para ver el **porcentaje de fallos**
- Devolvemos el **porcentaje de fallos**. El resultado de la ejecución del algoritmo evolutivo nos dará un objeto del que tendremos que obtener que variables son las que queremos usar.

Figure 3.7

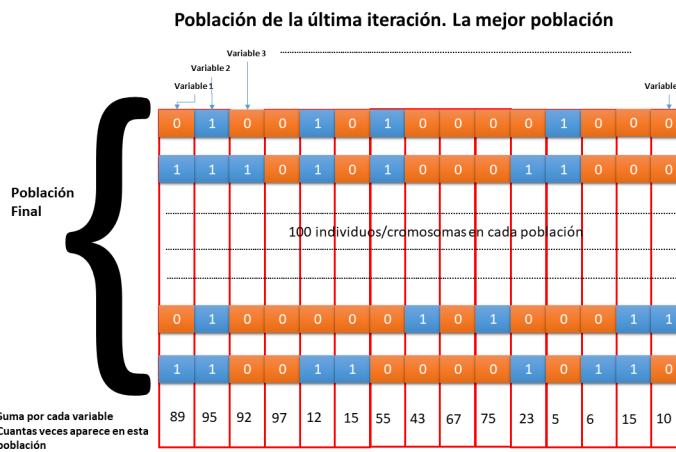


Imagen 3.7: Población Final

Una vez que nuestro algoritmo pare, deberíamos tener la población que mejor se ha adaptado según el fitness que habíamos definido.

En nuestro caso estarán por ejemplo las 100 mejores combinaciones de variables, que dan el menor error al clasificar. De esta manera si para cada variable contamos cuantas veces ha salido en cada elemento de la población, sabremos cuantas veces se ha usado en las combinaciones de variables de esta última iteración (que es la mejor hasta ese momento). Con lo cual podremos saber cuales han sido las **variables más usadas** en la población final.

El objeto **modelo_evolutivo**, que nos devuelve rbga.bin, tiene una variable **population** de dimensiones tamaño_población x numero_variables , en nuestro caso de 100x14, que tiene la información de la población de la **última iteración del algoritmo**, que en un principio debería ser la mejor. Este population contiene para cada fila (elemento en la población), 0 o 1 en la posición que corresponde a cada variable.

Para ver cuales son las variables que más se han usado tenemos que sumar por columnas y ese dato nos dará para cada columna (corresponde con una variable) la cantidad de veces que se ha usado en esta población. Una vez tenemos estos datos ya podemos quedarnos con el número de variables que deseemos cogiendo las que más alto valor tienen.

Figure 3.8

Imagen 3.8

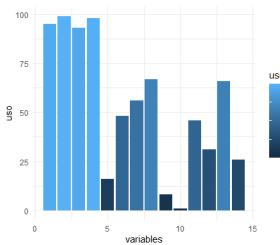


Imagen 3.8: Frecuencia de las Variables

3.3.2 Entrenamiento de Red Neuronal

Otro de los casos de uso sobre los que se podría trabajar con un **Algoritmo Genético** es el entrenamiento de una Red Neuronal. La forma estandard de entrenar una Red Neuronal es lo que se denomina el Back-propagation, que mediante el uso de las estrategias del **Descenso del Gradiente** se consigue optimizar los parámetros de la Red Neuronal.

Cuando entrenamos una Red Neuronal, lo que conseguimos es obtener una serie de valores de los **pesos** de la Red Neuronal. Estos pesos combinados con las funciones de activación serán los que nos darán el resultado de salida a partir de los datos de entrada.

Supongamos que tenemos una **Red Neuronal** con **500** parámetros distribuidos en las diferentes capas ocultas del mismo.

Tendremos que, para el algoritmo genético, nuestro cromosoma o individuo será un **vector** de tamaño 500 (**500 genes**), que representan los **500 valores** de los pesos de la Red Neuronal. En este caso los valores del vector tendrán una **representación real**.

A cada uno de estos posibles valores reales lo deberemos acotar en un rango de valores, que podría ser *(-2.0 , 2.0) de forma que cuando se generen los valores aleatorios de una población, cada dato deberá partir de este rango.

En la imagen Figure 3.6 mostramos la población de 100 individuos en una iteración N.

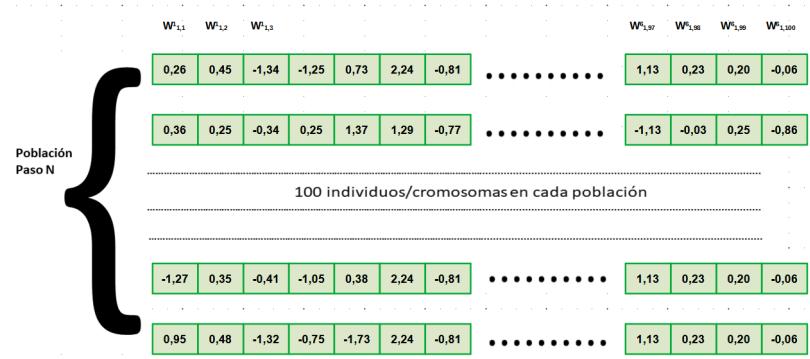


Imagen 3.9: Población en iteración N

Función Fitness (Evaluación)

Cuando estamos trabajando con el Entrenamiento de una Red Neuronal, el objetivo es conseguir la red neuronal que mejor valor de la **función de pérdida** de la red neuronal tenga.

Nuestra función fitness deberá seguir estos **pasos**:

- Recibe una **variable** que tiene el tamaño del numero de pesos en la red neuronal (el tamaño del cromosoma), en nuestro caso 500.
 - Los valores serán un número real
- Se cogen estos pesos y se asignan a la Red Neuronal.
 - Una vez tenemos la red neuronal, pasamos nuestros datos de entrenamiento y evaluamos el modelo obteniendo el valor de la función de pérdida.
- Devolvemos el **valor de la función de pérdida**.

Figure 3.7



Imagen 3.10: Población Final

Una vez que nuestro algoritmo pare, deberíamos tener la población que mejor se ha adaptado según el fitness que habíamos definido.

Seleccionamos aquella que menor valor de función de pérdida nos haya dado.

3.3.3 Arquitectura de RedNeuronal

Siguiendo con las redes neuronales, otro de los casos de uso sobre los que se podría trabajar con un **Algoritmo Genético** es la **Arquitectura de Red** de la Red Neuronal. En el sentido de poder definir cuantos nodos por cada capa oculta podemos tener, para conseguir una buena Red Neuronal. En ese caso estamos pensando en una red neuronal usada para datasets clásicos de datos y no de imágenes.

Cuando entrenamos una Red Neuronal, lo que conseguimos es obtener una serie de valores de los **pesos** de la Red Neuronal. Estos pesos combinados con las funciones de activación serán los que nos darán el resultado de salida a partir de los datos de entrada.

Supongamos que tenemos una **Red Neuronal** con **500** parámetros distribuidos en las diferentes capas ocultas del mismo.

Tendremos que, para el algoritmo genético, nuestro cromosoma o individuo será un **vector** de tamaño 500 (**500 genes**), que representan los **500 valores** de los pesos de la Red Neuronal. En este caso los valores del vector tendrán una **representación real**.

A cada uno de estos posibles valores reales lo deberemos acotar en un rango de valores, que podría ser *(-2.0 , 2.0) de forma que cuando se generen los valores aleatorios de una población, cada dato deberá partir de este rango.

En la imagen Figure 3.6 mostramos la población de 100 individuos en una iteración N.



Imagen 3.11: Población en iteración N

Función Fitness (Evaluación)

Cuando estamos trabajando con el Entrenamiento de una Red Neuronal, el objetivo es conseguir la red neuronal que mejor valor de la **función de pérdida** de la red neuronal tenga.

Nuestra función fitness deberá seguir estos **pasos**:

- Recibe una **variable** que tiene el tamaño del numero de pesos en la red neuronal (el tamaño del cromosoma), en nuestro caso 500.
 - Los valores serán un número real
- Se cogen estos pesos y se asignan a la Red Neuronal.
 - Una vez tenemos la red neuronal, pasamos nuestros datos de entrenamiento y evaluamos el modelo obteniendo el valor de la función de pérdida.

- Devolvemos el valor de la función de pérdida.

Figure 3.7



Imagen 3.12: Población Final

Una vez que nuestro algoritmo pare, deberíamos tener la población que mejor se ha adaptado según el fitness que habíamos definido.

Seleccionamos aquella que menor valor de función de pérdida nos haya dado.

3.4 Algoritmos genéticos con R

3.4.1 Paquetes para usar en R

3.4.2 Selección de variables

3.4.3 Entrenamiento de Red Neuronal

3.5 Algoritmos genéticos en Python

3.5.1 Paquetes para usar en Python

3.5.2 Optimización de funciones

3.5.3 Entrenamiento de Red Neuronal

```

1 import os import pandas as pd import numpy as np import matplotlib.pyplot as plt import
   seaborn as sns
2
3 from genetic_selection import GeneticSelectionCV
4
5 from sklearn.preprocessing import StandardScaler from sklearn.model_selection import
   train_test_split from sklearn.metrics import confusion_matrix

```

Cargar datos de trabajo

```

1 os.chdir('C:/Users/p_san/Desktop/Máster_2020/Módulo_5') #directorío
2   ↓  datos=pd.read_csv('german_credit.csv',encoding = 'ISO-8859-1', index_col=None)

```

Todas las variables son categóricas salvo:

duration

credit_amount

residence_since

age

existing_credits

num_dependents

Conversión a variables categóricas

```

1 datos\['checking_status'\]=datos\['checking_status'\].astype('category')
2   ↓  datos\['credit_history'\]=datos\['credit_history'\].astype('category')
3   ↓  datos\['purpose'\]=datos\['purpose'\].astype('category')
4   ↓  datos\['savings_status'\]=datos\['savings_status'\].astype('category')
5   ↓  datos\['employment'\]=datos\['employment'\].astype('category')
6   ↓  datos\['personal_status'\]=datos\['personal_status'\].astype('category')
7   ↓  datos\['other_parties'\]=datos\['other_parties'\].astype('category')
8   ↓  datos\['property_magnitude'\]=datos\['property_magnitude'\].astype('category')
9   ↓  datos\['other_payment_plans'\]=datos\['other_payment_plans'\].astype('category')
10  ↓  datos\['housing'\]=datos\['housing'\].astype('category')
11  ↓  datos\['job'\]=datos\['job'\].astype('category')
12  ↓  datos\['property_magnitude'\]=datos\['property_magnitude'\].astype('category')
13  ↓  datos\['own_telephone'\]=datos\['own_telephone'\].astype('category')
14  ↓  datos\['foreign_worker'\]=datos\['foreign_worker'\].astype('category')
15  ↓  datos\['class'\]=datos\['class'\].astype('category')

```

La variable class es una variable reservada en diferentes módulos de Python -> reemplazar por target

```

1 datos.rename(columns={'class': 'target'}, inplace=True)
2   ↓  datos\['target'\]=np.where(datos\['target'\]=='good', 0, 1) # cambio en la
3   ↓  codificación por sencillez en el preprocessado

```

Definición de la muestra de trabajo

```

1 datos_entrada=datos.drop('target', axis=1) # Datos de entrada datos_entrada=
2   ↓  pd.get_dummies(datos_entrada, drop_first=True) #conversión a variables dummy

```

datos de salida

```
1 respuesta=datos.loc[:, 'target']\n
```

Escalado de las variables, partición de la muestra y Cross Validation

```
1 seed=123 \# Escalado de los datos de entrada\n  ↳ x_esc=StandardScaler().fit_transform(datos_entrada) x_esc=pd.DataFrame(x_esc,\n  ↳ columns=datos_entrada.columns)
```

Partición de la muestra

```
test_size=0.3 #muestra para el test x_train, x_test, y_train, y_test = train_test_split(x_esc,respuesta,\n test_size=test_size, random_state=seed, stratify=respuesta) Usando un modelo Cart from sklearn.tree im-\n port DecisionTreeClassifier cart=DecisionTreeClassifier(max_depth=5, random_state=seed) cart_algoritmo_gen=\n cv=5, # 5 particiones verbose=0, # no se muestran los resultados en la pantalla scoring="roc_auc", #\n ejemplo métrica para evaluar max_features=15, # número de variables máximas en la selección de\n # características n_population=50, # tamaño de la población crossover_proba=0.5, # probabilidad\n de cruce entre parejas de genes mutation_proba=0.2, #probabilidad de mutación n_generations=40,\n #número de generaciones crossover_independent_proba=0.5, # prob. cruce para genes # independientes\n mutation_independent_proba=0.05, # prob. mutación de genes # independientes tournament_size=3,\n #tamaño de los grupos n_gen_no_change=10, # genes que se mantienen -> no pasan a # la segunda\n generación caching=True, n_jobs=1)
```

```
cart_algoritmo_gen=cart_algoritmo_gen.fit(x_train, y_train)\najuste del modelo usando algoritmo genéticos para la selección de variables # Variables Seleccionadas\nprint('Num. Var:', cart_algoritmo_gen.n_features_) # print(cart_algoritmo_gen.support_)\n# Resultados matriz numpy -> mala visualización. Los resultados se # convierten a df de pan-\ndas df=pd.DataFrame(cart_algoritmo_gen.support_, columns=['Variables'], index=x_train.columns)\n df=df.loc[~df['Variables'].isin([False])] #se elimina del df las variables no seleccionadas por el algoritmo\nlist(df.index) # variables seleccionadas por el algoritmo
```

Num. Var: 9 ['checking_status_0<=X<200', 'checking_status_<0', 'credit_history_‘critical/other existing credit’', “purpose_‘used car’”, ‘savings_status_>=1000’, “personal_status_‘male single’”, ‘other_parties_guarantor’, ‘other_payment_plans_stores’, ‘job_skilled’]

Resultados test - predicción & Matriz de confusión (modelo CART con selección de variables a través de Algoritmos Genéticos)

```
pred=cart_algoritmo_gen.predict(x_test)\nconfusion_matrix(y_test, pred) # Matriz de confusión\narray([[173, 37], [ 46, 44]], dtype=int64)
```

4 Lógica Difusa

4.1 Contexto y conceptos clave

La lógica difusa tiene un origen que se remonta a la antigua Grecia con Aristóteles, quien ya proponía la idea de grados de verdad o falsedad. Sin embargo, su desarrollo significativo ocurrió en el siglo XVIII, con filósofos como David Hume y Charles Sander Pierce, quienes exploraron conceptos como el razonamiento basado en la experiencia y la idea de vaguedad en lugar de la dicotomía cierto-falso.

El verdadero punto de inflexión llegó en 1962 con Lotfi Zadeh, quien cuestionó la rigidez de las matemáticas tradicionales frente a la imprecisión y las verdades parciales. Su trabajo en la Universidad de California en Berkeley condujo al concepto de conjuntos difusos, que generalizan los conjuntos tradicionales para trabajar con expresiones imprecisas, una noción inspirada en una discusión sobre la belleza de sus respectivas esposas.

Desde entonces, la lógica difusa ha encontrado aplicaciones prácticas en campos como el control de sistemas, la ingeniería y la inteligencia artificial. Ebrahim Mandani y Lauritz Peter Holmblad desarrollaron sistemas de control difuso prácticos en los años 70, mientras que en Japón se produjo un rápido avance en el uso de la lógica difusa en una amplia gama de aplicaciones, desde sistemas de transporte hasta electrodomésticos inteligentes.

La importancia de la lógica difusa radica en su capacidad para modelar la incertidumbre y la imprecisión en la toma de decisiones, permitiendo sistemas más adaptables y eficientes en numerosas aplicaciones. Se fundamenta en los conjuntos difusos y un sistema de inferencia basado en reglas, lo que ofrece una manera elegante de abordar problemas con información vaga o incompleta. En contraste con la lógica tradicional, que emplea conceptos absolutos, la lógica difusa permite grados variables de pertenencia a los conjuntos, imitando así el razonamiento humano.

Se definen a continuación los principales conceptos:

- **Lógica Difusa:** es un sistema matemático que modela funciones no lineales, que convierte unas entradas en salidas acorde con los planteamientos lógicos que usan el razonamiento aproximado.
- **Lógica Difusa en Inteligencia Artificial:** método de razonamiento de máquina similar al pensamiento humano, que puede procesar información incompleta o incierta, característico de muchos sistemas expertos. Con la lógica difusa o borrosa se puede gobernar un sistema por medio de reglas de “sentido común” las cuales se refieren a cantidades indefinidas. En general, la lógica difusa se puede aplicar tanto a sistemas de control como para modelar cualquier sistema continuo de ingeniería, física, biología o economía.
- **Conjuntos difusos:** son conjuntos que permiten la representación de la imprecisión y la incertidumbre. A diferencia de los conjuntos tradicionales, donde un elemento pertenece o no pertenece al conjunto de manera precisa, en los conjuntos difusos un elemento puede pertenecer al conjunto en diferentes grados. Por ejemplo, en lugar de definir un conjunto “alto” como una altura mayor a 180 cm, se podría definir de manera difusa, permitiendo grados de “altitud” para alturas que no son completamente altas o completamente bajas.

- **Sistema de Inferencia Difuso:** es el componente de la lógica difusa que utiliza reglas lingüísticas para mapear las entradas difusas a salidas difusas. Estas reglas se expresan en la forma “SI...ENTONCES...”, donde se especifica cómo se relacionan las variables de entrada con las de salida. Por ejemplo, una regla en un sistema de control de climatización podría ser: “SI la temperatura es FRÍA y la humedad es ALTA, ENTONCES aumentar la calefacción”.
- **Fuzzificación:** es el proceso de convertir entradas nítidas o precisas en valores difusos. Por ejemplo, en un sistema de control de velocidad de un automóvil, la entrada “velocidad del vehículo” puede ser fuzzificada para representar grados de “lento”, “medio” o “rápido”, en lugar de valores exactos de velocidad.
- **Defuzzificación:** es el proceso inverso de fuzzificación, donde se convierten las salidas difusas en valores nítidos o precisos. Después de que el sistema de inferencia difuso ha producido una salida difusa basada en las reglas y las entradas, la defuzzificación asigna un valor preciso a esa salida difusa.
- **Funciones de Pertenencia:** son funciones matemáticas que describen la membresía de un elemento a un conjunto difuso. Estas funciones determinan cómo se distribuyen los grados de pertenencia dentro del conjunto difuso. Algunas funciones comunes son las funciones triangulares, trapezoidales y gaussianas, que modelan diferentes formas de incertidumbre.
- **Variable lingüística:** es una variable cuyos valores se expresan en términos lingüísticos o en palabras en lugar de valores numéricos precisos. Estos términos lingüísticos se utilizan para describir la incertidumbre o la imprecisión asociada con la variable y pueden incluir etiquetas como “bajo”, “medio” y “alto” en lugar de valores numéricos específicos. Al utilizar términos lingüísticos en lugar de valores numéricos precisos, los sistemas difusos pueden manejar de manera más efectiva la información subjetiva y no lineal existente en el mundo real.

4.1.1 Operaciones sobre conjuntos difusos

Las operaciones sobre conjuntos difusos son operaciones matemáticas que se realizan sobre conjuntos difusos para obtener nuevos conjuntos difusos. Las operaciones sobre conjuntos difusos se pueden clasificar en dos categorías principales:

- **Operaciones Unarias:** se aplican a un solo conjunto difuso y generan un nuevo conjunto difuso como resultado. Algunas de las operaciones unarias más comunes son:
 - **Negación:** la negación de un conjunto difuso A , denotada como $\neg A$, se obtiene complementando la función de pertenencia de A . En otras palabras, para todo elemento x en el universo de discurso, $\mu_{\neg A}(x) = 1 - \mu_A(x)$.
 - **Inversión:** la inversión de un conjunto difuso A se obtiene invirtiendo la función de pertenencia de A . En otras palabras, para todo elemento x en el universo de discurso, $\mu_{A^x} = [1 - \mu_A(x)]^n$, donde n es un parámetro que controla la forma de la inversión.
 - **Concentración:** la concentración de un conjunto difuso A , denotada como $C(A)$, se obtiene estrechando la función de pertenencia de A alrededor de sus valores máximos. En otras palabras, para todo elemento x en el universo de discurso, $\mu_{C(A)}(x) = f(\mu_A(x))$, donde f es una función creciente que acerca los valores de $\mu_A(x)$ a 1.
 - **Dilatación:** La dilatación de un conjunto difuso A , denotada como $D(A)$, se obtiene ampliando la función de pertenencia de A . En otras palabras, para todo elemento x en el universo de discurso, $\mu_{D(A)}(x) = g(\mu_A(x))$, donde g es una función decreciente que aleja los valores de $\mu_A(x)$ de 1.

- **Operaciones Binarias:** se aplican a dos conjuntos difusos y generan un nuevo conjunto difuso como resultado. Algunas de las operaciones binarias más comunes son:
 - **Intersección:** la intersección de dos conjuntos difusos A y B , denotada como $A \cap B$, se obtiene como el conjunto difuso que contiene solo los elementos que pertenecen tanto a A como a B . En otras palabras, para todo elemento x en el universo de discurso, $\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$.
 - **Unión:** La unión de dos conjuntos difusos A y B , denotada como $A \cup B$, se obtiene como el conjunto difuso que contiene todos los elementos que pertenecen a A , a B o a ambos. En otras palabras, para todo elemento x en el universo de discurso, $\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$.
 - **Complemento:** El complemento de un conjunto difuso A , denotado como A^C , se obtiene como el conjunto difuso que contiene todos los elementos que no pertenecen a A . En otras palabras, para todo elemento x en el universo de discurso, $\mu_{A^C}(x) = 1 - \mu_A(x)$.
 - **Composición:** la composición de dos conjuntos difusos A y B , denotada como $A \circ B$, se obtiene como el conjunto difuso que representa la “relación” entre A y B . La definición precisa de la composición depende del tipo de conjuntos difusos y la aplicación específica.

Propiedades de las Operaciones sobre Conjuntos Difusos

Las operaciones sobre conjuntos difusos generalmente satisfacen ciertas propiedades matemáticas que garantizan su consistencia y aplicabilidad. Algunas de las propiedades comunes son:

- **Asociatividad:** la asociación de las operaciones define el orden en que se realizan las operaciones. Por ejemplo, para la intersección: $(A \cap B) \cap C = A \cap (B \cap C)$
- **Commutatividad:** el orden de los operandos no afecta el resultado de la operación. Por ejemplo, para la unión: $A \cup B = B \cup A$
- **Distributividad:** la distribución de una operación sobre otra define cómo se combinan las operaciones. Por ejemplo, para la distribución de la unión sobre la intersección: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- **Absorción:** la absorción de una operación por otra define cómo un elemento neutro afecta el resultado

4.1.1.1 T-normas y T-conormas

Las **T-normas** y **T-conormas** son operadores fundamentales que permiten combinar información difusa y obtener un resultado difuso. Estas operaciones juegan un papel esencial en la intersección y unión de conjuntos difusos.

- **T-normas:** es una función binaria que combina dos valores difusos en un nuevo valor difuso. Se representa como $T(a, b)$, donde a y b son valores difusos entre 0 y 1. Las T-normas se caracterizan por las siguientes propiedades:
 - Monotonidad: $T(a_1, b_1) \leq T(a_2, b_2)$ si $a_1 \leq a_2$ y $b_1 \leq b_2$.
 - Commutatividad: $T(a, b) = T(b, a)$.
 - Asociatividad: $T(T(a_1, a_2), b) = T(a_1, T(a_2, b))$.
 - Neutralidad: $T(a, 1) = a$ para todo a entre 0 y 1.

Las *T-normas* más destacadas son:

- **Producto T-norma:** $T(a, b) = a * b$. Es la T-norma más común y representa la “intersección” difusa
- **Mínima T-norma:** $T(a, b) = \min(a, b)$. Representa la “mínima” entre dos valores difusos
- **Diferencia limitada (o de Lukasiewick):** $T(a, b) = \max(0, a + b - 1)$.

- **T-conormas:** es una función binaria que combina dos valores difusos en un nuevo valor difuso. Se representa como $C(a, b)$, donde a y b son valores difusos entre 0 y 1. Las T-conormas se caracterizan por las siguientes propiedades:

- Monotonidad: $C(a_1, b_1) \geq C(a_2, b_2)$ si $a_1 \geq a_2$ y $b_1 \geq b_2$.
- Comutatividad: $C(a, b) = C(b, a)$.
- Asociatividad: $C(C(a_1, a_2), b) = C(a_1, C(a_2, b))$.
- Neutralidad: $C(a, 0) = a$ para todo a entre 0 y 1.

Las *T-conormas* más destacadas son: - **Producto T-conorma:** $C(a, b) = a + b - a * b$. Es la T-conorma más común y representa la “unión” difusa - **Máxima T-norma:** $T(a, b) = \max(a, b)$. Representa la “máxima” entre dos valores difusos - **Suma limitada (o de Lukasiewick):** $T(a, b) = \min(a + b, 1)$.

A continuación diferentes ejemplos para facilitar la interpretación de estos conceptos:

Producto T-norma ($a = 0.7, b = 0.5$)

Considera un sistema que evalúa la idoneidad de un candidato para un trabajo basado en dos criterios: habilidades técnicas y habilidades de comunicación. Cada criterio se evalúa en una escala de 0 a 1, donde 0 representa ninguna habilidad y 1 representa habilidades excepcionales. Se tienen así los siguientes valores para cada criterio:

- $a = 0.7$ representa la calificación de habilidades técnicas del candidato (70% sobre 100%)
- $b = 0.5$ representa la calificación de habilidades de comunicación del candidato (50% sobre 100%)

Interpretación: aplicando la Producto T-norma ($T(0.7, 0.5) = 0.35$) indica que la idoneidad general del candidato, considerando tanto habilidades técnicas como de comunicación, es del 35%. Esto sugiere que la idoneidad general del candidato es menor que sus calificaciones individuales de habilidades, reflejando la naturaleza “interseptada” de la T-norma.

Mínimo T-norma ($a = 0.8, b = 0.3$)

Imagina un sistema que evalúa el riesgo de una inversión financiera basado en dos factores: volatilidad del mercado y estabilidad económica. Cada factor se evalúa en una escala de 0 a 1, donde 0 representa ningún riesgo y 1 representa alto riesgo. Se tienen así los siguientes valores para cada criterio:

- $a = 0.8$ representa la calificación de volatilidad del mercado (80% sobre 100%)
- $b = 0.3$ representa la calificación de estabilidad económica (30% sobre 100%)

Interpretación: aplicando la Mínimo T-norma ($T(0.8, 0.3) = 0.3$) indica que el riesgo general de la inversión, considerando tanto factores de mercado como económicos, es del 30%. Esto sugiere que el riesgo general está determinado por el factor de calificación más baja (estabilidad económica), reflejando la naturaleza “mínima” de la T-norma.

Producto T-Conorma ($a = 0.4, b = 0.7$)

Considera un sistema que evalúa el éxito potencial de una campaña de marketing basado en dos factores: conocimiento de marca y atractivo del producto. Cada factor se evalúa en una escala de 0 a 1, donde 0 representa ningún potencial y 1 representa alto potencial. Se tienen así los siguientes valores para cada criterio:

- $a = 0.4$ representa la calificación de conocimiento de marca (40% sobre 100%)

- $b = 0.7$ representa la calificación de atractivo del producto (70% sobre 100%)

Interpretación: aplicando el Producto T-conorma ($C(0.4, 0.7) = 0.82$) indica que el éxito potencial general de la campaña, considerando tanto factores de marca como de producto, es del 82%. Esto sugiere que el potencial general es mayor que las calificaciones individuales, reflejando la naturaleza de “unión” de la T-conorma

Máximo T-conorma ($a = 0.6, b = 0.8$)

Imagina un sistema que evalúa la efectividad de un tratamiento médico experimental basado en dos criterios: reducción del dolor y mejora de la movilidad. Cada criterio se evalúa en una escala de 0 a 1, donde 0 representa ninguna mejora y 1 representa una mejora significativa. Se tienen así los siguientes valores para cada criterio:

- $a = 0.6$ representa la reducción del dolor obtenida con el tratamiento (60% de mejora)
- $b = 0.8$ representa la mejora de la movilidad lograda con el tratamiento (80% de mejora)

Interpretación: aplicando el máximo T-conorma ($C(0.6, 0.8) = 0.8$) indica que la efectividad general del tratamiento, considerando tanto la reducción del dolor como la mejora de la movilidad, es del 80%. Esto sugiere que la efectividad general está determinada por el factor de calificación más alto (mejora de la movilidad), reflejando la naturaleza de “máximo” de la T-conorma.

4.1.2 Funciones de membresía y modelamiento difuso

4.1.2.1 Funciones de membresía

La **función de membresía** representa la asociación entre un elemento y un conjunto difuso, asignando un grado de pertenencia a dicho elemento en relación con el conjunto difuso. En otras palabras, una función de membresía describe cómo un elemento pertenece a un conjunto difuso en un continuo que va desde 0 (no pertenencia) hasta 1 (pertenencia total).

Imaginemos que medimos la temperatura de una vivienda y queremos analizar el nivel de frío. De forma difusa se podría definir una función de membresía como la presentada en la imagen:

Existen diferentes funciones de membresía, las más destacadas son:

- **Triangular:** esta función de membresía se caracteriza por tener una forma triangular. Tiene tres parámetros principales: a , b y c , que definen los límites del triángulo en el eje x donde a es el punto de inicio del triángulo, b es el punto medio y c es el punto final. La función aumenta linealmente desde 0 hasta 1, alcanzando su máximo en b y luego disminuye linealmente hasta 0. Es ampliamente utilizada debido a su simplicidad y facilidad de interpretación.
- **Trapezoidal:** la función de membresía trapezoidal tiene cuatro parámetros principales: a , b , c y d , que definen los límites del trapecio en el eje x . La función aumenta linealmente desde 0 hasta 1 entre a y b , permanece constante en 1 entre b y c , y luego disminuye linealmente hasta 0 entre c y d . Esta función es útil cuando se necesita modelar conjuntos con rangos más amplios o cuando se requiere un grado de pertenencia constante en un intervalo.

Ej: $\mu_F(x)$ corresponde al nivel de frío medido en la variable x

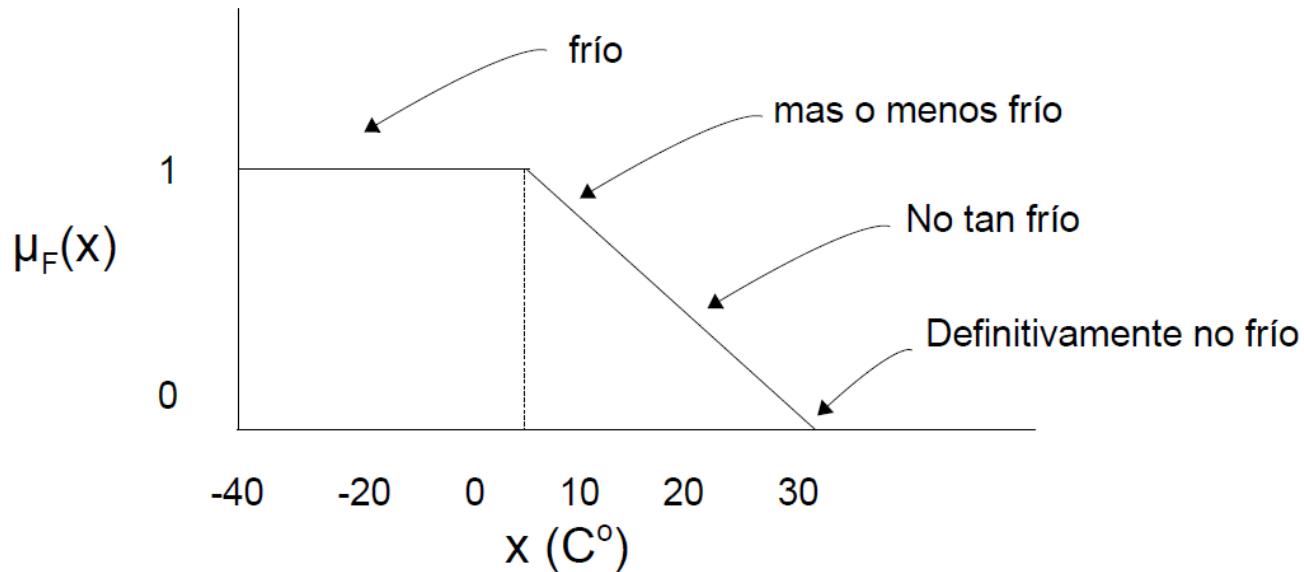


Imagen 4.1: Ejemplo Funciones Membresia

- **Gaussiana:** la función de membresía gaussiana tiene una forma de campana y se utiliza comúnmente para representar variables que tienen una distribución normal. Esta función tiene dos parámetros principales: μ , que representa la media de la distribución y σ que controla su anchura de la distribución. A medida que x se aleja de μ el grado de pertenencia disminuye gradualmente. La función gaussiana es útil para modelar conjuntos con distribuciones simétricas y para suavizar transiciones entre conjuntos difusos.
- **Sigmoidal:** se utiliza para representar relaciones no lineales entre variables. Esta función tiene tres parámetros principales: a , b y c , que controlan la pendiente y la posición de la curva sigmoidal. A medida que x aumenta desde a hasta b el grado de pertenencia aumenta gradualmente. Luego, a medida que x aumenta desde b hasta c el grado de pertenencia disminuye gradualmente. La función sigmoidal es útil para modelar relaciones complejas entre conjuntos difusos y es especialmente importante en problemas de inferencia difusa y control difuso.

4.1.2.2 Modelado difuso

El **modelado difuso** implica la creación de un sistema que puede interpretar y procesar información imprecisa o incierta mediante reglas lingüísticas. En este contexto, las reglas se componen de antecedentes y consecuentes.

- **Antecedentes:** son las condiciones o variables de entrada que se evalúan para determinar qué acciones o decisiones tomar. Estos antecedentes se expresan en términos lingüísticos, como “temperatura fría” o “velocidad rápida”.

El antecedente se expresa mediante una función de pertenencia difusa $\mu_A(x)$, donde A es el nombre del antecedente y x es el valor de la variable de entrada. Así, si A es la temperatura y x es el valor de la

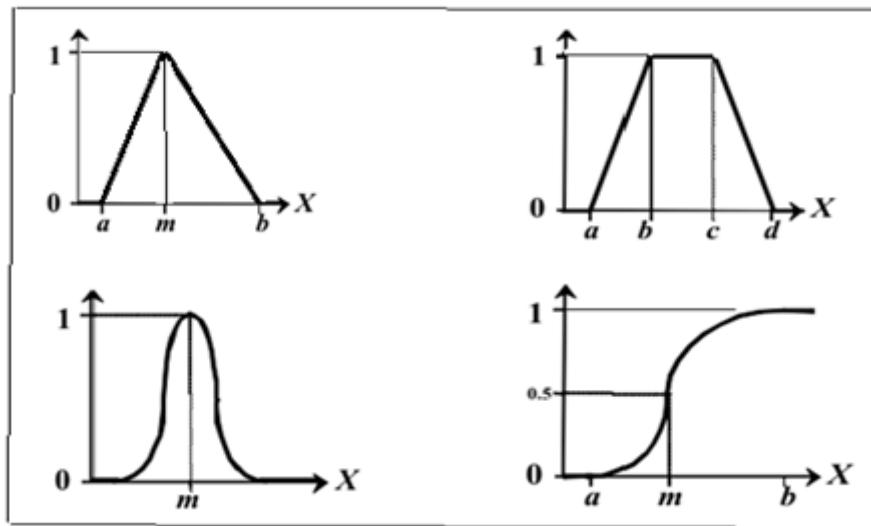


Imagen 4.2: Funciones Membresia

temperatura en grados Celsius entonces $\mu_A(x)$ podría representar el grado de pertenencia de x a la categoría “frío”, “templado” o “caliente”.

- **Consecuentes:** son las acciones o decisiones que se toman en respuesta a las condiciones evaluadas en los antecedentes. Estos consecuentes también se expresan en términos lingüísticos y representan las salidas del sistema difuso. En el ejemplo del sistema de control de clima, los consecuentes podrían ser ajustes en el sistema de calefacción, ventilación o aire acondicionado.

Existen diferentes modelamientos de consecuentes, los más destacados son:

- **Mamdani:** los consecuentes de las reglas difusas se expresan como funciones de pertenencia difusas. Estas funciones representan la contribución de cada regla al resultado final del sistema difuso. Por lo general, estas funciones son conjuntos difusos que se solapan entre sí, lo que significa que una regla puede contribuir parcialmente al resultado final del sistema. Por ejemplo, si tenemos un conjunto de reglas difusas del tipo *SI temperatura es fría Y humedad es alta ENTONCES activar calefacción*, el consecuente para esta regla podría expresarse como una función de pertenencia difusa $\mu_C(y)$, donde y es el valor de la salida. Así, esta función de pertenencia podría representar la intensidad de la calefacción.
- **Sugeno:** los consecuentes de las reglas difusas se expresan como funciones lineales de la forma $y = ax + b$, donde x es el valor de la entrada e y es el valor de la salida. En lugar de producir conjuntos difusos como en el enfoque de Mamdani, las reglas Sugeno generan un valor numérico preciso como resultado; lo que hace que el enfoque de Sugeno sea más adecuado para aplicaciones donde se requiere una salida numérica clara y no una respuesta lingüística. Así, continuando con el ejemplo anterior, *SI temperatura es fría Y humedad es alta ENTONCES activar calefacción con intensidad 0.7*, el consecuente para esta regla podría expresarse como $y = 0.7x$.
- **Tsukamoto:** los consecuentes se expresan como funciones de pertenencia escalonadas. Estas funciones asignan un grado de pertenencia a diferentes conjuntos difusos de salida en función de la evaluación de los antecedentes. El enfoque de Tsukamoto es útil cuando se necesita un sistema de control que pueda adaptarse a cambios en las condiciones de entrada de manera suave y gradual.

$$y = \frac{\sum_{i=1}^n \mu_{A_i}(x_i) \cdot V_i}{\sum_{i=1}^n \mu_{A_i}(x_i)}$$

donde y es el valor de salida, $\mu_A(x)$ es la función de pertenencia difusa del antecedente A_i evaluada en el valor de entrada, V_i es el valor asociado al consecuente para el antecedente A_i y n es el número de antecedentes en la regla

Esta fórmula calcula el valor de salida como una combinación ponderada de los valores asociados V_i para cada antecedente, donde los pesos están determinados por las funciones de pertenencia de los antecedentes. La suma de las funciones de pertenencia de los antecedentes normaliza el resultado, asegurando que esté en el rango adecuado.

Estas fórmulas proporcionan una manera matemática de expresar los antecedentes y consecuentes en un sistema de lógica difusa, lo que permite modelar y controlar sistemas basados en reglas lingüísticas y condiciones difusas.

La salida de un sistema difuso es, por tanto, un conjunto difuso. En cambio, los sistemas de control se relacionan con el mundo externo a través de valores exactos lo que implica, a la hora de hacer inferencia, es necesario realizar un proceso de **defuzzificación**. Así, los principales métodos de defuzzificación son:

- **Centroide:** calcula el centro de gravedad del conjunto difuso ponderado. Consiste en encontrar el punto en el eje de salida donde el área bajo la curva del conjunto difuso es dividida en dos áreas iguales. Matemáticamente, el valor de salida y se calcula como

$$y = \frac{\sum_{i=1}^n u_i \cdot x_i}{\sum_{i=1}^n u_i}$$

- **Máximo Valor de Membresía:** selecciona el valor de salida con el grado de membresía más alto. Es decir, el valor de salida es el punto en el conjunto difuso donde la membresía es máxima. Así, $y = \max(u(x))$
- **Media de los Máximos:** se calcula la media de todos los puntos en el conjunto difuso donde el grado de membresía es máximo. Es útil cuando hay múltiples picos en el conjunto difuso
- **Primera Máxima y Última Máxima:** se selecciona el primer y último punto de máximo grado de membresía, respectivamente, como valores de salida
- **Bisectriz:** este método divide el conjunto difuso en dos áreas de igual tamaño y toma el punto medio entre los dos puntos de intersección de la bisectriz con el conjunto difuso

4.2 Ejemplos prácticos

Para ilustrar un ejemplo de control de sistemas difuso, consideremos un sistema difuso simple para controlar la potencia de un calefactor basado en la temperatura ambiente. Supongamos que queremos mantener la temperatura en una habitación alrededor de un valor de referencia.

Definición de las Variables Difusas - Temperatura: esta variable representa la temperatura medida en grados Celsius. Creamos una variable de entrada temperatura que abarca valores desde 0 hasta 100 grados Celsius. - **Potencia:** esta variable representa el nivel de potencia de la calefacción. Creamos una variable de salida potencia que también abarca valores desde 0 hasta 100.

Funciones de Membresía - Temperatura: se definen tres funciones de membresía para la temperatura: “fría”, “templada” y “caliente”. - **Potencia:** de igual forma, se definen tres funciones de membresía para la potencia de la calefacción: “baja”, “media” y “alta”.

Reglas difusas Se define el sistema de reglas difusas: - Si la temperatura es fría, entonces la potencia debe ser alta - Si la temperatura es alta, entonces la potencia debe ser baja - Si la temperatura es templada, entonces la potencia debe ser media

Sistema de Control Se crea un conjunto de reglas difusas a partir de cada una de las reglas definidas previamente. Este conjunto de reglas representa el conocimiento difuso que guiará el comportamiento del sistema de control.

Controlador Difuso Se crea un controlador difuso que toma como argumento el sistema de control difuso creado anteriormente. Este controlador simula el comportamiento del sistema de control difuso y nos permite introducir valores de entrada (temperatura) para obtener el valor de salida (potencia) correspondiente.

Simulación Se simula un cambio en la temperatura estableciendo un valor para la variable de entrada temperatura en el controlador difuso. Luego, se calcula el valor de salida (nivel de potencia) utilizando el método compute() del controlador.

Finalmente, el código python:

```

1 import numpy as np
2 import skfuzzy as fuzz
3 from skfuzzy import control as ctrl
4
5 # Definición de las variables de entrada y salida
6 temperatura = ctrl.Antecedent(np.arange(0, 101, 1), 'temperatura')
7 potencia = ctrl.Consequent(np.arange(0, 101, 1), 'potencia')
8
9 # Definición de las funciones de membresía para la temperatura
10 temperatura['fría'] = fuzz.trimf(temperatura.universe, [0, 0, 50])
11 temperatura['templada'] = fuzz.trimf(temperatura.universe, [0, 50, 100])
12 temperatura['caliente'] = fuzz.trimf(temperatura.universe, [50, 100, 100])
13
14 # Definición de las funciones de membresía para la potencia
15 potencia['baja'] = fuzz.trimf(potencia.universe, [0, 0, 50])
16 potencia['media'] = fuzz.trimf(potencia.universe, [0, 50, 100])
17 potencia['alta'] = fuzz.trimf(potencia.universe, [50, 100, 100])
18
19 # Reglas difusas
20 regla1 = ctrl.Rule(temperatura['fría'], potencia['alta'])
21 regla2 = ctrl.Rule(temperatura['templada'], potencia['media'])
22 regla3 = ctrl.Rule(temperatura['caliente'], potencia['baja'])
23
24 # Sistema de control difuso
25 sistema_control = ctrl.ControlSystem([regla1, regla2, regla3])
26 controlador = ctrl.ControlSystemSimulation(sistema_control)
27
28 # Simulación de un cambio en la temperatura
29 controlador.input['temperatura'] = 25 # Temperatura medida en grados Celsius
30

```

```

31 # Computar el resultado
32 controlador.compute()
33
34 # Visualización del resultado
35 print("Nivel de potencia de la calefacción:", controlador.output['potencia'])

```

A continuación, se representan e interpretan las funciones de membresía antecedente y consecuente del sistema de control:

```

1 # Detalle de las funciones de membresía (antecedente y consecuente)
2
3 temperatura.view(sim=controlador)
4
5 potencia.view(sim=controlador)

```

4.3 Fuzzy C-Means

Los algoritmos de clustering difuso, como el Fuzzy C-Means (FCM), son esenciales en la agrupación de datos donde la pertenencia de un punto a un clúster no es binaria, sino que se modela como un grado de pertenencia difuso. FCM asigna a cada punto un grado de pertenencia a todos los clústeres, lo que permite una representación más flexible de la estructura de los datos. En lugar de asignar cada punto a un único clúster, FCM asigna grados de pertenencia que indican la probabilidad de que un punto pertenezca a cada clúster. Esto es especialmente útil cuando los datos pueden pertenecer a múltiples clústeres simultáneamente, o cuando la frontera entre clústeres no es clara. La importancia de los algoritmos de clustering difuso radica en su capacidad para manejar la incertidumbre en los datos y proporcionar una agrupación más completa y flexible que los métodos tradicionales de clustering.

A diferencia del algoritmo C-Means clásico, que trabaja con una partición dura, FCM realiza una partición suave del conjunto de datos. En tal partición los datos pertenecen en algún grado a todos los clusters.

El siguiente código muestra cómo hacer un C-Means difuso:

```

1 import pandas as pd
2 import numpy as np
3 import seaborn as sns
4 import matplotlib.pyplot as plt
5 import skfuzzy as fuzz
6 from sklearn import datasets
7
8 # Carga de datos de Iris desde scikit-learn
9 iris = datasets.load_iris()
10 X = iris.data # Características
11 y = iris.target # Etiquetas
12
13 # Gráfico de dispersión para visualizar los clusters originales según las dos primeras
   ↵ características (sépalos)

```

```

14 g = sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y, palette="Set2")
15 plt.title('Sépalo: longitud vs tamaño'), plt.xlabel('Longitud'), plt.ylabel('Ancho')
16 plt.show()
17
18 # Algoritmo Fuzzy C-Means
19 # Es necesario que los datos estén en un array 2D, por lo que se utiliza la función
20 # reshape
20 X2 = np.reshape(X.T, (X.shape[1], X.shape[0])) # Cambio de forma de los datos
21
22 # Aplicación del modelo Fuzzy C-Means
23 cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(X2, c=3, m=2, error=0.005, maxiter=1000,
24 init=None, seed=111)

```

Entre los parámetros que devuelve la función `fuzz.cluster.cmeans` se pueden citar: - `cntr` es el centro de los clusters - `u` es el grado de membresía - `u0` es la matriz inicial de membresía - `d` es la matriz de distancias euclíadiana

```

1 # Centro de los clusters
2 cluster_centers_df = pd.DataFrame(cntr, columns=['Longitud Sépalo', 'Ancho Sépalo',
3     'Longitud Pétalo', 'Ancho Pétalo'],
4                                     index=['Cluster 1', 'Cluster 2', 'Cluster 3'])
4 cluster_centers_df

```

```

1 # Grados de membresía (probabilidad de pertenencia a cada cluster)
2 membership_df = pd.DataFrame(u.T, columns=['Cluster 1', 'Cluster 2', 'Cluster 3'])
3 membership_df.head()

```

```

1 # Gráfico de dispersión para visualizar los clusters obtenidos
2 g = sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=np.argmax(u.T, axis=1), palette="Set2") # Asignación de colores según cluster
3 plt.title('Sépalo: longitud vs tamaño'), plt.xlabel('Longitud'), plt.ylabel('Ancho')
4 plt.show()

```

4.4 Fuzzy Matching

El **Fuzzy Matching** es una técnica utilizada para comparar cadenas de texto y determinar su grado de similitud, incluso cuando hay diferencias ortográficas, tipográficas u otras variaciones entre ellas. Esta técnica identifica la probabilidad de que dos registros sean realmente coincidentes en función de si coinciden o no en diferentes identificadores. Los identificadores elegidos y el peso asignado constituyen la base de la *Concordancia Difusa*. Así, cuando los parámetros son demasiado amplios, se encontrarán más coincidencias, pero también aumentarán las posibilidades de “falsos positivos”.

Existen diversas métricas a utilizar en Fuzzy Matching siendo la más destacada la *distancia de Levenshtein* pero existen otras como se indican a continuación:

- **Distancia de Levenshtein:** mide el número mínimo de operaciones necesarias para convertir una cadena en otra. Las operaciones incluyen inserción, eliminación o sustitución de caracteres

- **Distancia Damerau-Levenshtein:** similar a la distancia de Levenshtein, pero también considera la transposición de caracteres adyacentes como una operación válida
- **Distancia Jaro-Winkler:** diseñada para comparar cadenas de texto cortas, tiene en cuenta la frecuencia de los caracteres y la posición de las coincidencias comunes
- **Distancia del teclado:** evalúa la similitud entre dos cadenas basándose en la proximidad de las teclas en un teclado estándar
- **Distancia Kullback-Leibler:** utilizada en comparaciones de cadenas de texto que representan distribuciones de probabilidad

La **concordancia difusa** es crucial para identificar similitudes entre registros, permitiendo la gestión eficiente de datos duplicados. La distancia de Levenshtein es fundamental en este proceso, al calcular la diferencia entre dos cadenas de caracteres, siendo especialmente útil para detectar errores ortográficos y variaciones en la escritura. Más sobre Distancia de Levenshtein en el siguiente enlace: https://es.wikipedia.org/wiki/Distancia_de_Levenshtein

4.4.1 Ejemplo 1

```

1 from fuzzywuzzy import fuzz
2
3 # Función para comparar nombres completos
4 def comparar_nombres(nombre1, nombre2):
5     # Usamos el ratio completo para comparar nombres
6     ratio = fuzz.ratio(nombre1.lower(), nombre2.lower())
7     return ratio
8
9 # Ejemplo de comparación de nombres
10 nombre1 = "Juan Pérez"
11 nombre2 = "Juan Pérez Gómez"
12 nombre3 = "Pedro López"
13 nombre4 = "María Pérez Gómez"
14
15 print("Comparación de nombres completos:")
16 print(f"{nombre1} vs {nombre2}: {comparar_nombres(nombre1, nombre2)}")
17 print(f"{nombre1} vs {nombre3}: {comparar_nombres(nombre1, nombre3)}")
18 print(f"{nombre2} vs {nombre4}: {comparar_nombres(nombre2, nombre4)})
```

4.4.2 Ejemplo 2

```

1 # Función para comparar direcciones de vivienda
2 def comparar_direcciones(direccion1, direccion2):
3     # Usamos el ratio parcial para comparar direcciones
4     ratio = fuzz.partialratio(direccion1.lower(), direccion2.lower())
5     return ratio
6
7 # Ejemplo de comparación de direcciones
```

```
8 direccion1 = "123 Calle Principal, Madrid"
9 direccion2 = "245 Calle Principal, Madrid"
10 direccion3 = "456 Calle Secundaria, Ávila"
11 direccion4 = "123 Calle Principal, Ávila"
12
13 print("\nComparación de direcciones de vivienda:")
14 print(f"{direccion1} vs {direccion2}: {comparar_direcciones(direccion1, direccion2)}")
15 print(f"{direccion1} vs {direccion3}: {comparar_direcciones(direccion1, direccion3)}")
16 print(f"{direccion2} vs {direccion4}: {comparar_direcciones(direccion2, direccion4)}")
17 print(f"{direccion3} vs {direccion4}: {comparar_direcciones(direccion3, direccion4)})")
```

En la biblioteca **fuzzywuzzy**, las funciones **ratio()** y **partial_ratio()** se utilizan para calcular la similitud entre dos cadenas de texto. La diferencia principal radica en cómo se realiza la comparación y qué tan flexible es cada método en términos de coincidencia.

- **ratio()**: esta función calcula la similitud entre dos cadenas comparándolas en su totalidad. Compara las cadenas carácter por carácter y devuelve un valor que representa la similitud entre las dos cadenas en términos de coincidencia exacta. Es útil cuando se busca una coincidencia exacta entre las cadenas y se prefiere una comparación estricta.
- **partial_ratio()**: esta función calcula la similitud entre dos cadenas tomando en cuenta solo una parte de las mismas. Se utiliza para buscar subcadenas que coincidan parcialmente entre las cadenas de texto. Esto significa que puede ser más útil cuando se enfrenta a casos donde las cadenas pueden tener variaciones menores o cuando se desea encontrar coincidencias incluso si las cadenas difieren en longitud o tienen diferencias menores.

Entonces, para decidir cuál usar, considera la naturaleza de los datos y el nivel de flexibilidad para realizar la comparación. Suele usarse **ratio()** cuando se necesita una comparación estricta y quieras asegurarte de que las cadenas coincidan exactamente; mientras que **partial_ratio()** se emplea cuando se buscan coincidencias parciales o flexibles entre las cadenas (p.e: cuando solo importa una parte específica de las mismas).

A Anexo 1

Anexo 1

Bibliography

- [1] C. Darwin. *On the Origin of Species by Means of Natural Selection*. 1859.
- [2] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York., 1966.
- [3] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.
- [4] Donald E. Knuth. “Literate Programming”. In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: [10.1093/comjnl/27.2.97](https://doi.org/10.1093/comjnl/27.2.97). URL: <https://doi.org/10.1093/comjnl/27.2.97>.
- [5] A. Marczyk. “Algoritmos genéticos y computación evolutiva”. In: (2004). URL: <https://the-geek.org/docs/algen/>.
- [6] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart., 1973.