# REDES NEURONALES

## ¿QUÉ ES UNA NEURONA ARTIFICIAL?

Una neurona artificial es una función matemática concebida como un modelo de una neurona biológica.

Diagrama

Descripción generada automáticamente

*Una neurona biológica*

Un dibujo de un árbol

Descripción generada automáticamente con confianza baja

*Múltiples capas en una red neuronal biológica*

La neurona artificial recibe una o más entradas y las suma para producir una salida. Por lo general, cada entrada se pondera por separado con un peso y la suma se pasa a través de una función no lineal conocida como **función de activación**.

Diagrama

Descripción generada automáticamente

Video: [¿Qué es una Red Neuronal? Parte 1: La Neurona](https://www.youtube.com/watch?v=MRIv2IwFTPg) (9:14min)

[¿Qué es una Red Neuronal? Parte 2: La Red](https://www.youtube.com/watch?v=uwbHOpp9xkc) (11:33min)

¿Qué es una Red Neuronal? Parte 3 : Backpropagation (11:21min)

¿Qué es una Red Neuronal? Parte 3.5 : Las Matemáticas de Backpropagation (17:20min)

## ¿QUÉ ES UNA RED NEURONAL?

Una red neuronal artificial (llamada simplemente red neuronal) se basa en una colección de neuronas artificiales. Cada neurona está conectada a otras neuronas y la señal de salida de una neurona es la entrada de la siguiente neurona conectada a ella.

Una neurona artificial recibe señales (número real), las procesa (calcula mediante alguna función la suma de sus entradas) y si el resultado cruza cierto umbral envía la salida a las neuronas conectadas a ella.

Las neuronas y las conexiones suelen tener un peso que se ajusta a medida que avanza el aprendizaje. El peso aumenta o disminuye la fuerza de la señal en una conexión.

Por lo general, las neuronas se agrupan en capas con diferentes números de neuronas. Las señales viajan desde la primera capa (la capa de entrada) hasta la última capa (la capa de salida), posiblemente después de a travesar varias capas ocultas.

La unión de todas las neuronas permite descubrir patrones complejos en los datos de entrada.

Video: [Cómo funcionan las redes neuronales - Inteligencia Artificial](https://www.youtube.com/watch?v=CU24iC3grq8) (11:51 min)

### TIPOS DE REDES NEURONALES

Existen varios tipos de redes neuronales, cada una diseñada para abordar diferentes tipos de problemas y tareas en el ámbito del aprendizaje profundo. A continuación, se describen algunos de los tipos más comunes de redes neuronales:

**Redes Neuronales Densamente Conectadas (DNN):**

* También conocidas como redes neuronales totalmente conectadas o perceptrones multicapa.
* Cada neurona en una capa está conectada a todas las neuronas en la capa siguiente.
* Utilizadas comúnmente en tareas de clasificación y regresión.

**Redes Neuronales Convolucionales (CNN):**

* Diseñadas específicamente para procesar datos estructurados en cuadrículas, como imágenes.
* Utilizan capas de convolución para aprender características locales.
* Ampliamente utilizadas en visión por computadora para tareas como clasificación de imágenes.

**Redes Neuronales Recurrentes (RNN):**

* Diseñadas para procesar secuencias de datos, como series temporales o texto.
* Contienen conexiones retroalimentadas, lo que les permite recordar información pasada.
* Útiles para tareas como procesamiento del lenguaje natural (NLP) y predicción de series temporales.

**Redes Neuronales Transformer:**

* Diseñadas para procesar secuencias de datos, como en tareas de NLP.
* Utilizan mecanismos de atención para capturar relaciones entre diferentes partes de la secuencia.

**Redes Neuronales Generativas Adversarias (GAN):**

* Combinan dos redes, un generador y un discriminador, que compiten entre sí.
* Utilizadas para generar datos nuevos y realistas, como imágenes.
* Pueden aprender a generar datos sintéticos de alta calidad.

**Redes Neuronales Autoencoder:**

* Aprenden representaciones compactas de los datos a través de una estructura codificador-decodificador.
* Útiles para reducción de dimensionalidad y reconstrucción de datos.

…

### EJEMPLO RED NEURONAL

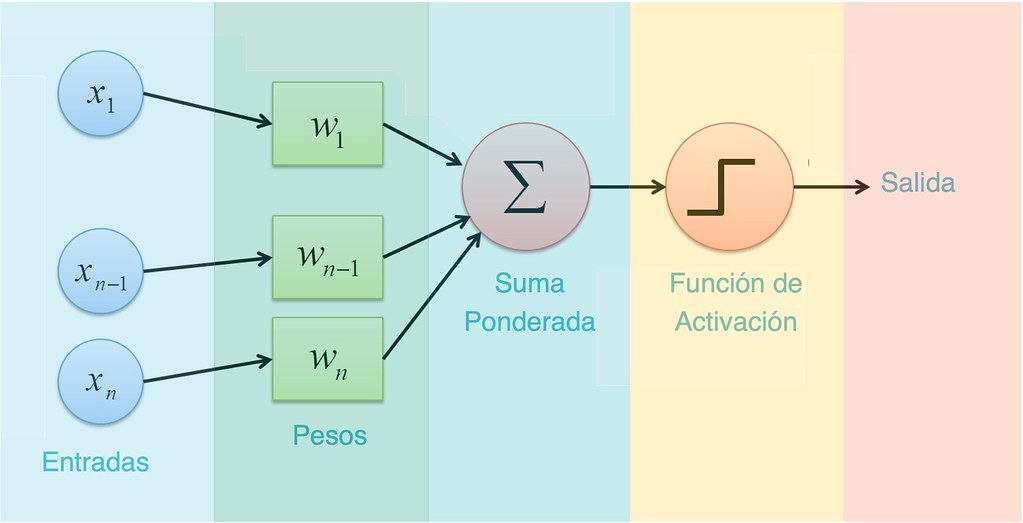
En la siguiente figura vemos una red neuronal de 4 capas: una de entrada (input layer) que recibe los datos de entrada, una de salida (output layer) que devuelve la predicción realizada, y dos capas ocultas internas (hidden layers).

Diagrama

Descripción generada automáticamente

### FUNCIONES DE ACTIVACIÓN

La función de activación determina, como su propio nombre indica, el nivel de activación que alcanza cada neurona una vez que ha recibido los impulsos transmitidos. Estas funciones ostentan un rol muy importante en la determinación del poder computacional de la red neuronal.



La función de activación se encarga de devolver una salida a partir de un valor de entrada, normalmente el conjunto de valores de salida en un rango determinado como (0,1) o (-1,1).

Se buscan funciones que las derivadas sean simples, para minimizar con ello el coste computacional.

Algunas de las funciones de activación más usadas son:

|  |  |  |  |
| --- | --- | --- | --- |
| NOMBRE | VISUALIZACÓN | FUNCIÓN | OBSERVACIONES |
| Lineal |  |  | No es útil para las capas ocultas. |
| ReLU  (Rectified Linear) |  |  | Se comporta bien con imágenes.  Buen desempeño en redes convolucionales. |
| Tanh |  |  | Se utiliza para decidir entre uno opción y la contraria.  Buen desempeño en redes recurrentes. |
| Sigmoide |  |  | Buen rendimiento en la última capa. |
| Softmax |  |  | La función Softmax transforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas de 1.  Buen rendimiento en las últimas capas, para clasificación multiclase. |

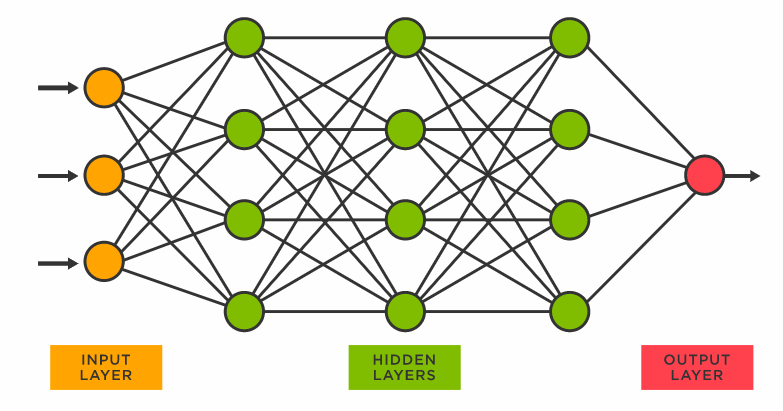
Gráfico, Gráfico de líneas

Descripción generada automáticamente

Más información: <https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/>

## REDES NEURONALES DENSAMENTE CONECTADAS - DNN

Cuando todas las neuronas de una capa están conectadas con todas las neuronas de la capa anterior, la capa se denomina completamente/densamente conectada.



### CONJUNTOS DE DATOS: ENTRENAMIENTO, VALIDACIÓN, TEST

La división en conjuntos de entrenamiento, validación y prueba es crucial para desarrollar modelos de aprendizaje automático que sean capaces de generalizar bien a datos no vistos.

Aquí hay un ejemplo básico de cómo realizar la división utilizando Python y la biblioteca scikit-learn:

*X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.3, random\_state=42)*

*X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)*

La división típica es en proporciones como 70-15-15 o 80-10-10, donde el 70% o 80% se destina al conjunto de entrenamiento, el 15% o 10% al conjunto de validación, y el 15% o 10% restante al conjunto de prueba. La elección exacta de estas proporciones puede depender de la cantidad de datos disponible y de la complejidad de la tarea.

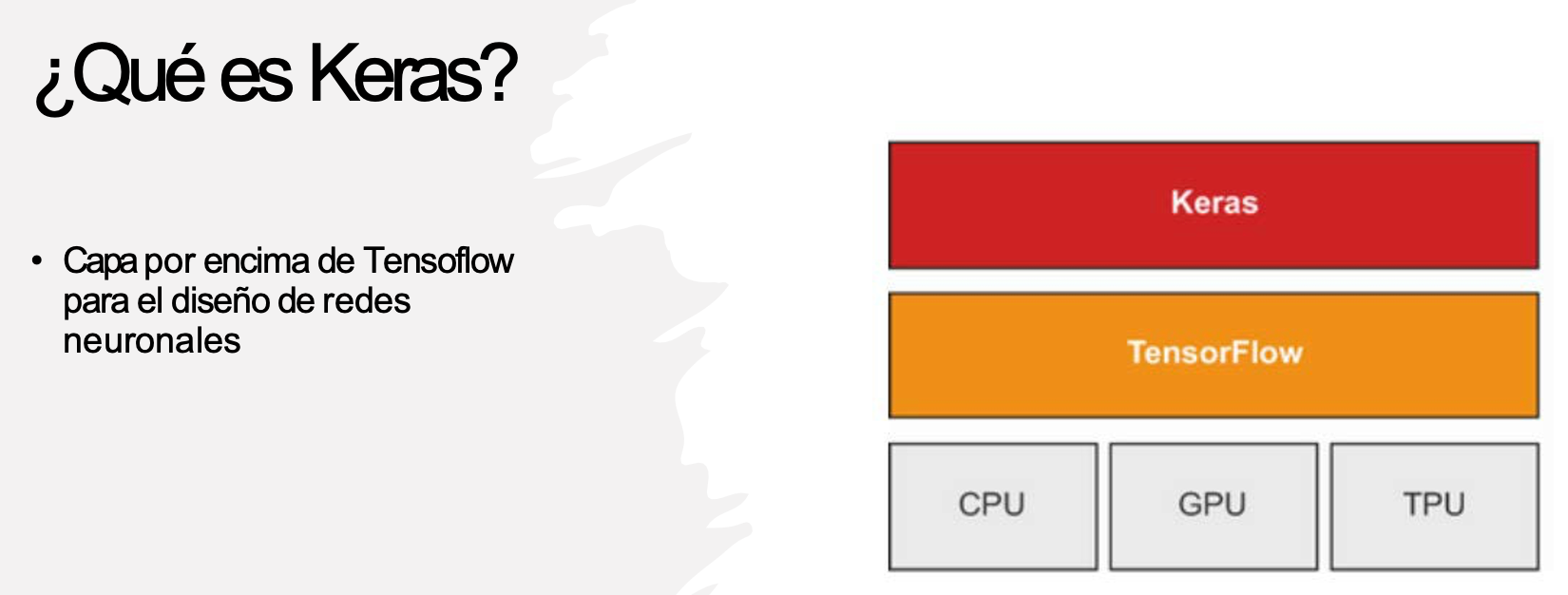
Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Gráfico

Descripción generada automáticamente con confianza media

### KERAS



**¿Qué es Keras?**

Keras es una interfaz de programación de aplicaciones (API) de alto nivel para la construcción y entrenamiento de modelos de aprendizaje profundo. A partir de TensorFlow 2.0, Keras se convirtió en la interfaz de alto nivel oficial de TensorFlow para construir y entrenar modelos.

**¿Qué es Tensor Flow?**

TensorFlow es una biblioteca de código abierto desarrollada por Google que se utiliza para construir y entrenar modelos de aprendizaje profundo. Proporciona una plataforma flexible y eficiente para desarrollar algoritmos de aprendizaje automático y realizar cálculos numéricos utilizando tensores.

### ETAPAS PARA CREAR UNA RED NEURONAL DENSAMENTE CONECTADA

Estas son las principales etapas para crear una red neuronal densamente conectada:

* **Preparación de Datos**:
  + Recopilación de Datos: obtener un conjunto de datos representativo y suficiente para la tarea que deseas abordar.
  + Exploración de Datos: analizar y comprender la estructura, distribución y características del conjunto de datos.
  + Preprocesamiento de Datos: realizar tareas como normalización, escalado y manejo de valores faltantes para preparar los datos para el entrenamiento.
* **Dividir los datos**: conjunto de Entrenamiento, Validación y Prueba
* **Creación del modelo y definición de cada una de las capas**:

*model = Sequential()*

*model.add(Dense(<número de neuronas>, activation=<función de activación (‘sigmoid’, ‘softmax’, ‘relu’...))*

* + Definición de arquitectura: determinar el número de capas ocultas, el número de neuronas en cada capa, etc.
  + Seleccionar **funciones de activación**: elegir funciones de activación para introducir no linealidades en la red, como ReLU (Rectified Linear Unit) en las capas ocultas y softmax en la capa de salida para clasificación.
  + Cuidado con:
    - Demasiadas neuronas 🡪 overfitting
    - Demasiado pocas 🡪 pérdida de información
* Configuración del proceso de aprendizaje: **método compile**

*model.compile(loss="mean\_squared\_error", optimizer="adam", metrics=["accuracy"])*

* + **Función de pérdida (loss)**: evalúa el grado de error entre las salidas calculadas y las salidas deseadas de los datos de entrenamiento. El objetivo está en reducir dicho valor en cada iteración.
* **Métrica (metrics):** es la que usaremos para monitorizar el proceso de aprendizaje y prueba de nuestra red neuronal. En este ejemplo, se está utilizando 'accuracy' para medir la precisión del modelo. Puedes especificar varias métricas en forma de lista al utilizar el parámetro metrics en el método compile.
  + **Optimizador (optimizer**): es la manera que tenemos de indicar los detalles del algoritmo de optimización que permite a la red neuronal calcular los pesos de los parámetros durante el entrenamiento a partir de los datos de entrada y de la función de coste definida. Habitualmente utilizaremos optimizadores basados en el descenso de gradiente como SGD (Descenso de Gradiente Estocástico).

Video: [¿Qué es el Descenso del Gradiente? Algoritmo de Inteligencia Artificial](https://www.youtube.com/watch?v=A6FiCDoz8_4) (9:23min)

Diagrama

Descripción generada automáticamente

* Entrenamiento con los datos de training: **método fit**
  + **epochs:** número de veces que iteraremos en el proceso de aprendizaje.
  + **verbose**: permite ver el avance del entrenamiento, así como una estimación de cuánto tarda cada época.
* Evaluación con los datos de testing: **método evaluate**
* Generación de predicciones: **método predict**

Mirar los ficheros de ejemplos 4\_2\_XEjemplos redes neuronales.ipynb

### FUNCIÓN DE PÉRDIDA (LOSS FUNCTION)

La Función de Pérdida (Loss Function) es una medida que cuantifica la discrepancia entre las predicciones del modelo y los valores reales o etiquetas en el conjunto de datos. La función de pérdida juega un papel crucial en el entrenamiento del modelo, ya que proporciona una señal de retroalimentación que guía los ajustes de los pesos y sesgos de la red durante la fase de aprendizaje.

El objetivo principal es proporcionar una medida cuantitativa de qué tan bien está realizando el modelo en la tarea específica para la cual se está entrenando. Al minimizar esta función durante el proceso de entrenamiento, el modelo busca aprender patrones y relaciones en los datos que le permitan realizar predicciones más precisas.

Existen diferentes funciones de error según trabajamos con problemas de regresión o clasificación.

Algunas funciones de pérdida usadas en regresión:

* **MeanSquaredError**: calcula el promedio de los cuadrados de las diferencias entre las etiquetas y predicciones.
* **MeanAbsoluteError**: calcula el promedio de las diferencias absolutas entre etiquetas y predicciones.
* **MeanAbsolutePercentageError**: similar a MeanAbsoluteError, pero expresado como un porcentaje.
* **MeanSquaredLogarithmicError**: útil cuando las predicciones tienen un rango amplio; calcula el logaritmo del error cuadrático medio.
* **CosineSimilarity**: mide la similitud coseno entre las etiquetas y las predicciones, comúnmente usada para problemas de aprendizaje de representaciones.
* **Huber**: combina MeanSquaredError y MeanAbsoluteError; menos sensible a los valores atípicos.
* **LogCosh**: calcula el logaritmo del coseno hiperbólico del error; una función de error suave.

Algunas funciones de pérdida usadas en clasificación:

* **BinaryCrossentropy**: para problemas de clasificación binaria. Mide la discrepancia entre las etiquetas verdaderas y las predicciones.
* **CategoricalCrossentropy**: utilizada para clasificación multiclase donde las etiquetas son one-hot encoded.
* **SparseCategoricalCrossentropy**: similar a CategoricalCrossentropy, pero para etiquetas como enteros.
* **Poisson**: para problemas de conteo, asume que las etiquetas siguen una distribución de Poisson.
* **KLDivergence (Divergencia Kullback-Leibler)**: Mide cómo una distribución de probabilidad difiere de otra distribución de referencia.

<https://keras.io/api/losses/>

### OPTIMIZADORES

Se utilizan durante el proceso de entrenamiento para ajustar iterativamente los pesos y sesgos del modelo con el objetivo de minimizar la función de pérdida. La mayoría de los optimizadores están basados en el descenso de gradiente:

Diagrama

Descripción generada automáticamente

Aquí hay una lista de algunos optimizadores disponibles y una breve descripción de sus propiedades:

1. **SGD (Descenso de Gradiente Estocástico):** es un optimizador tradicional, adecuado para problemas a gran escala y con hiperparámetros como la tasa de aprendizaje que se pueden ajustar.
2. **RMSprop:** se diseñó para abordar las deficiencias del descenso de gradiente estocástico. Utiliza una media móvil de los gradientes al cuadrado para normalizar el gradiente.
3. **Adam:** es un optimizador que combina la idea de tasas de aprendizaje adaptativas para diferentes parámetros, usando promedios de primer y segundo orden de los gradientes.
4. **AdamW:** una variante de Adam que incluye la regularización del decay de los pesos como parte de la actualización de los parámetros.
5. **Adagrad:** adapta la tasa de aprendizaje para cada parámetro, escalando los gradientes por la raíz cuadrada del sumatorio de todos los gradientes pasados al cuadrado.
6. **Adamax:** una variante de Adam basada en la norma infinita, que a menudo es menos sensible a los errores de configuración de los hiperparámetros.
7. **Nadam:** integra la idea de tasas de aprendizaje adaptativas en cada paso del gradiente, inspirándose en la eficiencia de Adam.

<https://keras.io/api/optimizers/>

Durante una iteración del entrenamiento (época), el conjunto de entrenamiento se pasa hacia adelante a través de la red para obtener las predicciones y se calcula la función de pérdida.

Luego, el algoritmo de backpropagation se utiliza para calcular los gradientes de la función de pérdida con respecto a los parámetros de la red.

Finalmente, el optimizador utiliza estos gradientes para realizar ajustes en los pesos y sesgos de la red, moviéndolos en la dirección que minimiza la pérdida.

Algunas funciones clave del optimizador:

La elección del optimizador y sus hiperparámetros puede afectar significativamente el rendimiento y la eficiencia del entrenamiento de la red neuronal.

### MÉTRICAS

Las métricas se utilizan para evaluar el rendimiento del modelo durante o después del entrenamiento. Las funciones de métrica son similares a las funciones de pérdida, excepto que los resultados de evaluar una métrica no se utilizan al entrenar el modelo. Ten en cuenta que puedes usar cualquier función de pérdida como una métrica.

Veamos algunas de las métricas más usadas

**Métricas de Precisión (Accuracy):** estas métricas evalúan la precisión de las predicciones en tareas de clasificación.

* **Accuracy**: la precisión general de las predicciones.
* **BinaryAccuracy**: la precisión para clasificación binaria.
* **CategoricalAccuracy**: la precisión para clasificación multiclase con etiquetas one-hot.
* **SparseCategoricalAccuracy**: la precisión para clasificación multiclase con etiquetas como enteros.

**Métricas Probabilísticas:** estas métricas son adecuadas para evaluar modelos que predicen distribuciones de probabilidad.

* **BinaryCrossentropy**: la entropía cruzada binaria entre las etiquetas verdaderas y las predicciones.
* **CategoricalCrossentropy**: la entropía cruzada para clasificación multiclase con etiquetas one-hot.
* **SparseCategoricalCrossentropy**: la entropía cruzada para clasificación multiclase con etiquetas como enteros.
* **KLDivergence**: la divergencia Kullback-Leibler, una medida de cómo una distribución de probabilidad difiere de una distribución de referencia.
* **Poisson**: mide la diferencia entre las predicciones y el logaritmo de las predicciones, un enfoque inspirado por la distribución de Poisson.

**Métricas de Regresión:** estas métricas son para problemas donde se predice un valor continuo.

* **MeanSquaredError**: el error cuadrático medio entre las etiquetas y las predicciones.
* **RootMeanSquaredError**: la raíz cuadrada del error cuadrático medio.
* **MeanAbsoluteError**: el error absoluto medio entre las etiquetas y las predicciones.
* **MeanAbsolutePercentageError**: el error porcentual absoluto medio.
* **MeanSquaredLogarithmicError**: el error logarítmico cuadrático medio.
* **CosineSimilarity**: mide la similitud coseno entre las predicciones y las etiquetas.
* **LogCoshError**: el logaritmo del coseno hiperbólico del error.

**Métricas de Clasificación Basadas en Positivos y Negativos Verdaderos/Falsos:** estas métricas evalúan la clasificación en términos de positivos y negativos verdaderos y falsos.

* **AUC (Área Bajo la Curva ROC)**: mide la calidad del modelo de clasificación independientemente del umbral de decisión.
* **Precision**: la precisión de las predicciones positivas.
* **Recall**: la capacidad del modelo para encontrar todas las muestras positivas.

<https://keras.io/api/metrics/>

## AJUSTES DE UNA RED NEURONAL

### NÚMERO DE CAPAS OCULTAS

Para muchos problemas, podemos empezar con una sola capa oculta y obtener resultados razonables. Pero, para problemas complejos, las redes profundas tienen mucha más eficiencia.

Al agregar capas ocultas, es importante tener en cuenta el riesgo de sobreajuste. El sobreajuste puede ocurrir si el modelo se vuelve demasiado complejo para el conjunto de datos disponible, memorizando en lugar de generalizar. Se pueden aplicar técnicas como la regularización y el dropout para mitigar este riesgo.

Para entender por qué, imagina que te piden que dibujes un bosque con un software de dibujo, pero no puedes copiar ni pegar nada. Te llevaría un montón de tiempo: habría que dibujar cada árbol de forma individual, rama por rama, hoja por hoja. En cambio, si pudieses dibujar una hoja, copiarla y pegarla para hacer una rama, luego copiar la rama para crear un árbol y al final copiar el árbol para crear el bosque, acabarías enseguida. Los datos del mundo real se estructuran a menudo de la misma forma jerárquica y las redes neuronales profundas lo aprovechan automáticamente: las capas ocultas bajas modelan estructuras de bajo nivel (por ejemplo, segmentos lineales de varias formas y orientaciones), las capas ocultas intermedias combinan estas estructuras de bajo nivel para modelar estructuras de nivel intermedio (por ejemplo, cuadrados, círculos) y las capas ocultas más altas y la capa de salida combinan esas estructuras intermedias para modelar estructuras de alto nivel (por ejemplo, caras).

Esta estructura jerárquica no solo ayuda a las redes neuronales profundas a converger en una solución buena más rápido, sino que además mejora su capacidad para generalizar con conjuntos de datos nuevos. Por ejemplo, si ya hemos entrenado un modelo para que reconozca caras en imágenes y queremos entrenar una nueva red neuronal para que reconozca peinados, podemos empezar el entrenamiento reutilizando las capas bajas de la primera red. En lugar de inicializar aleatoriamente los pesos y sesgos de las primeras capas de la nueva red neuronal, podemos inicializarlas con los valores de los pesos y los sesgos de las capas inferiores de la primera red. De ese modo, la red no tendrá que aprender desde cero todas las estructuras de bajo nivel que hay en la mayoría de las imágenes; solo tendrá que aprender las estructuras de nivel superior (por ejemplo, peinados). Esto se denomina "aprendizaje por transferencia".

En resumen, para muchos problemas se puede empezar con una o dos capas ocultas y la red neuronal funcionará bien. Para problemas más complejos, podemos aumentar el número de capas ocultas hasta que el conjunto de entrenamiento empiece a sobreajustarse. Las tareas muy complejas, como la clasificación de imágenes grandes o el reconocimiento del discurso, requieren por lo general redes con decenas de capas (incluso cientos, pero no capas completamente conectadas, como veremos en un futuro) y necesitan una cantidad enorme de datos de entrenamiento. Es raro tener que entrenar ese tipo de redes desde cero: es mucho más habitual reutilizar partes de una red actualizada preentrenada que realice una tarea similar. Entonces el entrenamiento será mucho más rápido y requerirá muchos menos datos.

### NÚMERO DE NEURONAS POR CAPAS

El número de neuronas de las capas de entrada y salida está determinado por el tipo de entrada y salida que requiera la tarea:

* **Capa de entrada**: el número de neuronas en la capa de entrada debe ser igual al número de características o dimensiones en tus datos.
* **Capas ocultas**: la elección del número de neuronas a menudo implica experimentación. En general, se suele comenzar con un número moderado de neuronas y ajustar según sea necesario.
* **Capa de salida**: el número de neuronas en la capa de salida depende del tipo de tarea:
  + Para problemas de clasificación binaria, se suele utilizar una sola neurona con una función de activación sigmoide.
  + Para problemas de clasificación multiclase, se utilizan tantas neuronas como clases, con una función de activación softmax.
  + Para problemas de regresión, se utiliza una sola neurona sin función de activación específica.

Por ejemplo, la tarea MNIST (clasificar imágenes de números escritos a mano) requiere 28 x 28 = 784 entradas y 10 neuronas de salida (clasifica 10 números).

En cuanto a las capas ocultas, lo habitual era ajustar el tamaño para formar una pirámide, con menos neuronas en cada capa, con la idea de que muchas características de bajo nivel pueden fusionarse en muchas menos características de alto nivel. Una red neuronal típica para MNIST podría tener 3 capas ocultas, la primera con 300 neuronas, la segunda con 200 y la tercera con 100. Sin embargo, esta práctica se ha abandonado porque parece que usar el mismo número de neuronas en todas las capas ocultas funciona igual de bien, o incluso mejor, en la mayoría de los casos; además, solo hay que ajustar un hiperparámetro, en lugar de uno por capa. Dicho esto, dependiendo del conjunto de datos, a veces puede ayudar hacer que la primera capa oculta sea más grande que las demás.

Al igual que con el número de capas, puedes podemos a aumentar el de neuronas gradualmente hasta que la red empiece a sobreajustar. Como alternativa, podemos probar a construir un modelo con algunas capas y neuronas más de las que necesitamos realmente y luego usar la detención temprana u otras técnicas de regularización para evitar demasiado sobreajuste.

### TASA DE APRENDIZAJE, TAMAÑO DE LOTE Y OTROS HIPERPARÁMETROS

Veamos otros hiperparámetros que se pueden configurar:

* **Tasa de aprendizaje (learning\_rate)**: es seguramente el hiperparámetro más importante. Una forma de encontrar una buena tasa de aprendizaje es entrenar el modelo durante cientos de iteraciones, empezando con una tasa de aprendizaje muy baja (por ejemplo, 10-5) y aumentándola gradualmente hasta un valor muy grande (por ejemplo, 10). Es un parámetro del optimizador
* **Tamaño de lote (batch\_size)**: cuántas muestras del conjunto de datos procesa el modelo en cada paso de aprendizaje. Puede tener un impacto significativo en el rendimiento del modelo y el tiempo de entrenamiento. La principal ventaja de usar tamaños de lote grandes es que los aceleradores hardware como las GPU. Este hiperparámetros se indica en el método fit.
* **Número de iteraciones (epochs):** En la mayoría de los casos, el número de iteraciones de entrenamiento no tiene por qué ajustarse: basta con usar la detención temprana. Este hiperparámetros se indica en el método fit

La tasa de aprendizaje óptima depende de los otros hiperparámetros, sobre todo del tamaño de lote, así que, si se modifica cualquiera de ellos, hay que asegurarse de actualizar también la tasa de aprendizaje.

### UTILIZACIÓN DE RETROLLAMADAS

El método fit() acepta un argumento callbacks que permite especificar una lista de objetos a los que Keras llamará antes y después del entrenamiento, antes y después de cada repetición, incluso antes y después de procesar cada lote. Por ejemplo, la retrollamada ModelCheckpoint guarda puntos de control del modelo en intervalos regulares durante el entrenamiento, por defecto, al final de cada repetición:

*checkpoint\_cb = tf.keras.callbacks.ModelCheckpoint("my\_checkpoints.weights.h5", save\_weights\_only=True)*

*history = model.fit([...], callbacks=[checkpoint\_cb])*

Además, si usas un conjunto de validación durante el entrenamiento, puedes configurar save\_best\_only=True al crear el punto de control ModelCheckpoint. En este caso, solo se guardará el modelo cuando su rendimiento con el conjunto de validación sea el mejor hasta el momento. De ese modo, no hay que preocuparse por entrenar demasiado tiempo ni sobreajustar el conjunto de entrenamiento: simplemente restaura el último modelo guardado después del entrenamiento, que será el mejor modelo con el conjunto de validación. Esta es una forma sencilla de implementar una detención temprana, pero, en realidad, no detendrá el entrenamiento.

Otra forma es usar la retrollamada EarlyStopping. Interrumpirá el entrenamiento cuando no mida progreso en el conjunto de validación durante un número de repeticiones determinado (definido por el argumento patience) y, si configuras restore\_best\_weights=True, volverá al mejor modelo al final del entrenamiento. Puedes combinar ambas retrollamadas para guardar puntos de control de tu modelo por si falla el ordenador e interrumpir el entrenamiento pronto cuando no haya más progreso, para evitar malgastar tiempo y recursos y para reducir el sobreajuste:

*early\_stopping\_cb = tf.keras.callbacks.EarlyStopping(patience=10, restore\_best\_weights=True)*

*history = model.fit([...], callbacks=[checkpoint\_cb, early\_stopping\_cb])*

El número de repeticiones se puede configurar en un valor más alto, ya que el entrenamiento se detendrá automáticamente cuando ya no haya más progreso (solo asegúrate de que la tasa de aprendizaje no es demasiado pequeña o, de lo contrario, seguirá haciendo un progreso lento hasta el final). La retrollamada EarlyStopping almacenará los pesos del mejor modelo en la RAM y los restaurará por ti al final del entrenamiento

Hay otras muchas retrollamadas disponibles en el paquete tf.keras.callbacks (<https://keras.io/api/callbacks>).

### ARQUITECTURA TÍPICA DE UNA RED NEURONAL DE REGRESIÓN (SENCILLA)

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

### ARQUITECTURA TÍPICA DE UNA RED NEURONAL DE clasificación (SENCILLA)

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## GUARDAR Y RESTAURAR UN MODELO

### GUARDAR EL MODELO

Guardar el modelo entero

*model.save("my\_keras\_model.keras")*

Guardar solo los pesos

*model.save\_weights("my\_weights.weights.h5")*

### RESTAURAR EL MODELO

Restaurar el modelo entero

*model = tf.keras.models.load\_model("my\_keras\_model.keras")*

Cargar los datos

*model.load\_weights("Resultados/my\_weights.weights.h5")*

## EXPERIMENTACIÓN CON TENSORFLOW PLAYGROUND

[Jugando con Redes Neuronales - Parte 2.5](https://youtu.be/FVozZVUNOOA?si=dLc9oTrpFvbkOAfc) (19:47min)

[TensorFlow Playground](https://playground.tensorflow.org/) es una aplicación web de visualización interactiva, escrita en JavaScript, que nos permite simular redes neuronales densamente conectadas que se ejecutan en nuestro navegador y ver los resultados en tiempo real.

Permite añadir hasta 6 capas internas con hasta 8 neuronas por capa. Al entrenar la red neuronal, vemos si lo estamos consiguiendo o no por la métrica de “Training loss”, es decir, por la función de pérdida para los datos de entrenamiento. Posteriormente, para comprobar que el modelo generaliza, se debe conseguir también minimizar la “Test loss”, es decir, el error calculado por la función de pérdida para los datos de test.

Prueba los siguientes cambios:

1. Prueba a entrenar la red predeterminada haciendo clic en el botón Run (arriba, a la izquierda). Observa cómo enseguida encuentra una solución para la tarea de clasificación. Las neuronas de la primera capa oculta han aprendido patrones sencillos, mientras que las neuronas de la segunda capa oculta han aprendido a combinar los patrones sencillos de la primera en patrones más complejos. En general, cuantas más capas hay, más complejo puede ser el modelo.
2. Las funciones de activación. Prueba a sustituir la función de activación tanh por una ReLU y entrena la red de nuevo. Observa que encuentra una solución más deprisa todavía, pero esta vez los límites son lineales. Esto se debe a la forma de la función ReLU.
3. El riesgo de mínimos locales. Modifica la arquitectura de la red para tener una sola capa oculta con tres neuronas. Entrénala varias veces (para reiniciar los pesos de la red, haz clic en el botón Reset). Observa que el tiempo de entrenamiento cambia mucho y, a veces, incluso se atasca en un mínimo local.
4. Lo que pasa cuando las redes neuronales son demasiado pequeñas. Quita una neurona para dejar solo dos. Observa que ahora la red neuronal es incapaz de encontrar una buena solución, aunque pruebes varias veces. El modelo no tiene suficientes parámetros y subajusta sistemáticamente el conjunto de entrenamiento.
5. Lo que pasa cuando las redes neuronales son lo bastante grandes. Configura el número de neuronas en ocho y entrena la red varias veces. Observa que ahora es consistentemente rápida y no se atasca nunca. Esto subraya un hallazgo importante en la teoría de las redes neuronales: las redes neuronales grandes rara vez se atascan en mínimos locales y, si lo hacen, estos óptimos locales son a menudo casi tan buenos como el óptimo global. Sin embargo, pueden seguir atascándose en mesetas largas durante mucho tiempo.
6. El riesgo de desvanecimiento del gradiente en redes profundas. Selecciona el conjunto de datos espiral (el de la esquina inferior derecha de la sección DATA), y cambia la arquitectura de la red para tener cuatro capas ocultas con ocho neuronas cada una. Observa que el entrenamiento tarda mucho más y a menudo se atasca en mesetas durante periodos de tiempo largos. Observa también que las neuronas de las capas más altas (a la derecha) tienden a evolucionar más deprisa que las neuronas de las capas inferiores (a la izquierda). Este problema, llamado "desvanecimiento del gradiente", se puede paliar con una mejor inicialización de pesos y otras técnicas, mejores optimizadores (como AdaGrad o Adam)…
7. Ve más lejos. Tomate un tiempo para jugar con otros parámetros y hacerte una idea de lo que hacen, consigue una comprensión intuitiva de las redes neuronales.