UNIDAD 5

PROGRAMACIÓN DE REDES NEURONALES ARTIFICIALES

Programación de Inteligencia Artificial

Curso de Especialización en Inteligencia Artificial y Big Data

CHATGPT prompt: Genera una imagen de un cerebro humano trabajando como una red neuronal en arte digital

Carlos M. Abrisqueta Valcárcel

IES Ingeniero de la Cierva 2024/25

ÍNDICE

1. INTRODUCCIÓN A LAS RNAs

Las **Redes Neuronales Artificiales** (RNA), también conocidos como perceptrones multicapa, son una rama avanzada de la inteligencia artificial, inspiradas en la estructura y funcionamiento del cerebro humano. Su diseño se basa en una aproximación computacional para emular el modo en que las redes neuronales biológicas procesan la información. Las RNA consisten en unidades de procesamiento llamadas neuronas artificiales, organizadas en capas.

Una neurona artificial típica recibe señales de entrada (input), las cuales son procesadas mediante una función de suma ponderada. Cada entrada tiene asociado un peso, reflejando la importancia relativa de esa entrada. La suma ponderada es luego transformada por una función de activación, que puede ser lineal o no lineal, como la función sigmoide, la función tangente hiperbólica o la función de unidad lineal rectificada (ReLU).

Las RNA se clasifican en diferentes tipos, según su arquitectura y el método de aprendizaje. Las redes *Feedforward*, donde las conexiones entre las neuronas no forman ciclos, son las más simples. En contraste, las *Redes Neuronales Recurrentes* (RNN) permiten conexiones hacia atrás, lo que las hace adecuadas para procesar secuencias temporales de datos.

El entrenamiento de una RNA se realiza a través de algoritmos de aprendizaje, siendo el más común el de *retropropagación* (backpropagation) junto con el descenso de gradiente. Durante el entrenamiento, los pesos de las conexiones entre neuronas se ajustan iterativamente con el objetivo de minimizar una función de pérdida, que mide la diferencia entre la salida de la red y el resultado deseado.

Las capas ocultas en una red (aquellas entre la entrada y la salida) permiten el aprendizaje de características jerárquicas en los datos. En el contexto de **Redes Neuronales Convolucionales** (CNN), especializadas en procesar datos con una topología de rejilla (como imágenes), estas capas aprenden a reconocer patrones espaciales.

La capacidad de una RNA para aprender y modelar relaciones no lineales complejas en grandes conjuntos de datos las hace poderosas para una variedad de aplicaciones, desde el reconocimiento de voz e imagen hasta la predicción de series temporales y el procesamiento del lenguaje natural. Sin embargo, su eficacia depende en gran medida de la arquitectura de la red, la calidad y cantidad de los datos de entrenamiento, y la configuración de sus hiperparámetros.

1.1. LIMITACIONES DEL PERCEPTRÓN

Aunque el perceptrón es un modelo fundamental en el aprendizaje automático, posee limitaciones significativas, especialmente en su capacidad para resolver ciertos tipos de problemas:

- **Separabilidad Lineal**: El perceptrón es eficaz para clasificar conjuntos de datos que son linealmente separables. Sin embargo, su eficacia disminuye significativamente con conjuntos de datos que no son linealmente separables. Esto se debe a que el perceptrón utiliza una función de decisión lineal, lo que significa que solo puede trazar una línea recta (o un hiperplano en dimensiones superiores) para separar las clases.

- **Problema del XOR**: Un ejemplo clásico de la limitación del perceptrón es su incapacidad para resolver el problema del XOR (exclusive OR). En este caso, no existe un hiperplano

lineal que pueda separar efectivamente las clases de salida para una operación XOR. Esta limitación es un ejemplo de cómo el perceptrón no puede manejar patrones de datos no lineales.

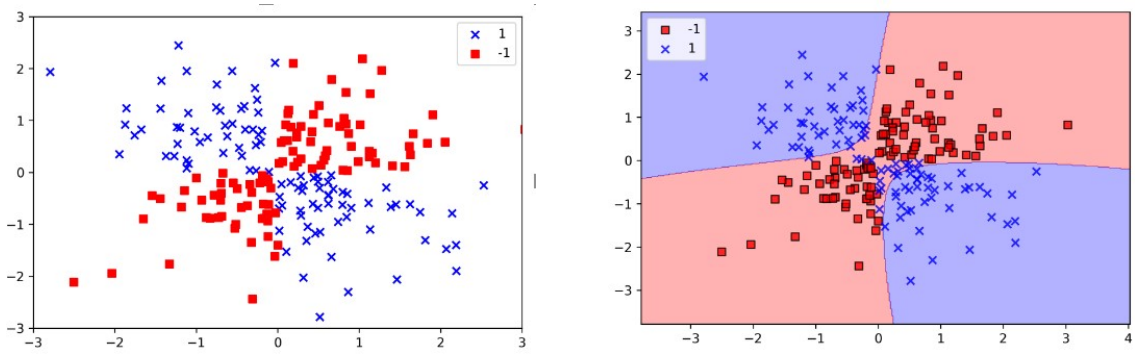

Figura 1: El perceptrón no puede separar las dos clases del XOR

- **Modelado de Relaciones Complejas**: El perceptrón, al ser un modelo lineal simple, no es capaz de capturar la complejidad y las relaciones no lineales presentes en muchos conjuntos de datos reales. Esto limita su uso en problemas más avanzados como el reconocimiento de patrones complejos, el procesamiento del lenguaje natural y tareas de visión por computadora, donde se requieren modelos capaces de aprender y representar relaciones no lineales.

Estas limitaciones llevaron al desarrollo de modelos más avanzados, como las **Redes Neuronales Artificiales**, que superan estas restricciones incorporando múltiples capas y funciones de activación no lineales, permitiendo así el modelado de relaciones más complejas y abstractas en los datos.

1.2. DEL PERCEPTRÓN A LAS RNAs

Las Redes Neuronales Artificiales (RNA) representan una evolución significativa en el campo del aprendizaje automático, superando muchas de las limitaciones inherentes al perceptrón simple. A diferencia de este, las RNA se caracterizan por su estructura en capas y su capacidad para modelar relaciones no lineales en los datos. A continuación se detallan algunos aspectos clave de las RNA:

- **Estructura en Capas**: Una RNA típica consta de una capa de entrada, varias capas ocultas y una capa de salida. Cada capa contiene un número de neuronas, que son análogas al perceptrón pero con capacidades extendidas. Estas capas permiten a la RNA procesar información en una forma más compleja y abstracta que un simple perceptrón.

- **Funciones de Activación No Lineales**: A diferencia del perceptrón, que utiliza funciones de activación lineales o escalonadas, las RNA emplean funciones de activación no lineales como la función sigmoide, tanh o ReLU (Rectified Linear Unit). Estas funciones no lineales son esenciales para que las RNA puedan aprender y modelar complejas relaciones no lineales en los datos.

- **Aprendizaje de Características Jerárquicas**: Las capas ocultas en las RNA permiten el aprendizaje de características jerárquicas y representaciones más profundas de los datos. Cada capa oculta puede aprender a detectar características cada vez más abstractas y

complejas, lo cual es fundamental en tareas como el reconocimiento de imágenes y el procesamiento del lenguaje natural.

- **Flexibilidad y Adaptabilidad**: La arquitectura de las RNA les confiere una gran flexibilidad, permitiéndoles adaptarse a una amplia gama de problemas y tipos de datos. Esta adaptabilidad se ve reforzada por la capacidad de ajustar la cantidad de capas y neuronas, lo que afecta la complejidad y capacidad del modelo.

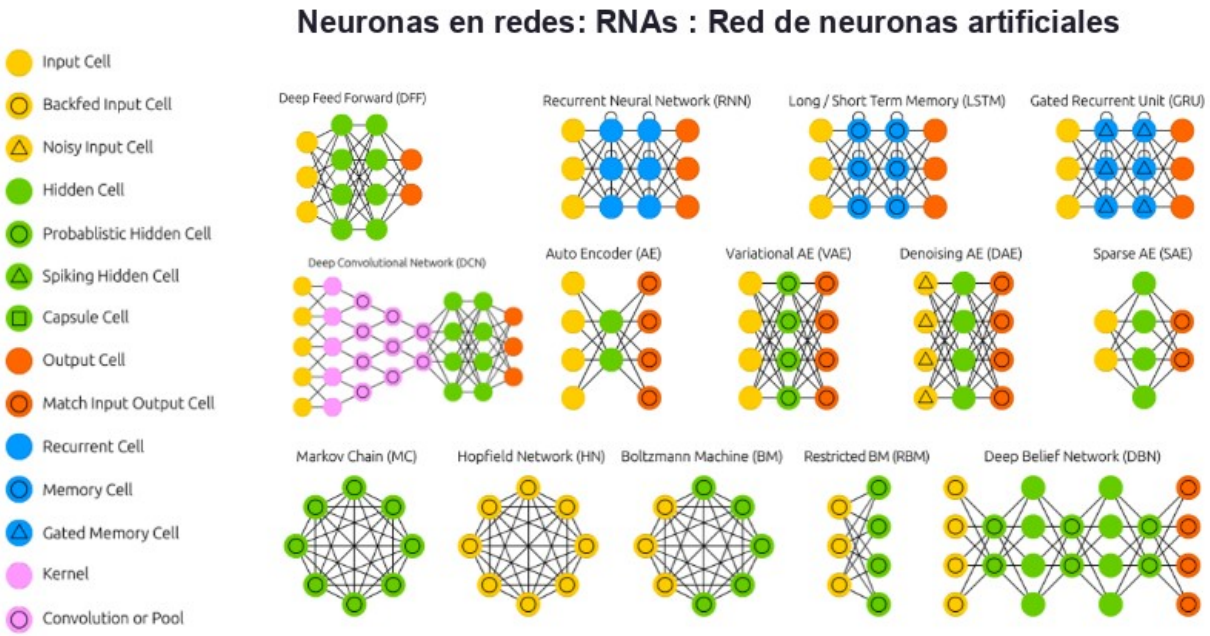

Figura 2: Ejemplos de tipos de perceptrones multicapa o RNAs

2. APRENDIZAJE EN REDES NEURONALES ARTIFICIALES

El aprendizaje en Redes Neuronales Artificiales (RNA) es un proceso más sofisticado y potente comparado con el aprendizaje en el perceptrón simple. Este proceso involucra no solo el ajuste de los pesos y sesgos de la red, sino también el manejo de estructuras de múltiples capas y la utilización de algoritmos avanzados. Los puntos clave en el aprendizaje de las RNA son:

- **Proceso Iterativo de Ajuste**: Similar al perceptrón, el aprendizaje en una RNA implica ajustar iterativamente los pesos y sesgos. Sin embargo, debido a la presencia de múltiples capas, este proceso es más complejo y se realiza a través de algoritmos como el de retropropagación (backpropagation).

- **Uso de Backpropagation o Retropropagación y Descenso de Gradiente**: La retropropagación es un método clave en el entrenamiento de RNA. Permite calcular el gradiente de la función de pérdida con respecto a cada peso y sesgo en la red, utilizando la regla de la cadena del cálculo diferencial. Este gradiente se utiliza luego en algoritmos de optimización como el descenso de gradiente para actualizar los pesos y sesgos.

- **Aprendizaje de Características Complejas**: A través de las múltiples capas y la no linealidad introducida por las funciones de activación, las RNA son capaces de aprender

características complejas y patrones en los datos. Cada capa sucesiva puede construir y refinar representaciones abstractas derivadas de los datos de entrada.

- **Prevención del Sobreajuste**: Dado que las RNA tienen una capacidad de modelado más potente, es crucial implementar técnicas para prevenir el sobreajuste, como la regularización, el dropout, o la validación cruzada. Estas técnicas ayudan a asegurar que la RNA generalice bien a nuevos datos, no solo al conjunto de entrenamiento.

- **Ajuste de Hiperparámetros**: El aprendizaje en las RNA también involucra el ajuste de varios hiperparámetros, como la tasa de aprendizaje, el número de capas y neuronas, y el tipo de funciones de activación. La elección adecuada de estos hiperparámetros es crucial para el rendimiento óptimo de la red.

2.1. EJEMPLOS Y APLICACIONES DE REDES NEURONALES ARTIFICIALES

Las Redes Neuronales Artificiales (RNA) han encontrado aplicaciones en una amplia gama de campos, demostrando su capacidad para abordar tareas que son considerablemente complejas para los modelos más simples como el perceptrón. Algunos ejemplos y aplicaciones destacadas de las RNA incluyen:

- **Reconocimiento de Imágenes**: Las RNAs, especialmente las Redes Neuronales Convolucionales (CNN), han revolucionado el campo del reconocimiento y procesamiento de imágenes. Son capaces de identificar patrones, objetos y rostros con un alto grado de precisión, siendo fundamentales en aplicaciones como el diagnóstico médico por imagen y los sistemas de vigilancia.

- **Procesamiento del Lenguaje Natural** (PLN): Las Redes Neuronales Recurrentes (RNN) y las transformadoras han cambiado radicalmente la manera en que las máquinas entienden y generan lenguaje humano. Se utilizan en traducción automática, generación de texto, sistemas de respuesta a preguntas y asistentes virtuales.

- **Predicción de Series Temporales**: Las RNA son herramientas poderosas para analizar y predecir datos temporales, como los precios de las acciones, el clima, o las tendencias de consumo. Su capacidad para aprender patrones temporales complejos las hace ideales para estas tareas.

- **Juegos y Simulaciones**: Las RNA también han mostrado un rendimiento sobresaliente en el aprendizaje y la ejecución de estrategias en juegos complejos, como el Go o el ajedrez, superando a los campeones humanos y abriendo nuevas fronteras en la inteligencia artificial.

- **Vehículos Autónomos**: Las RNA forman parte integral de los sistemas de conducción autónoma, donde se utilizan para interpretar y tomar decisiones en tiempo real basadas en un flujo constante de datos sensoriales, contribuyendo a la seguridad y eficiencia de estos sistemas.

Estos ejemplos ilustran la versatilidad y potencia de las RNA en el aprendizaje automático y la inteligencia artificial. Su capacidad para aprender de grandes cantidades de datos y modelar complejidades no lineales les permite abordar y resolver problemas que antes se consideraban

intratables. La continua investigación y desarrollo en este campo prometen aún más aplicaciones innovadoras en el futuro.

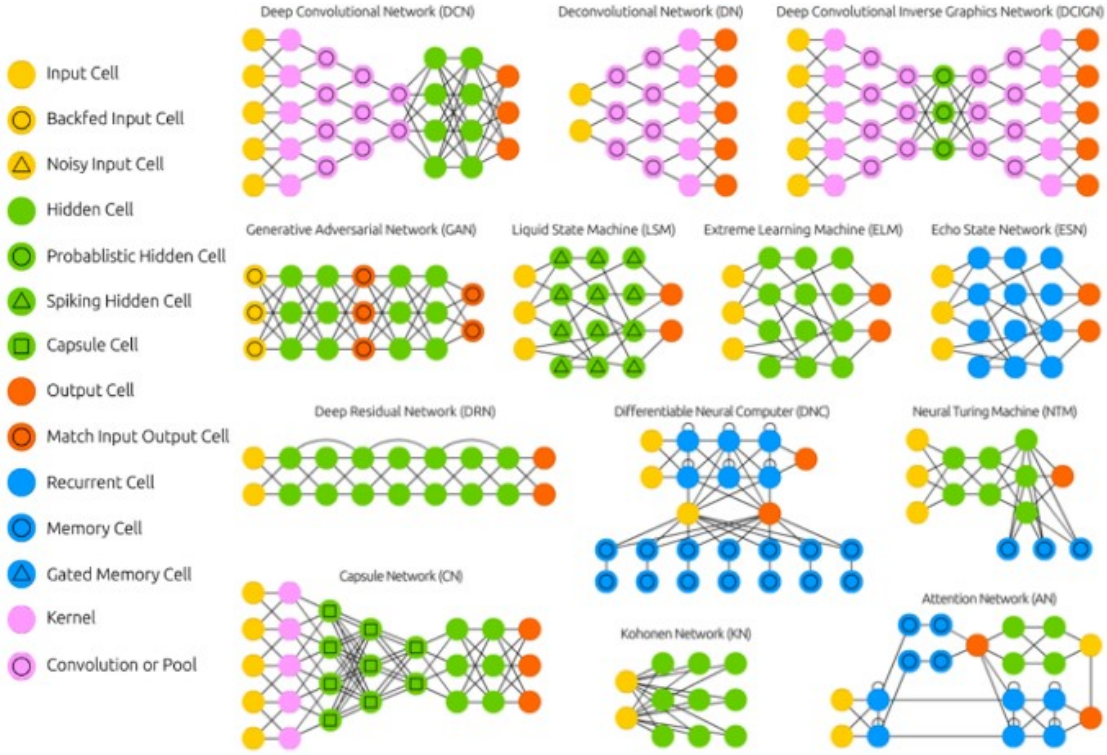

Figura 3: Ejemplos de tipos de perceptrones multicapa o RNAs complejas

2.2. ALGORITMO DE BACKPROPAGATION

El algoritmo de *backpropagation*, o *retropropagación*, es el corazón del entrenamiento de Redes Neuronales Artificiales, particularmente en arquitecturas de múltiples capas. Este algoritmo se fundamenta en la regla de la cadena del cálculo diferencial y se emplea para computar el gradiente de la función de pérdida con respecto a los pesos de la red.

En el contexto de una red neuronal, la función de pérdida L mide la discrepancia entre las salidas predichas de la red $\hat{y}$ y los valores reales y. El objetivo del entrenamiento es minimizar esta función de pérdida ajustando los pesos W y los sesgos b de la red.

El algoritmo de backpropagation se ejecuta en dos fases: propagación hacia adelante y propagación hacia atrás:

1) **Propagación hacia adelante**: En esta etapa, se calculan las activaciones a través de las sucesivas capas de la red. Para una capa l, la salida $a^{[l]}$ se calcula como $a^{[l]} = f\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$, donde f es la función de activación.

2) **Propagación hacia atrás**: Esta fase comienza calculando el gradiente de la función de pérdida con respecto a la salida de la red. Luego, se calcula el gradiente de la función de pérdida con respecto a los pesos y sesgos de cada capa, retrocediendo desde la última hasta la primera capa. La regla de la cadena es fundamental aquí para relacionar estos gradientes a través de las capas.

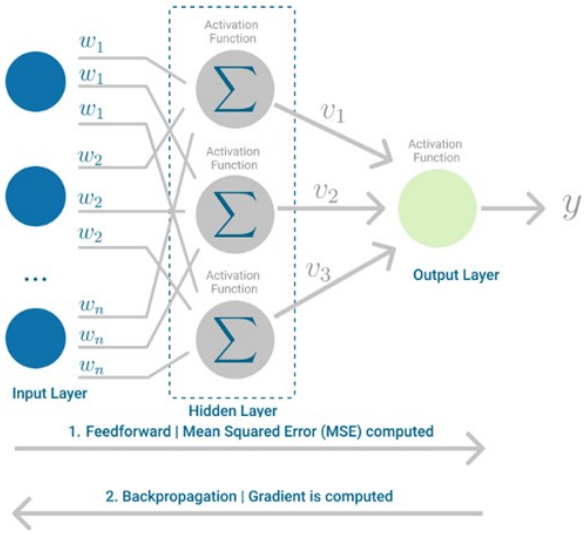

Figura 4: Aprendizaje en RNAs

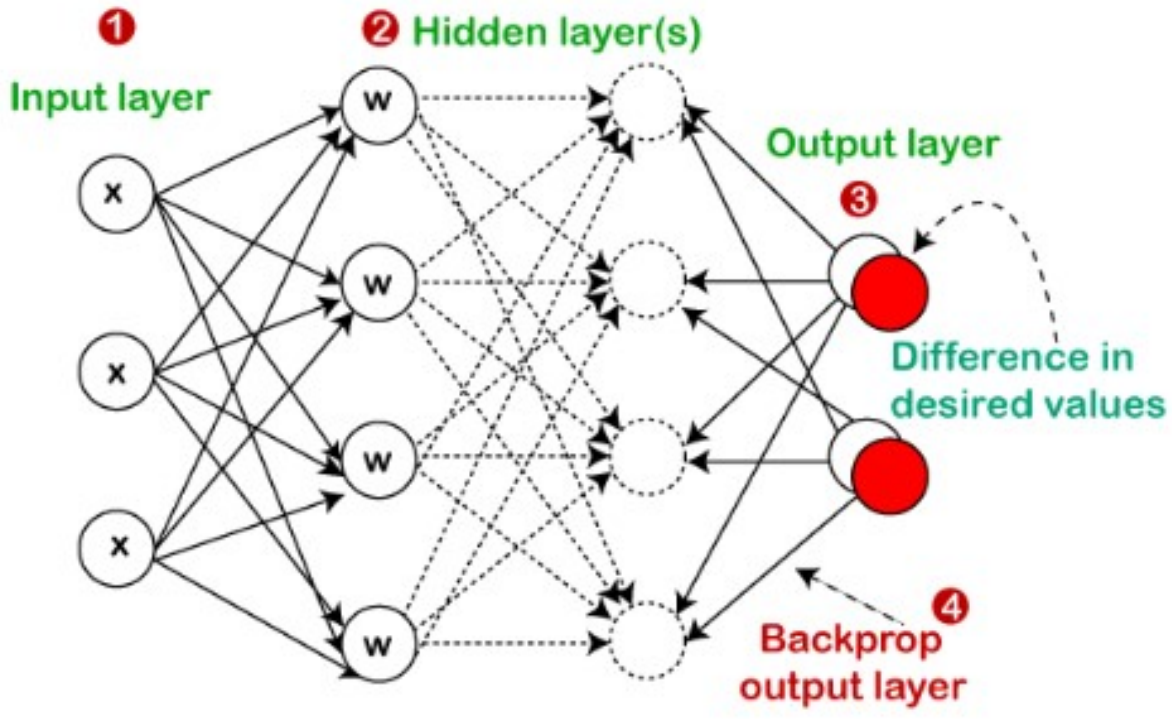

Figura 5: Aprendizaje en RNAs. La regla de la cadena

El ajuste de los pesos se realiza típicamente utilizando el método del descenso de gradiente, donde los pesos se actualizan en la dirección opuesta al gradiente de la función de pérdida.

2.3. BACKPROPAGATION EN PYTHON

Se ilustra un ejemplo simplificado en Python para demostrar cómo se implementa este algoritmo en una red neuronal con una capa oculta. La red se entrena para aprender la función XOR.

```
import numpy as np
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

input_data = np.array([[0,0], [0,1], [1,0], [1,1]])
expected_output = np.array([[0], [1], [1], [0]])
epochs = 10000
lr = 0.1
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2, 2, 1
hidden_weights = np.random.uniform(
    size=(inputLayerNeurons, hiddenLayerNeurons))
hidden_bias =np.random.uniform(
    size=(1, hiddenLayerNeurons))
output_weights = np.random.uniform(
    size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(
    size=(1,outputLayerNeurons))

for _ in range(epochs):
    hidden_layer_activation = np.dot(input_data, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer*sigmoid_derivative(hidden_layer_output)

    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
    hidden_weights += input_data.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("Salida final después del entrenamiento: \n", predicted_output)
print("Pesos finales de la capa oculta: \n", hidden_weights)
print("Sesgos finales de la capa oculta: \n", hidden_bias)
print("Pesos finales de la capa de salida: \n", output_weights)
print("Sesgos finales de la capa de salida: \n", output_bias)

>SALIDA:
Salida final después del entrenamiento:
    [[0.07053248]
```

```
    [0.93350693]
    [0.93333741]
    [0.07293866]]

Pesos finales de la capa oculta:
    [[3.50397237 5.64597314]
    [3.51251273 5.69124863]]

Sesgos finales de la capa oculta:
    [[-5.360207   -2.30794385]]

Pesos finales de la capa de salida:
    [[-7.69915563]
    [ 7.10937871]]

Sesgos finales de la capa de salida:
    [[-3.18583246]]
```

Listing 1: Algoritmo de Backpropagation en Python

Los pasos clave son:

1) **Inicialización**: Se establecen pesos y sesgos aleatorios para la capa oculta y la capa de salida.

2) **Propagación hacia adelante**: Calculo de las activaciones de las capas oculta y de salida.

3) **Cálculo del error**: Diferencia entre la salida esperada y la salida predicha por la red.

4) **Propagación hacia atrás**: Cálculo de los gradientes de la función de pérdida respecto a la salida y propagación de estos gradientes hacia atrás.

5) **Actualización de pesos y sesgos**: Ajuste de los pesos y sesgos en la dirección opuesta al gradiente.

Tras múltiples iteraciones, los pesos y sesgos se ajustan para que la red pueda predecir con precisión la función XOR.

2.4. EL TAMAÑO DE LAS REDES NEURONALES ARTIFICIALES

El tamaño de una red neuronal artificial (RNA) puede variar ampliamente, desde unas pocas decenas de parámetros hasta muchos miles de millones, dependiendo de su arquitectura y del problema que esté diseñada para resolver. En la figura 6, se ilustra el rápido crecimiento en el tamaño de los modelos de lenguaje más destacados a lo largo del tiempo, comenzando con modelos como ELMO, que tenía alrededor de 94 millones de parámetros, hasta llegar a modelos muy grandes como el Megatron-Turing NLG con 530 mil millones de parámetros.

Este incremento en el tamaño de los modelos de RNA está impulsado por la búsqueda de un mejor rendimiento en tareas como el procesamiento del lenguaje natural, la generación de texto, la traducción automática y otras tareas de inteligencia artificial. Los modelos más grandes a menudo

pueden capturar matices más finos del lenguaje y tener un mejor rendimiento en tareas específicas, aunque también requieren mucho más poder computacional para ser entrenados y operados.

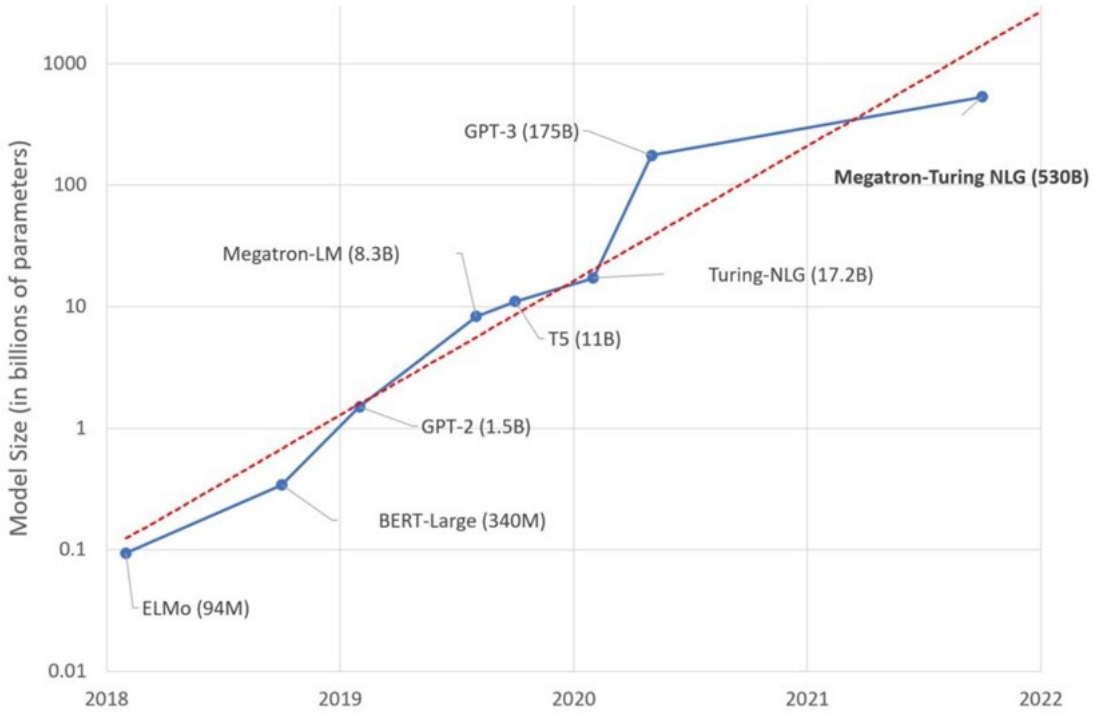

Figura 6: Tamaño de las RNAs

El tamaño de una RNA es solo uno de los factores que determina su efectividad y eficiencia. La calidad de los datos de entrenamiento, la arquitectura de la red y la eficiencia de los algoritmos de optimización también juegan papeles críticos. Además, a medida que los modelos se vuelven más grandes, la investigación también se centra en cómo hacer que estos modelos sean más eficientes energéticamente y cómo reducir su huella de carbono.

3. TENSORFLOW

TensorFlow, desarrollado por **Google Brain**, es una biblioteca de código abierto para computación numérica y aprendizaje automático, particularmente conocida por su eficacia en la construcción y entrenamiento de Redes Neuronales Artificiales. Su nombre proviene del flujo de tensores, que son estructuras de datos multidimensionales utilizadas en estos procesos. Características clave de TensorFlow incluyen:

- **Grafo de Cómputo**: En TensorFlow, los modelos y algoritmos se representan como un grafo de cómputo, donde los nodos representan operaciones matemáticas y los bordes representan tensores.

- **Ejecución Eficiente**: TensorFlow permite la distribución del procesamiento en múltiples CPUs, GPUs y TPUs, optimizando así el rendimiento en grandes conjuntos de datos y modelos complejos.

- **Flexibilidad**: Permite la creación de modelos personalizados y avanzados, ofreciendo al mismo tiempo herramientas y funciones pre-construidas para facilitar el desarrollo.

- **TensorBoard**: Una herramienta de visualización integrada que permite monitorear y visualizar diversos aspectos del entrenamiento de modelos.

Figura 7: Logo de tensorflow

3.1. TENSORES EN TENSORFLOW: CONSTANTES Y VARIABLES

TensorFlow utiliza estructuras de datos fundamentales llamadas tensores. Los tensores son contenedores multidimensionales de datos, generalmente de tipo numérico, que se utilizan para representar datos en cálculos de aprendizaje automático.

3.1.1. Tipos de Tensores: Constantes y Variables

En TensorFlow, los tensores se pueden clasificar principalmente en dos categorías: **constantes** y **variables**. Las constantes, como su nombre indica, son tensores de valor fijo que no cambian durante la ejecución del programa, mientras que las variables son tensores cuyos valores pueden cambiar durante la ejecución.

Ejemplos en Python

Para ilustrar, a continuación se presentan ejemplos de cómo crear diferentes tipos de tensores en TensorFlow usando Python:

- **Escalar**: Un tensor de rango 0, es decir, un único número.

```
import tensorflow as tf
escalar = tf.constant(4)
```

- **Vector**: Un tensor de rango 1, que representa una lista de valores.

```
vector = tf.constant([1, 2, 3])
```

- **Matriz**: Un tensor de rango 2, similar a una matriz bidimensional.

```
matriz = tf.constant([[1, 2, 3], [4, 5, 6]])
```

- **Tensor Multidimensional**: Un tensor de rango superior, como un tensor 3D o más.

```
tensor_3d = tf.constant([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

- **Escalar variable**: Un tensor variable de rango 0.

```
escalar = tf.Variable(4.0)
```

- **Vector variable**: Un tensor variable de rango 1, que representa una lista de valores.

```
vector = tf.Variable([1.0, 2.0, 3.0])
```

- **Matriz variable**: Un tensor variable de rango 2, similar a una matriz bidimensional.

```
matriz = tf.Variable([[1.0, 2.0],
                      [3.0, 4.0]])
```

- **Tensor variable Multidimensional**: Un tensor de rango superior, como un tensor 3D o más.

```
tensor = tf.Variable([[[1.0, 2.0], [3.0, 4.0]],
                      [[5.0, 6.0], [7.0, 8.0]]])
```

La diferencia entre tf.constant y tf.Variable es que los valores de un tensor creado con tf.constant no se pueden modificar. Los tensores variables, se pueden modificar sus valores a través del método assign:

```
# Crear el mismo tensor con tf.Variable() y tf.constant()
tensor_modificable = tf.Variable([10, 7])
tensor_constante = tf.constant([10, 7])

# producirá error, pero se puede cambiar con el método assign
tensor_modificable[0] = 7

# No producirá error
tensor_modificable[0].assign(7)

# Producirá un error (no se puede cambiar un tf.constant())
tensor_constante[0].assign(7)
```

3.1.2. Obteniendo toda la información de un tensor

Mientras que el tamaño de un tensor es el número de elementos que contiene, el rango de un tensor se refiere a su dimensión y la forma describe el número de elementos en cada dimensión. Por otro lado, el concepto de ejes o axis, se refiere a los elementos de una determinada dimensión. TensorFlow proporciona métodos para obtener fácilmente estas propiedades de un tensor:

- Para obtener el rango de un tensor, se utiliza el método tf.rank o el atributo ndim:

```
rango = tf.rank(tensor_3d)
# otra forma rango = tensor_3d.ndim
```

- La forma de un tensor se puede obtener a través de la propiedad shape:

```
forma = tensor_3d.shape
```

Podemos obtener toda la información del tensor de la siguiente manera:

```
# Con un tensor de 4 dimensiones
rank_4_tensor = tf.zeros([2, 3, 4, 5])

print("tipo de datos de cada elemento:", rank_4_tensor.dtype)
print("número de dimensiones (rank):", rank_4_tensor.ndim)
print("Forma del tensor:", rank_4_tensor.shape)
print("Elementos del eje 0 del tensor:", rank_4_tensor.shape[0])
print("Elementos del último eje del tensor:", rank_4_tensor.shape[-1])
print("Número total de elementos (2*3*4*5):", tf.size(rank_4_tensor).numpy())

#salida:
tipo de datos de cada elemento: <dtype: 'float32'>
número de dimensiones (rank): 4
Forma del tensor: (2, 3, 4, 5)
Elementos del eje 0 del tensor: 2
Elementos del último eje del tensor: 5
Número total de elementos (2*3*4*5): 120
```

3.2. GENERACIÓN Y REPRODUCCIÓN DE TENSORES ALEATORIOS

Para generar tensores aleatorios en TensorFlow, se pueden utilizar funciones específicas de la biblioteca tensorflow.random. Los números generados por estas funciones son pseudoaleatorios, lo que significa que son el resultado de algoritmos determinísticos y pueden ser reproducidos exactamente utilizando una semilla (seed). La semilla asegura que el generador de números aleatorios produzca la misma secuencia de números para la misma semilla, permitiendo la reproducibilidad de los resultados en experimentos y pruebas. A continuación, se presenta un ejemplo de cómo generar tensores aleatorios con y sin semilla.

```
import tensorflow as tf

# Crear un tensor aleatorio sin semilla
tensor_aleatorio = tf.random.normal([3, 2])

# Establecer una semilla
tf.random.set_seed(42)

# Crear un tensor aleatorio con semilla
tensor_aleatorio_con_semilla = tf.random.normal([3, 2])
otro_tensor_con_la_misma_semilla = tf.random.normal([3, 2])
# los dos tensores anteriores serán idénticos
```

zeros & ones

TensorFlow proporciona varias funciones útiles para la inicialización de tensores. Entre ellas, *tf.zeros* y *tf.ones* son ampliamente utilizadas para crear tensores llenos de ceros o unos, respectivamente. Estos tensores pueden ser de cualquier forma especificada por el usuario.

- **tf.zeros**: Esta función crea un tensor de la forma especificada, donde todos los elementos son ceros. Por ejemplo, tf.zeros([2, 3]) creará un tensor de forma 2x3 (dos filas y tres columnas) lleno de ceros.

- **tf.ones**: Similar a tf.zeros, pero en lugar de ceros, llena el tensor con unos. Así, tf.ones([2, 3]) resultará en un tensor de forma 2x3 lleno de unos.

A continuación, se muestran ejemplos de cómo utilizar estas funciones:

```
import tensorflow as tf

# Crear un tensor 2x3 lleno de ceros
tensor_ceros = tf.zeros([2, 3])
print("Tensor de ceros:")
print(tensor_ceros)

# Crear un tensor 2x3 lleno de unos
tensor_unos = tf.ones([2, 3])
print("Tensor de unos:")
print(tensor_unos)
```

En estos ejemplos, se crea primero un tensor de tamaño 2x3 lleno de ceros y luego otro tensor de igual tamaño lleno de unos. Estas funciones son especialmente útiles para la inicialización de tensores.

3.3. MEZCLAR LOS ELEMENTOS DE UN TENSOR

TensorFlow ofrece la función *tf.shuffle* que reordena aleatoriamente los elementos de un tensor a lo largo de su primera dimensión. El reordenamiento aleatorio de los datos es una práctica común en aprendizaje automático para garantizar que el modelo no aprenda patrones no deseados relacionados con el orden específico de los datos de entrenamiento.

En el contexto de las RNA, barajar los datos antes del entrenamiento ayuda a prevenir que el modelo se sobreajuste a un orden particular de los datos. Además, garantiza que cada batch de datos (si se utiliza entrenamiento por lotes) tenga una variedad representativa de ejemplos, lo que puede mejorar la generalización del modelo.

```
import tensorflow as tf
# Crear un tensor ejemplo
tensor_ejemplo = tf.constant([[1, 2], [3, 4], [5, 6], [7, 8]])

# Barajar el tensor
tensor_barajado = tf.random.shuffle(tensor_ejemplo)
```

En el ejemplo anterior, *tf.random.shuffle* reordena aleatoriamente las filas del tensor ejemplo. Es importante tener en cuenta que cada ejecución de *tf.random.shuffle* puede producir un orden diferente, a menos que se establezca una semilla para el generador de números aleatorios.

3.4. TRANSFORMACIÓN DE ARRAYS DE NUMPY A TENSORES

La conversión de arrays de *NumPy* a tensores de *TensorFlow* es una práctica común en el procesamiento de datos para el aprendizaje automático.

La función *tf.constant* toma un array de *NumPy* (u otros tipos de datos) y crea un tensor de *TensorFlow* con el mismo contenido y forma. Esta función es particularmente útil para convertir datos ya existentes en un formato compatible con *TensorFlow*.

```
import numpy as np
import tensorflow as tf

# Crear un array de NumPy usando np.arange
array_np = np.arange(10)
# Crea un array con los números del 0 al 9

# Convertir el array de NumPy en un tensor de TensorFlow
tensor_tf = tf.constant(array_np, dtype=tf.float32)
```

En este ejemplo, se utiliza *np.arange* para generar un array de *NumPy*. Luego, el array se convierte en un tensor de *TensorFlow* mediante *tf.constant*. Esta conversión es directa y eficiente, lo que facilita la manipulación de datos y su posterior uso en algoritmos y modelos de aprendizaje automático.

3.5. MANIPULACIÓN BÁSICA DE TENSORES

TensorFlow proporciona una variedad de operaciones para manipular tensores, incluyendo operadores aritméticos básicos, multiplicación elemento a elemento, multiplicación de matrices y el producto punto. Para encontrar patrones en los tensores, tenemos que saber manipularlos.

- **Operadores Aritméticos**: *TensorFlow* soporta operaciones aritméticas básicas como suma, resta, multiplicación y división. Estas operaciones se pueden realizar directamente entre tensores de la misma forma.

- **Método Multiply**: Para realizar multiplicaciones elemento a elemento entre tensores, se utiliza el método *tf.multiply*. Esta operación multiplica los elementos correspondientes de dos tensores.

- **Multiplicación de Matrices con *matmul***: La función *tf.matmul* se utiliza para realizar la multiplicación de matrices. Esta operación es diferente de *tf.multiply*, ya que sigue las reglas de multiplicación de matrices en lugar de la multiplicación elemento a elemento.

- **Operación Dot Product**: El producto punto entre dos tensores se puede realizar utilizando *tf.tensordot*. Esta operación es útil en muchas aplicaciones, especialmente en álgebra lineal. A continuación se presentan ejemplos de estas operaciones:

```
import tensorflow as tf

# Crear tensores de ejemplo
tensor_a = tf.constant([[1, 2], [3, 4]])
tensor_b = tf.constant([[5, 6], [7, 8]])

# Operaciones aritméticas básicas
suma = tensor_a + tensor_b
resta = tensor_a - tensor_b

# Multiplicación elemento a elemento
producto_elemento = tf.multiply(tensor_a, tensor_b)

# Multiplicación de matrices
producto_matriz = tf.matmul(tensor_a, tensor_b)

# Producto punto
producto_punto = tf.tensordot(tensor_a, tensor_b, axes=1)

print("Suma:", suma)
print("Resta:", resta)
print("Producto Elemento a Elemento:", producto_elemento)
print("Producto de Matrices:", producto_matriz)
print("Producto Punto:", producto_punto)
```

$$\begin{bmatrix} a & b \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{bmatrix}$$

Figura 8: La operación producto punto o dot product

3.5.1. El Método *Reshape* en *TensorFlow*

El método *reshape* en *TensorFlow* nos que permite cambiar la forma de un tensor. Esta operación es útil cuando necesitamos alterar las dimensiones de los datos para ajustarlos a las necesidades de diferentes estructuras de redes neuronales o algoritmos.

Vamos a ver un ejemplo simple de cómo se puede utilizar *reshape* para cambiar las dimensiones de un tensor. Este código transforma un tensor unidimensional en un tensor bidimensional de tamaño 2x3.

```
import tensorflow as tf

# Creando un tensor 1D
tensor_1d = tf.constant([1, 2, 3, 4, 5, 6])

# Cambiando la forma del tensor a 2x3
tensor_2d = tf.reshape(tensor_1d, [2, 3])
print(tensor_2d.numpy())

# [[1 2 3]
# [4 5 6]]
```

El método *reshape* también es útil para dimensiones más complejas.

```
# Creando un tensor 3D
tensor_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

# Cambiando la forma del tensor a 2x4
tensor_reshaped = tf.reshape(tensor_3d, [2, 4])

print(tensor_reshaped.numpy())
# [[1 2 3 4]
# [5 6 7 8]]
```

En este ejemplo, un tensor tridimensional se convierte en un tensor bidimensional, demostrando la flexibilidad de reshape para manipular las dimensiones de los tensores.

3.5.2. El Método *Transpose* en *TensorFlow*

El método *transpose* en *TensorFlow* nos permite cambiar el orden de las dimensiones de un tensor. Es especialmente útil en operaciones matriciales y en la reorganización de datos para el procesamiento de redes neuronales.

En este ejemplo, se muestra cómo utilizar *transpose* para cambiar las dimensiones de un tensor bidimensional.

Este código intercambia las filas y columnas del tensor, transformando su forma de 3x2 a 2x3.

```
import tensorflow as tf

# Creando un tensor 2D
tensor_2d = tf.constant([[1, 2], [3, 4], [5, 6]])

# Transponiendo el tensor
tensor_transpuesto = tf.transpose(tensor_2d)
```

```
print(tensor_transpuesto.numpy()) # Salida:
# [[1 3 5]
# [2 4 6]]
```

El método *transpose* también puede usarse con permutaciones más complejas en tensores de mayor dimensión.

```
# Creando un tensor 3D
tensor_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

# Transponiendo el tensor con una permutación específica
tensor_transpuesto = tf.transpose(tensor_3d, perm=[1, 0, 2])

print(tensor_transpuesto.numpy())
#Salida:
#[[[1 2]
#  [5 6]]
#
# [[3 4]
#  [7 8]]]
```

En este ejemplo, el tensor tridimensional es transpuesto reordenando sus ejes según la permutación especificada, lo que demuestra la flexibilidad de *transpose* en la manipulación de tensores de alta dimensión.

3.5.3. Casting de Tensores

El método *cast* se utiliza para cambiar el tipo de datos de un tensor.

```
import tensorflow as tf

# Creando un tensor de tipo float
tensor_float = tf.constant([1.8, 2.2, 3.3, 4.4])

# Cast a tensor entero
tensor_int = tf.cast(tensor_float, tf.int32)

print(tensor_int.numpy())
# Salida: [1 2 3 4]
```

3.5.4. Valor Absoluto de un Tensor

El método *abs* retorna el valor absoluto de cada elemento en el tensor.

```
# Creando un tensor con valores negativos
tensor_negativo = tf.constant([-1, -2, -3, -4])

# Valor absoluto del tensor
tensor_abs = tf.abs(tensor_negativo)
```

```
print(tensor_abs.numpy())
# Salida: [1 2 3 4]
```

3.5.5. Mínimo, Máximo, Media y Suma

Los métodos *min*, *max*, *mean* y *sum* proporcionan estadísticas básicas sobre los tensores.

```
# Creando un tensor
tensor = tf.constant([1, 2, 3, 4, 5])

# Mínimo
min_val = tf.reduce_min(tensor)

# Máximo
max_val = tf.reduce_max(tensor)

# Media
mean_val = tf.reduce_mean(tensor)

# Suma
sum_val = tf.reduce_sum(tensor)

print(f"Min: {min_val.numpy()}, Max: {max_val.numpy()}, Mean: {mean_val.numpy()
}, Sum: {sum_val.numpy()}")

# Salida: Min: 1, Max: 5, Mean: 3, Sum: 15
```

3.5.6. Métodos Argmax y Argmin en TensorFlow

En *TensorFlow*, los métodos *argmax* y *argmin* son herramientas esenciales en el análisis de probabilidades, especialmente en tareas de clasificación. Estos métodos identifican la posición del valor máximo (*argmax*) y mínimo (*argmin*) en un tensor a lo largo de un eje especificado.

- ***argmax*** se utiliza comúnmente para determinar la clase más probable en modelos de clasificación.

```
import tensorflow as tf

# Probabilidades de pertenencia a tres clases
probabilidades = tf.constant([0.1, 0.8, 0.1])

# Clase con la mayor probabilidad
clase_predicha = tf.argmax(probabilidades)

print(clase_predicha.numpy())
# Salida: 1
```

En este ejemplo, *argmax* identifica que la segunda clase tiene la mayor probabilidad.

- Aunque menos común en clasificación, ***argmin*** puede ser útil para identificar la clase menos probable.

```
# Usando el mismo tensor de probabilidades
clase_menos_probable = tf.argmin(probabilidades)

print(clase_menos_probable.numpy())
# Salida: 0 o 2
```

Aquí, *argmin* encuentra la clase o clases con la menor probabilidad, que en este caso son las clases 0 y 2.

3.5.7. El Método Squeeze en *TensorFlow*

El método *squeeze* en *TensorFlow* es útil para eliminar dimensiones de tamaño 1 de la forma de un tensor. Esto es particularmente importante cuando se trabaja con tensores que tienen dimensiones redundantes o innecesarias, lo cual puede ocurrir frecuentemente después de ciertas operaciones como reshape.

El siguiente ejemplo muestra cómo squeeze elimina las dimensiones de tamaño 1 de un tensor.

```
import tensorflow as tf

# Creando un tensor con una dimensión extra de tamaño 1
tensor = tf.constant([[[1], [2], [3]]])

# Aplicando squeeze para eliminar dimensiones de tamaño 1
tensor_squeezed = tf.squeeze(tensor)

print(tensor_squeezed.numpy())
# Salida: [1 2 3]
```

En este caso, *squeeze* transforma el tensor de forma (3,1) a un tensor de forma (3).

Especificando Ejes en *Squeeze*

Se pueden especificar ejes particulares para aplicar *squeeze*, en lugar de eliminar todas las dimensiones de tamaño 1.

```
# Creando un tensor con múltiples dimensiones de tamaño 1
tensor = tf.constant([[[1, 2, 3]]])

# Aplicando squeeze solo en el primer eje
tensor_squeezed = tf.squeeze(tensor, axis=[0])

print(tensor_squeezed.numpy())
# Salida: [[1 2 3]]
```

Aquí, *squeeze* se aplica solo al primer eje, cambiando la forma del tensor de (1,1,3) a (1,3), manteniendo una dimensión de tamaño 1.

3.5.8. El Método *One Hot* en *TensorFlow*

El método *one hot* en *TensorFlow* es fundamental en tareas de clasificación, especialmente en el procesamiento de etiquetas. Convierte enteros en vectores binarios donde la posición correspondiente al entero se marca con un '1', mientras que todas las demás posiciones se marcan con un '0'.

El siguiente ejemplo demuestra cómo usar *one hot* para convertir un conjunto de etiquetas numéricas en su representación *one hot*.

```
import tensorflow as tf

# Etiquetas en formato numérico
etiquetas = [0, 1, 2]

# Aplicando one hot encoding
etiquetas_one_hot = tf.one_hot(etiquetas, depth=3)

print(etiquetas_one_hot.numpy())

# Salida:
# [[1. 0. 0.]
# [0. 1. 0.]
# [0. 0. 1.]]
```

Este código transforma la lista de etiquetas numéricas en un tensor donde cada fila corresponde a la representación *one hot* de la etiqueta.

Personalizando One Hot Encoding

TensorFlow permite personalizar los valores de *on* y *off* en la representación *one hot*.

```
# Personalizando los valores de one y off
etiquetas_one_hot_custom = tf.one_hot(etiquetas, depth=3, on_value=5.0,
off_value=-5.0)

print(etiquetas_one_hot_custom.numpy())

# Salida:
# [[ 5. -5. -5.]
# [-5. 5. -5.]
# [-5. -5. 5.]]
```

En este ejemplo, los valores *on* y *off* se han personalizado a *5* y *-5*, respectivamente.

El método *one hot* es muy importante en la preparación de datos para modelos de clasificación, permitiendo una representación clara y eficiente de las etiquetas de clase.

3.5.9. Detección de GPU en *TensorFlow*

La detección y utilización de una GPU en *TensorFlow* es crucial para mejorar el rendimiento en el entrenamiento de modelos de aprendizaje automático. *TensorFlow* automáticamente asigna operaciones a la GPU si está disponible.

Verificación de la Disponibilidad de GPU

Para verificar si *TensorFlow* puede detectar la GPU, se utiliza el módulo *tensorflow.config*. Este código lista los dispositivos físicos del tipo 'GPU' y verifica si hay alguno disponible:

```
import tensorflow as tf

# Listando los dispositivos disponibles
dispositivos = tf.config.list_physical_devices('GPU')

# Verificando si hay GPUs disponibles

hay_gpu = len(dispositivos) > 0
print(f"¿GPU disponible?: {hay_gpu}")

# Salida: ¿GPU disponible?: True o False
```

Obteniendo Detalles de la GPU

Si se detecta una GPU, se pueden obtener más detalles sobre ella:

```
if hay_gpu:
    for gpu in dispositivos:
        print(tf.config.experimental.get_device_details(gpu))

# Salida: Detalles de la GPU como el nombre y otros atributos
```

4. *KERAS*. UNA LIBRERÍA DE ALTO NIVEL PARA RNAs

Keras es una interfaz de programación de aplicaciones (API) de alto nivel para redes neuronales, que opera sobre *TensorFlow* (y anteriormente, otras bibliotecas como *Theano* o *Microsoft Cognitive Toolkit*). Fue diseñada con el objetivo de permitir una experimentación rápida y eficiente. Aspectos destacados de Keras incluyen:

- **Facilidad de Uso**: *Keras* es conocida por su facilidad de uso, con una API clara y concisa que simplifica el proceso de construcción de modelos de aprendizaje automático.
- **Modularidad**: Los modelos en *Keras* se construyen ensamblando bloques de construcción (como capas, funciones de activación y optimizadores) de manera flexible y fácil.
- **Enfoque en la Practicidad**: *Keras* está diseñada para ser práctica y orientada a la resolución de problemas reales, sin sacrificar la capacidad de implementar modelos complejos y personalizados.

- **Integración con *TensorFlow***: Como parte del ecosistema *TensorFlow*, *Keras* se beneficia de su robustez, escalabilidad y capacidades de despliegue.

Figura 9: Logo de keras

La combinación de *TensorFlow* y *Keras* ofrece una plataforma poderosa y flexible para el desarrollo de aplicaciones de aprendizaje automático, desde prototipos rápidos hasta implementaciones a gran escala en producción.

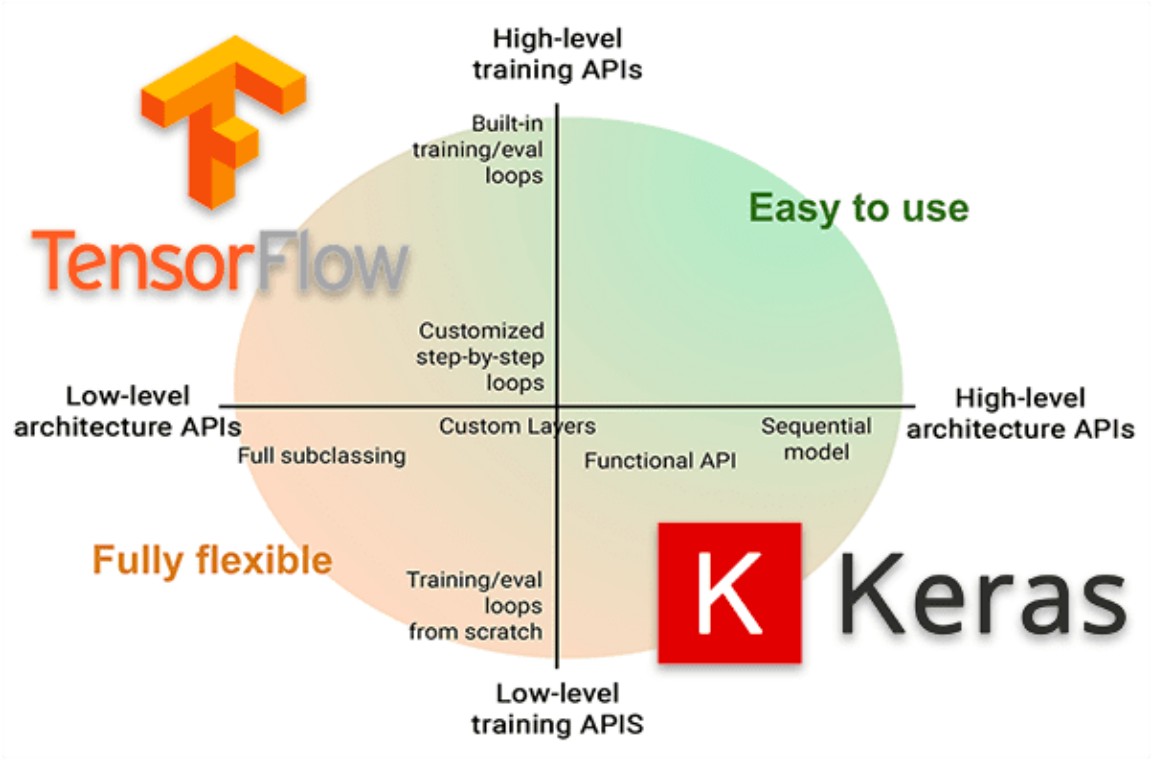

Figura 10: Relación entre Keras y Tensorflow

4.1. CONSTRUCCIÓN DE UN MODELO

En *Keras*, un modelo se construye a partir de capas. El tipo más común de modelo es la secuencia, que se crea utilizando la clase *Sequential*.

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
```

Las capas se añaden al modelo secuencialmente. Este ejemplo muestra la adición de capas densas con funciones de activación *ReLU* y *Softmax*.

```
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Después de definir el modelo, se debe compilar con un optimizador, una función de pérdida y métricas de rendimiento.

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

Entrenamiento del Modelo

El modelo se entrena con datos. Aquí se muestra cómo entrenar el modelo con un conjunto de datos de ejemplo durante 10 épocas.

```
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

Evaluación y Predicción

Finalmente, el modelo se evalúa con datos de prueba, y se pueden hacer predicciones sobre nuevos datos.

```
test_loss, test_acc = model.evaluate(x_test, y_test)
predictions = model.predict(x_new)
```

4.2. TENSORFLOW PLAYGROUND

TensorFlow Playground es una herramienta interactiva en línea desarrollada por el equipo de *TensorFlow* en Google. Esta herramienta visualiza el funcionamiento de las *Redes Neuronales Artificiales* (RNAs) y permite a los usuarios experimentar con ellas en tiempo real. Es especialmente útil para la enseñanza y el aprendizaje, ya que proporciona una forma intuitiva y gráfica de entender cómo las diferentes configuraciones y parámetros afectan el comportamiento y el rendimiento de una red neuronal.

Con TensorFlow Playground, los usuarios pueden:

- Observar cómo los algoritmos aprenden patrones en los datos.
- Experimentar con distintas arquitecturas de red, incluyendo la cantidad de capas ocultas y neuronas.
- Ajustar varios hiperparámetros como la tasa de aprendizaje y el tipo de activación.
- Visualizar el proceso de aprendizaje en tiempo real y entender cómo la red minimiza el error a lo largo de las iteraciones.

Esta herramienta es accesible directamente desde un navegador web y no requiere instalación, lo que la convierte en un recurso educativo accesible para aquellos que están comenzando a aprender sobre aprendizaje profundo y redes neuronales. Además, al ser una herramienta de código abierto, ofrece a los usuarios la oportunidad de explorar y modificar el código subyacente para una comprensión más profunda de su funcionamiento.

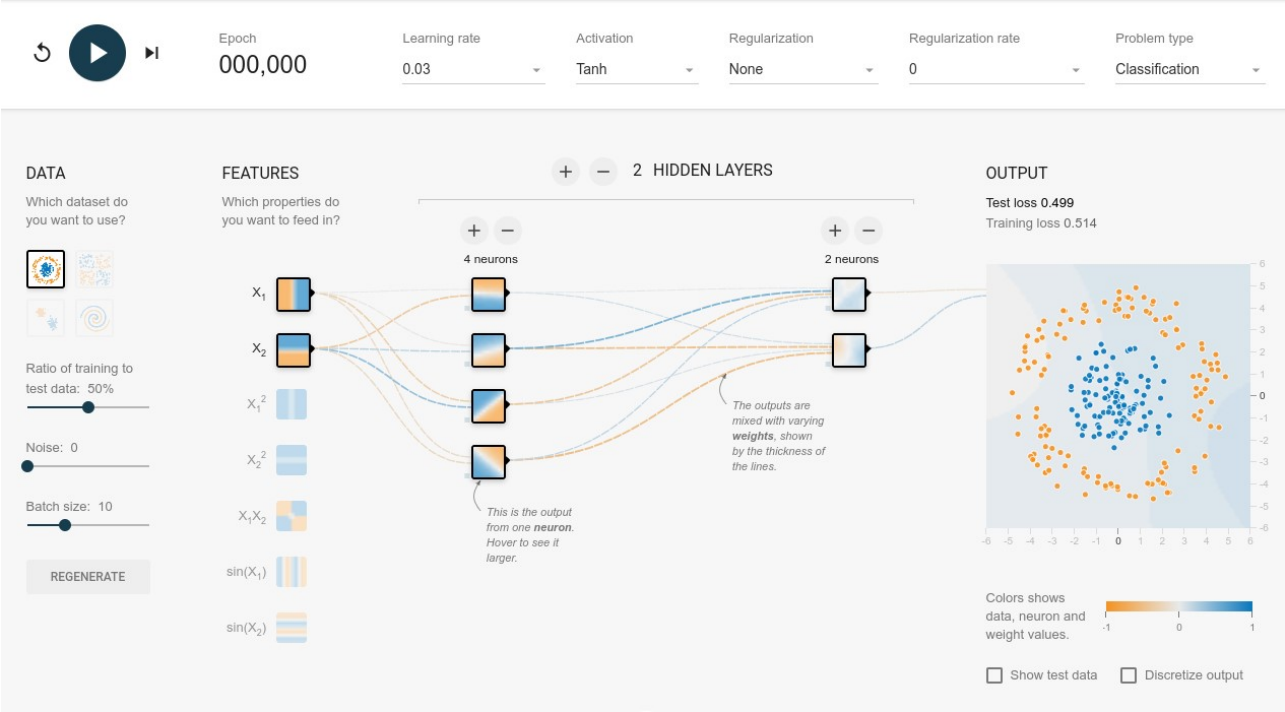

Figura 11: Tensorflow Playground