

vuela

andalucia.vuela.es

CONECTA CON ANDALUCÍA

**PLATAFORMA
DIGITAL DE
ANDALUCÍA**



DEEP LEARNING

ÍNDICE DE CONTENIDO

1.	Introducción a las Redes Neuronales Artificiales	3
1.1.	<i>Modelos de grafos para ANNs y Deep Learning</i>	5
1.2.	<i>Deep Learning y Machine Learning</i>	13
1.3.	<i>Herramientas de Deep Learning</i>	16
2.	Introducción a TensorFlow	23
3.	Redes <i>feedforward</i>	31
4.	Redes Neuronales Convolucionales	44
5.	Redes Neuronales Recurrentes	66
6.	Autoencoders	75
7.	Aprendizaje por refuerzo profundo	78
8.	Máquinas de Boltzmann	83
9.	Generative Adversarial Networks	88
10.	Alternativas a TensorFlow. Pytorch	91

1. Introducción a las Redes Neuronales Artificiales

Las redes neuronales son modelos matemáticos, inspirados en el funcionamiento del cerebro humano. Se puede considerar que los modelos de redes neuronales son máquinas numéricas para tareas cognitivas como aprendizaje u optimización. El origen de las redes neuronales artificiales se remonta a principios de los años 40, cuando los avances producidos en la biología permitieron a McCulloch y Pitts proponer una teoría general de procesamiento de información basada en redes de elementos de decisión o conmutadores binarios, a los que llamaron neuronas. No obstante, estos elementos eran mucho más simples que sus equivalentes biológicos.

En 1949, Donald Hebb propuso el primer algoritmo de aprendizaje para algunos modelos neuronales básicos existentes. Más tarde, Rosenblatt concibió la idea del perceptrón de una única capa (1958). En esta época, también fue notable el trabajo de Widrow y Hoff para formular reglas de aprendizaje, aplicables al modelo del perceptrón de Rosenblatt, mediante el algoritmo Adaline (1960). En 1969, Minsky y Papert realizaron un estudio de las posibilidades del perceptrón, señalando sus limitaciones.

A partir de entonces, hubo un periodo de tiempo de aproximadamente entre 15 y 20 años en el que las redes neuronales artificiales perdieron popularidad, y su desarrollo quedó estancado. No obstante, durante este periodo hubo algunas investigaciones de gran relevancia: Considerando esencialmente modelos de redes neuronales recurrentes, destacan los trabajos de Hopfield y Kohonen, con el desarrollo de la red de Hopfield y los mapas autoorganizativos, respectivamente.

A finales de la década de los 80 y principios de los 90, surgieron nuevos modelos de redes neuronales artificiales recurrentes, cuya principal característica es el procesamiento de patrones de datos de longitud variable. Destacan los trabajos de Jordan y posteriormente de Elman, dando lugar a los inicios de los modelos hoy conocidos como redes neuronales recurrentes.

Posteriormente, desde finales de la década de los 90 hasta la actualidad, han surgido nuevos modelos de redes neuronales recurrentes tanto feedforward como recurrentes, bien con el fin de emular alguna característica del cerebro humano, bien con el propósito de solucionar algún problema práctico. Como ejemplos, podemos citar los modelos LSTM, algunas arquitecturas de red basadas en modelos probabilísticos, las redes convolucionales, redes de impulsos, y las máquinas de Boltzmann. Adicionalmente, otras líneas de investigación han desembocado en el desarrollo de modelos híbridos que toman características de distintos tipos de redes, como por ejemplo las redes FIR, los perceptrones recurrentes o las redes de base radial (RBF) y sus versiones recurrentes recurrentes. Cabe destacar también la importancia que tienen hoy día distintas estrategias de entrenamiento, dando lugar a metodologías de diseño como *autoencoders* o *generative adversarial networks*.

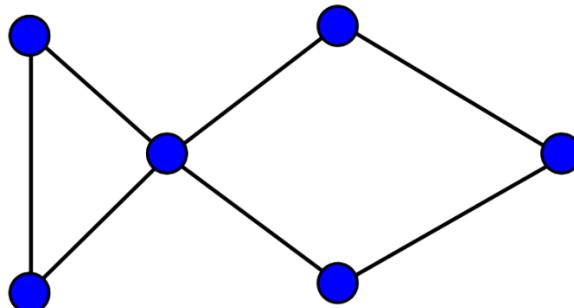
Aunque las redes neuronales surgieron como un modelo para tratar de imitar el funcionamiento del cerebro humano, hoy en día ningún modelo de red conocido ha sido capaz de duplicarlo. Sin embargo, la aplicación de diferentes modelos de redes a diversos problemas (clasificación, control, reconocimiento de patrones, etc.), con resultados muy prometedores, ha propiciado la expansión y la publicidad que esta herramienta tiene en la actualidad.

1.1. Modelos de grafos para ANNs y Deep Learning

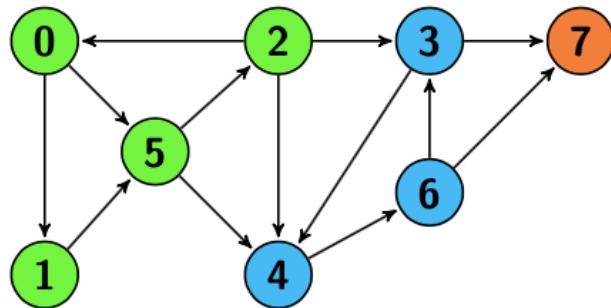
Las redes neuronales artificiales se pueden representar gráficamente como un modelo de grafo. Un grafo \mathbf{G} es un par $\mathbf{G}=(\mathbf{V}, \mathbf{A})$, donde \mathbf{V} es un conjunto de **nodos** o **vértices**, y \mathbf{A} un conjunto de **aristas**. Las aristas se definen como:

$$\mathbf{A} \subseteq \mathbf{V} \times \mathbf{V}$$

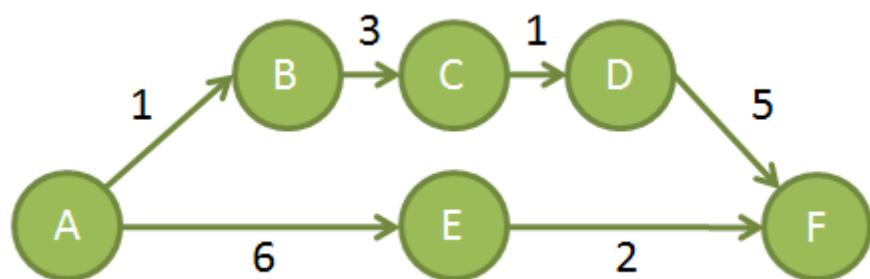
Es decir, una arista \mathbf{a} del conjunto \mathbf{A} es un par de vértices de \mathbf{V} , $(\mathbf{v}_i, \mathbf{v}_j)$. Su representación gráfica básica se suele realizar mediante círculo (vértices) y línea o flechas que los unen (aristas). Si para toda arista $(\mathbf{v}_i, \mathbf{v}_j)$ existe en el grafo también la arista $(\mathbf{v}_j, \mathbf{v}_i)$, entonces se dice que el grafo es **no dirigido**, y las aristas se suelen dibujar con líneas uniendo los vértices:



En caso contrario, el grafo es **dirigido** y la dirección (origen, destino) se representa con flechas:

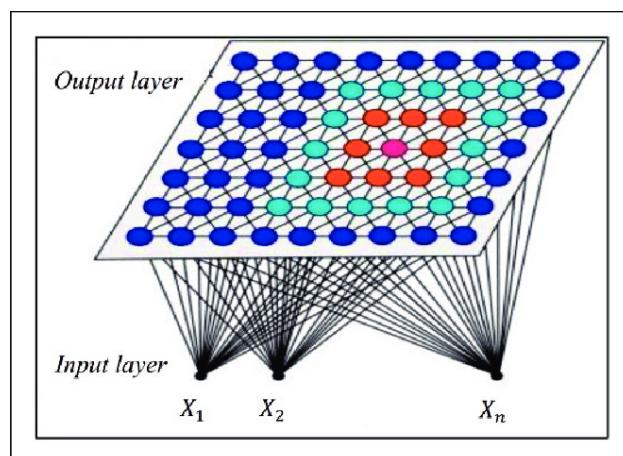


En un grafo, las aristas también pueden tener un valor numérico asociado (de nominado **peso w_i** para la arista a_i). Si este es el caso, el grafo se dice que es **ponderado**:



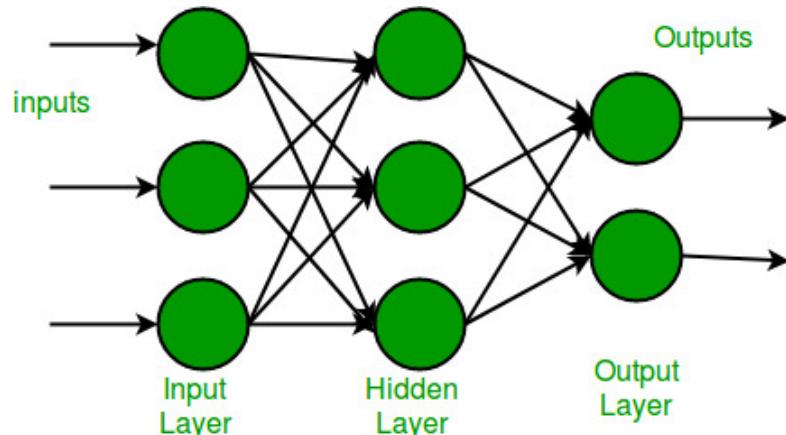
En redes neuronales artificiales, la representación de grafo asocia los nodos a las neuronas artificiales, mientras que las aristas se asocian a conexiones entre neuronas. Esta terminología suele completarse con los siguientes conceptos:

- | **Conexión simétrica.** Se da cuando, si existe una conexión del nodo i al nodo j , entonces también existe una conexión del nodo j al nodo i , y los pesos asociados $w_{ij} = w_{ji}$. Equivale al concepto de grafo **no dirigido**. Ejemplo de redes neuronales con este tipo de conexiones son los mapas autoorganizativos:

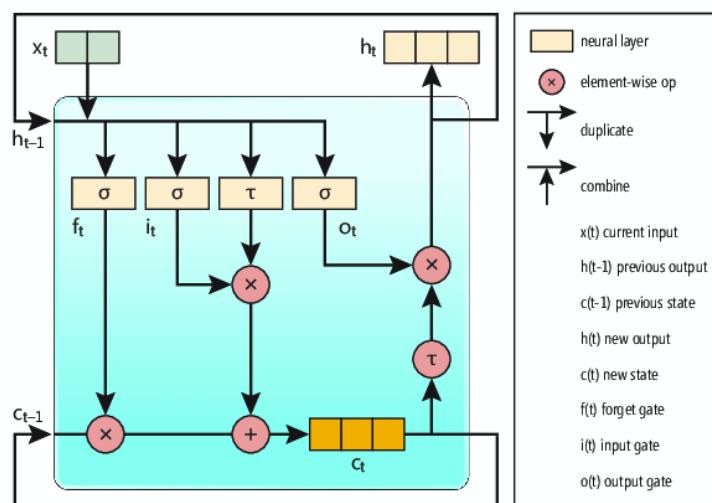


| **Conexión asimétrica.** Se da cuando no se cumplen las condiciones anteriores.

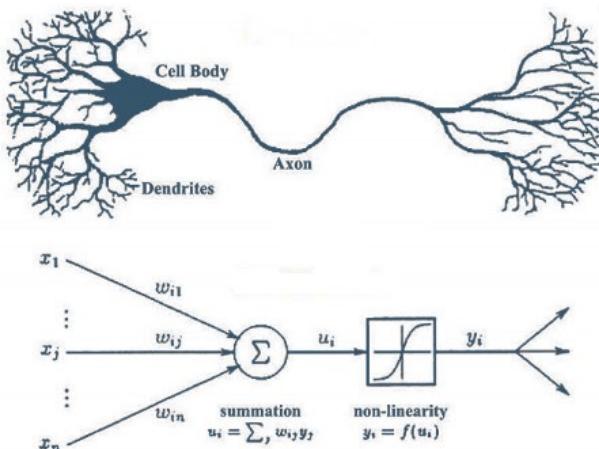
| **Conexión feedforward.** Todas las conexiones van en una dirección, desde un conjunto de neuronas de entrada hacia las neuronas de salida y, por tanto, no existen ciclos en el grafo. El ejemplo más corriente es el **Perceptrón Multicapa**:



| **Conexión recurrente.** Produce un cierto grado de retroalimentación, formando ciclos en la estructura topológica de la red. Ejemplo de este tipo de conexiones son comunes en las redes neuronales recurrentes, como por ejemplo los modelos Long-Short Term Memory (LSTM):



Esta representación de redes neuronales artificiales como grafos no es propia ni única de estos modelos, sino que también se utilizaban en los estudios de conexiones neuronales biológicas (y se siguen utilizando). De hecho, los modelos más tradicionales de redes neuronales artificiales se basan en el equivalente biológico:



Una **neurona biológica** recibe estímulos medidos como corriente eléctrica, que pueden ser externos, o procedentes de otra conexión neuronal previa. Esta capacidad de *percepción* se realiza a través de las **dendritas**. Seguidamente, el **interior** de la neurona o **cuerpo** procesa los estímulos percibidos y realiza una agregación ponderada de todos ellos, transmitiendo esta agregación a través del **axón** de la neurona. El axón produce un estímulo de salida (señal eléctrica), o no, dependiendo de si la agregación de los estímulos supera un cierto umbral de percepción, transmitiendo este estímulo eléctrico de salida a las siguientes neuronas. A esto se le conoce como la **activación** de la neurona.

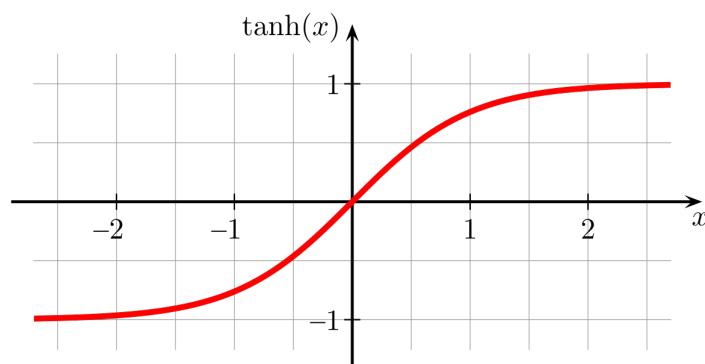
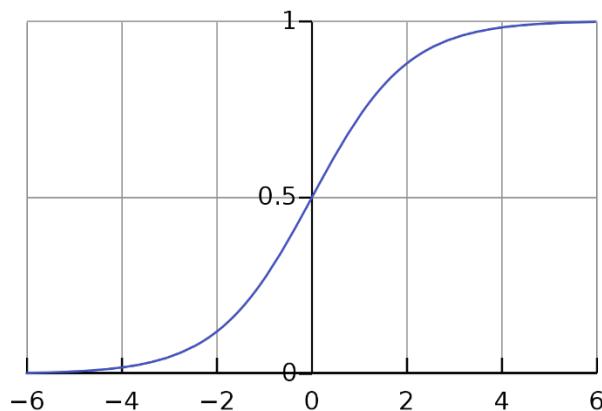
De forma análoga, una **neurona artificial** recibe un conjunto de estímulos (entradas x_1, x_2, \dots, x_n), cuya importancia se pondera multiplicando sus valores por pesos ($x_i * w_i$). El conjunto de todos estos estímulos ponderados es agregado en el interior de la neurona artificial mediante una suma de todos ellos.

Posteriormente, a estos estímulos agregados se le aplica una función de activación f , que dará como resultado la respuesta de salida de la neurona:

$$y = f(\sum_{i=1}^n w_i x_i)$$

Existen diversas propuestas de funciones de activación, aunque entre las más tradicionales encontramos la función sigmoide (que devuelve estímulo en el rango $[0, 1]$), o la función tangente hiperbólica (que devuelve estímulo en el rango $[-1, 1]$):

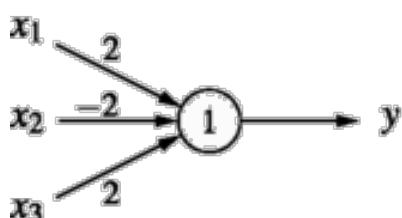
$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Una de las primeras propuestas de neurona artificial, conocida como **Unidad Lógica con Umbral (ULU)**, utilizaba también una función escalón como función de activación, simulando lo que los experimentos biológicos de los años 50 del s. XX habían dado a conocer: Si los estímulos ponderados de la neurona ($w_i \cdot x_i$) superan un cierto umbral h , entonces esta se activa y transmite una corriente eléctrica constante:

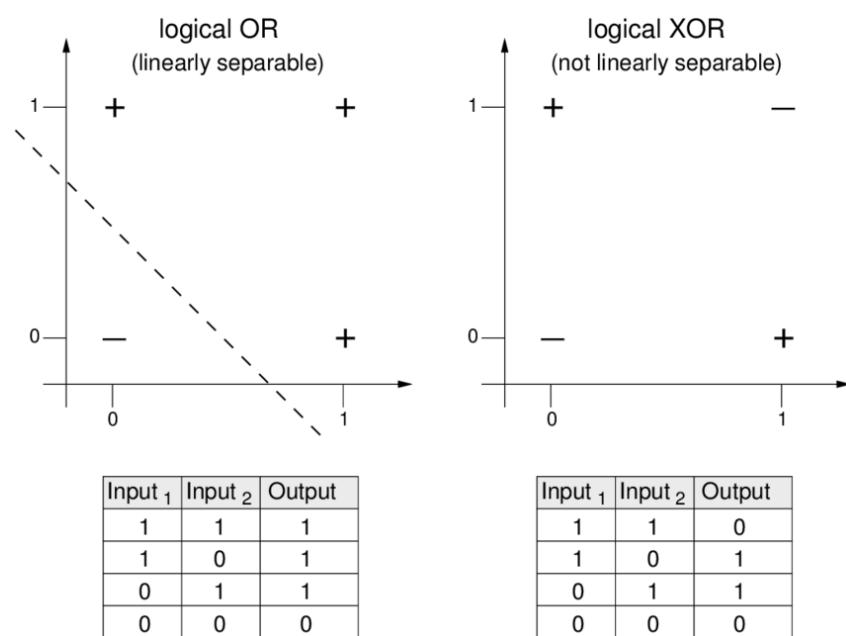
$$f(x) = \begin{cases} 1 & \sum_i w_i x_i \geq h \\ 0 & \sum_i w_i x_i < h \end{cases}$$

Su denominación se debe a que, originalmente, estos modelos se utilizaron para representar funciones lógicas:

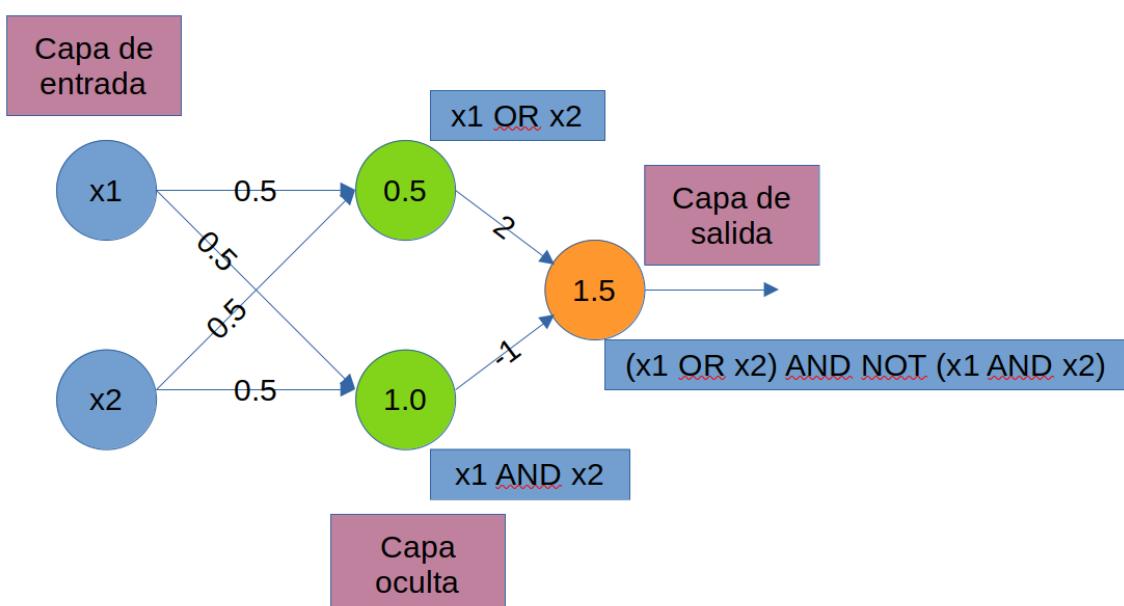


x_1	x_2	x_3	$\sum_i w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	-2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

No obstante, rápidamente se descubrió que el potencial de una neurona de tipo ULU estaba restringido a que las funciones a modelar fuesen linealmente separables, de modo que habría funciones lógicas (como el OR exclusivo, XOR), que no podrían modelarse con este tipo de neurona:



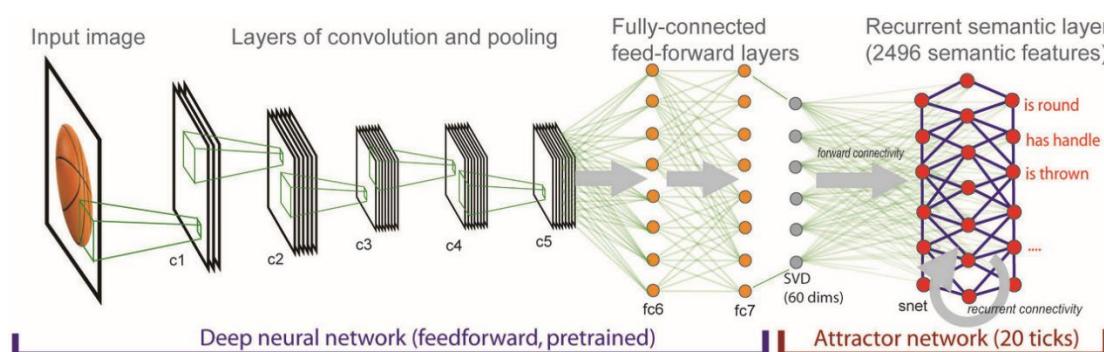
La solución a este problema surgió al tratar de encadenar diferentes ULUs, organizándolas en **capas de neuronas**, de modo que los cómputos realizados en las capas intermedias **se encargasen de realizar una transformación del espacio no linealmente separable a otro que sí lo fuera**:



Las redes neuronales artificiales contemporáneas también **organizan su estructura en capas**. Cada capa tiene un conjunto de neuronas que reciben como entrada los estímulos de las neuronas de la capa anterior, y dan su salida como entrada a las neuronas de la capa siguiente. Se distinguen 3 tipos de capas:

- | **Capa de entrada:** Recibe los datos de entrada del problema (x).
- | **Capas intermedias u ocultas:** Se encargan de hacer el procesamiento de los datos (cambios de espacio, proyecciones, inyecciones, etc.).
- | **Capa de salida:** Agrega los datos de la última capa oculta y devuelve el valor esperado **y** para las entradas a la red x .

Actualmente, los modelos de Deep Learning pueden contener cientos de capas con miles de neuronas, millones de parámetros a entrenar en total. Por este motivo, la **representación de grafo entre neuronas** no es una opción viable. En su lugar, en **Deep Learning se representa cada capa como un nodo del grafo**, y **las conexiones entre capas como aristas**:



1.2. Deep Learning y Machine Learning

Uno de los principales atractivos que han fomentado el desarrollo y uso contemporáneo de las redes neuronales artificiales se debe al **Teorema de Aproximación Universal** el cual indica que una red neuronal, equipada con el suficiente número de neuronas, es capaz de aproximar cualquier función continua con un error tan pequeño como se desee. Gracias a este avance, y a los desarrollos en algoritmos de optimización que se pueden usar para el aprendizaje de una red neuronal artificial, han surgido a lo largo de los años modelos de redes neuronales artificiales que se alejan del concepto de simulación de la neurona biológica para centrarse más en la resolución de problemas de ingeniería. De este modo, podemos encontrar hoy día modelos de redes neuronales artificiales (profundas o no) aplicables a los tres tipos de aprendizaje existentes en Machine Learning.

Redes neuronales para aprendizaje supervisado

Son las que más se conocen popularmente. Abarcan muchos tipos de modelos como el Perceptrón multicapa, redes de base radial, redes neuronales recurrentes, redes de impulsos, etc. Se caracterizan por la existencia de un conjunto de datos etiquetado, con patrones de entrada/salida $\{(x, y)\}$. Dichos patrones se presentan a la red, cuyo aprendizaje es como sigue:

- | Paso hacia adelante (**forward pass**): Los patrones de entrada $\{x\}$ se aportan como entrada a la red, que proporcionan una aproximación de las salidas $\{y'\}$.

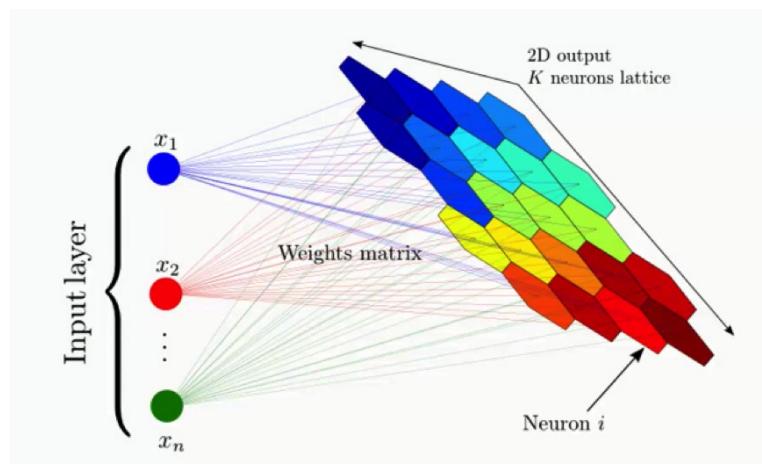
- | Paso hacia atrás (**backward pass**): Se calcula el error cometido por la red, tras emitir las salidas $\{y'\}$ para las correspondientes salidas esperadas $\{y\}$, mediante una función de pérdida. Estos errores se propagan hacia atrás (desde la salida hasta las entradas), calculando cuánto se tiene que modificar cada peso de la red para que el error sea menor del existente.
- | Actualización de pesos (**update**): Los pesos de la red se actualizan con respecto a los resultados del paso anterior y una velocidad de aprendizaje (**tasa de aprendizaje**).

En el aprendizaje supervisado, las redes neuronales artificiales se utilizan para resolver todos los problemas de esta sub-área, como clasificación (binaria/multiclas) o regresión.

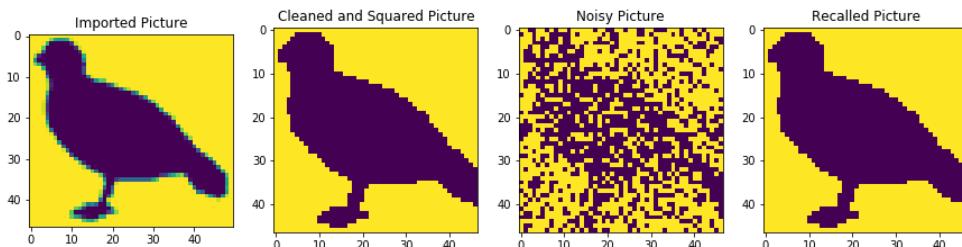
Redes neuronales para aprendizaje no supervisado

Son menos conocidas popularmente que las existentes para el aprendizaje supervisado, aunque no por ello menos importantes. Se utilizan para descubrir patrones dentro de un conjunto de datos no etiquetados, mediante la auto-organización de las neuronas de la red. Suelen ser modelos recurrentes en su mayoría, para responder a la necesidad de auto-adaptación.

En este contexto, las conexiones o estructura topológica de la red requieren de una definición de modelo conexionista (en malla, hexagonal, etc) que permita definir neuronas y relaciones con sus vecinas. El mecanismo de aprendizaje es el mismo que el guiado para el aprendizaje supervisado, aunque los algoritmos de entrenamiento difieren sustancialmente. Al no existir un patrón de salida con el que guiar el aprendizaje, estos modelos suelen utilizar medidas de información, tales como por ejemplo la minimización de la entropía.



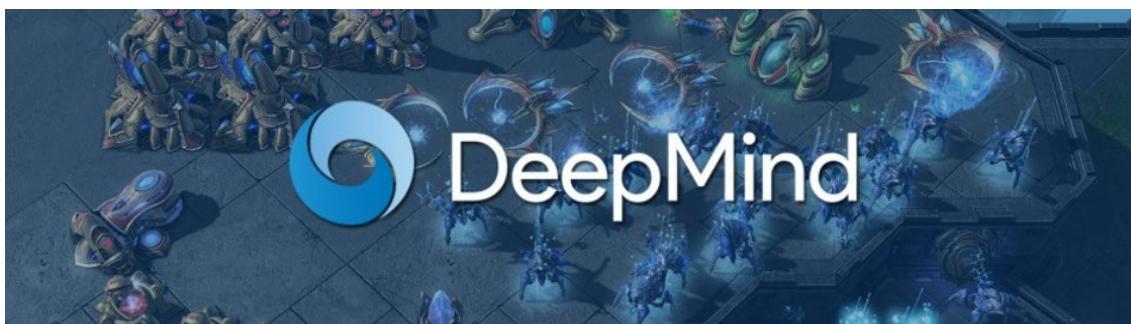
Al igual que ocurre con los algoritmos de clustering en aprendizaje no supervisado, los modelos de redes neuronales no supervisadas pueden también utilizarse para tareas de clasificación, aunque su rango de aplicaciones es bastante más amplio (reconstrucción de imágenes, limpieza de datos, etc.):



Redes neuronales para aprendizaje por refuerzo

Se utilizan para modelar la política del agente $\pi(a|s)$; es decir, selección de acción en un estado percibido del entorno. Su objetivo es, dado un estado s como entrada, proporcionar como salida la acción a a ejecutar.

Por este motivo, dado que existen pares entrada/salida, se suelen utilizar los mismos modelos de red que para aprendizaje no supervisado. No obstante, dada la naturaleza del aprendizaje por refuerzo (no existen datasets a priori), los algoritmos de aprendizaje son distintos e inspirados en la teoría clásica del aprendizaje por refuerzo. Ejemplo de este tipos de modelos son las **Deep Q-Networks**, redes **Actor-Critic**, **Policy** networks, etc. En particular, Deep Learning ha supuesto un gran avance en esta área del aprendizaje automático, proporcionando modelos de redes neuronales capaces no sólo de ganar a maestros de Go o de Ajedrez, sino también de aprender a interpretar videos e imágenes y jugar con gran maestría a algunos videojuegos (Atari, Starcraft, etc.).



1.3. Herramientas de Deep Learning

Deep Learning es una disciplina reciente, aunque de un crecimiento y adaptación elevados, por lo que existen varias herramientas en el mercado que se encuentran en constante desarrollo. Entre ellas destacamos:

Tensorflow

Es una librería Open Source escrita en C++ para computación numérica (aunque con APIs en varios lenguajes, principalmente Python), creada inicialmente en 2015 por el equipo de Google Brain. Aunque inicialmente fue pensada como biblioteca de Machine Learning, rápidamente fue adoptada como una de los principales repositorios y software fundamental para Deep Learning en particular. Se basa en el modelo de grafo de cómputo para la ejecución de operaciones, de modo que se facilite la aceleración de cálculos masivos a través de herramientas de paralelización (como GPUs o clusters). Igualmente, fue la primera herramienta en definir un tensor (array) como elemento fundamental de cómputo, facilitando de este modo el cálculo matricial y su escalabilidad.



Algunos de los valores añadidos de TensorFlow con respecto a su competencia directa son la facilidad para enviar modelos de Deep Learning para despliegue en empresas, y también la posibilidad de hacerlo en dispositivos empotrados o de bajo consumo (Edge Computing, Tensor Processing Units). Además, es una herramienta versátil que puede ejecutarse de manera local, en máquinas con una o múltiples tarjetas GPU, o hasta entornos de larga escala con múltiples máquinas de múltiples GPUs.

Como herramientas adicionales, TensorFlow ofrece varios paquetes:

- | **TensorBoard**, una herramienta visual que permite depurar y optimizar los modelos ejecutados.
- | **TensorFlow Serving**: herramienta diseñada para facilitar el despliegue de nuevos algoritmos y experimentos en ambientes de producción.

Keras

Keras es una biblioteca actualmente integrada y desplegada junto con TensorFlow. Está especializada en Deep Learning (al contrario del carácter genérico de TensorFlow), cuya principal meta es el prototipado rápido de modelos de Deep learning. Las principales características de Keras son:

- | Permite el rápido prototipado de los modelos de Deep Learning.
- | Soporta multitud de tipos de redes (Feed Forward, redes Convolucionales, redes Recurrentes, etc.).
- | Los principios de diseño que sigue son: alta modularidad, alta escalabilidad, simplicidad de uso y extensibilidad.

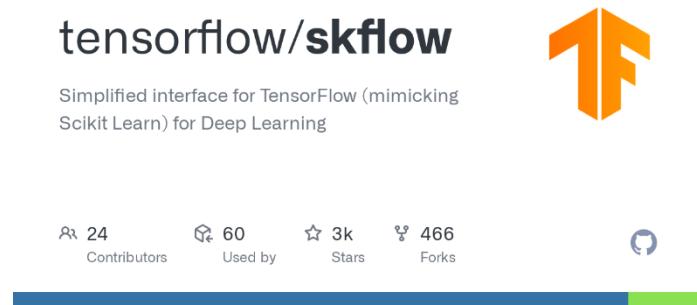


Para ello, Keras cuenta con múltiples paquetes que permiten extender la funcionalidad de las redes existentes, entre los que se destacan:

- | Capas regulares ó densas, de tipo Perceptrón multicapa (MLP)
- | Capas recurrentes, LSTM, GRU.
- | Capas convolucionales 1D, 2D y 3D
- | Autoencoders

Scikit-Flow

El paquete Scikit-Flow también se conoce como *TF Learn* ó *skFlow*. Es una biblioteca de Deep Learning basada en TensorFlow, aunque altamente compatible con los elementos de la biblioteca **scikit-Learn**. Resulta sencillo, por tanto, realizar tareas de Deep Learning a los usuarios familiarizados con el proceso de ML de SkLearn. Entre las capas preconstruídas que poseen, resaltan las capas densas, recurrentes y convolucionales, y también tiene una alta integración con el sistema de depuración **TensorBoard**.



Theano

Theano es otra biblioteca Open Source escrita en Python, que ofrece un marco de trabajo global para otras bibliotecas de Deep Learning, creada en la Universidad de Montreal (LISA Lab) en un grupo liderado por Yoshua Bengio (uno de los considerados como padres del Deep Learning). Theano permite definir, optimizar y evaluar expresiones matemáticas que involucran arrays multidimensionales o tensores de manera eficiente. Combina aspectos del Sistema Algebraico Computacional SAC (Computer Algebra System) con aspectos de optimización de compilación para el tratamiento de grandes cantidades de datos y para la ejecución de operaciones matemáticas complejas.

Al igual que en las bibliotecas anteriormente mencionadas, los tipos de datos en Theano están basados en tensores, o arrays multidimensionales. Theano permite también la ejecución transparente de las operaciones en CPU y en GPU, previa compilación de los modelos en lenguaje C/C++.

Theano proporciona toda la base necesaria para realizar cálculos matemáticos propios de Machine Learning de manera eficiente, aunque no dispone de una batería de elementos de Deep Learning tan extensa como las bibliotecas anteriores. Por esta razón, existen otras bibliotecas que lo usan como base para construir sistemas de más alto nivel. Por ejemplo, Keras es también compatible con Theano (en sus primeras versiones, de hecho, era la opción de instalación primaria frente a TensorFlow).

Lasagne

Lasagne es una biblioteca de Python que permite la construcción y el entrenamiento de Redes Neuronales usando Theano como base. Soporta *Feed Forward Networks*, Redes Neuronales Convolucionales, Redes Neuronales Recurrentes y combinaciones entre los tipos de redes mencionados.

Lasagne

Debido al principio de transparencia que tiene, Lasagne admite el uso de instrucciones propias y nativas de Theano, por lo que permite la incorporación de mayores restricciones y comportamientos particulares, en caso de que sea requerido.

NengoDL

Nengo es una biblioteca Python que facilita la construcción de modelos de redes neuronales. Su principio está orientado a la simulación de sistemas neuronales, más que al despliegue de redes neuronales como sistemas de cálculo en ingeniería. Por este motivo, incorpora distintos modelos de neuronas inspiradas en el comportamiento de diferentes sistemas nerviosos biológicos, al igual que en el cerebro humano. Por tanto, además de los modelos clásicos de redes neuronales artificiales utilizados en Deep Learning (perceptrones, capas convolucionales, etc.), incorpora otros como redes de impulsos o sistemas de percepción.



Pytorch

Pytorch es una biblioteca para Deep Learning desarrollada por Facebook. Se puede decir que Pytorch es el competidor directo de TensorFlow de Google, dado que sus características de desarrollo de modelos así como de despliegue son similares. Pytorch tiene una API que, al contrario que en TensorFlow, ha permanecido con pocos cambios a lo largo de su historia, lo que facilita la migración de modelos de Deep Learning entre diferentes versiones, al contrario que con los sistemas de TensorFlow.



Para finalizar este apartado, cabe destacar que existen multitud de frameworks más (aparte de los comentados) para Deep Learning y que, aunque clásicos como TensorFlow y Pytorch siguen siendo los más populares, algunas empresas también hacen uso de otros como Caffe, MxNet, DeepLearning4J, etc.

2. Introducción a TensorFlow

Como se ha comentado previamente, TensorFlow es una de las herramientas más populares para la elaboración de modelos de Deep Learning, motivado principalmente por el respaldo de una gran empresa como Google, su gran cantidad de desarrollos y velocidad en la evolución y adaptación del sistema, o su simplicidad de uso para tareas simples mediante herramientas como **Keras** o **TensorBoard**. Inicialmente fue una biblioteca pensada para realizar tareas de Machine Learning en general, aunque su evolución ha ido ligada a la expansión de Deep Learning y sus aplicaciones. La instalación de la biblioteca para Python se realiza de forma sencilla:

```
pip install tensorflow
```

En la URL <https://www.tensorflow.org/tutorials> podemos encontrar multitud de tutoriales para desarrollo rápido de modelos de Deep Learning, así como utilizar TensorFlow y Keras a nivel básico.

Para utilizar TensorFlow desde un script Python, lo usual es importarlo con alias. La sentencia más común es:

```
import tensorflow as tf
```

TensorFlow

- Instalación
- Aprende
- API
- Recursos
- Comunidad
- Más

Buscar
 Español - A...
[GitHub](#)
[Acceder](#)

TensorFlow Core

Descripción general Instructivos Guía TF 1 ↗

Filtrar

Instructivos de TensorFlow

Guía de inicio rápido para principiantes
Guía de inicio rápido para expertos

PRINCIPIANTE

Conceptos básicos de AA con Keras

- Clasificación de imágenes básica
- Clasificación de texto básica
- Clasificación de texto con TF Hub
- Regresión
- Sobreajuste y subajuste
- Guarda y carga modelos
- Ajusta los hiperparámetros con Keras Tuner
- Más ejemplos en keras.io ↗

Carga y procesamiento previo de datos

AVANZADO

Personalización

Entrenamiento distribuido

Imágenes

Texto

Audio

Datos estructurados

Los instructivos de TensorFlow se escriben como notebooks de Jupyter y se ejecutan directamente en Google Colab, un entorno para notebooks alojado que no requiere configuración. Haz clic en el botón *Ejecutar en Google Colab*.

Para principiantes

El mejor punto de partida es la API secuencial de Keras, que es fácil de usar. Conecta componentes básicos para compilar modelos. Después de finalizar estos instructivos, lee la [guía de Keras](#).

Guía de inicio rápido para principiantes

El notebook "¡Hello, World!" muestra la API secuencial de Keras y `model.fit`.

Conceptos básicos de Keras

Esta colección de notebook muestra tareas básicas de aprendizaje automático con Keras.

Carga datos

Estos instructivos usan `tf.data` para cargar varios formatos de datos y compilar canalizaciones de entrada.

Para expertos

Las API funcionales y de subclases de Keras proporcionan una interfaz definida por ejecución para la personalización y la investigación avanzada. Compila tu modelo y, luego, escribe la propagación hacia adelante y hacia atrás. Crea capas personalizadas, activaciones y bucles de entrenamiento.

Guía de inicio rápido avanzada

El notebook "¡Hello, World!" usa la API de subclases de Keras y un bucle de entrenamiento personalizado.

Personalización

Esta colección de notebooks muestra cómo compilar capas personalizadas y bucles de entrenamiento en TensorFlow.

Entrenamiento distribuido

Distribuye el entrenamiento de tu modelo en varias GPU, varias máquinas o TPU.

A nivel básico, los usuarios de NumPy pueden encontrar ciertas similitudes a la hora de utilizar TensorFlow, dado que muchas de las operaciones guardan una estrecha correlación con las de la mencionada biblioteca. Por ejemplo, mientras que en NumPy la estructura de datos central es el **ndarray**, en TensorFlow es el **Tensor**, aunque el tratamiento a nivel práctico es el mismo como un **array multidimensional**.

Al igual que en NumPy, en TensorFlow se puede crear un Tensor a partir de estructuras tipo lista/tupla de Python. No obstante, encontramos la primera diferencia al poder definir tensores por defecto como **inmutables**; es decir, con valores constantes:

```
a= tf.constant([1, 2, 3], shape=(3,), dtype=tf.float32)
```

Se puede también crear tensores variables (**mutables**):

```
b= tf.Variable([1, 2, 3], shape=(3,), dtype=tf.float32)
```

No obstante, un tensor no puede modificarse parcialmente, como en numpy (por ejemplo, haciendo `b[0]= 5`). En TensorFlow, se debe modificar el tensor al completo:

```
b.assign([3, 2, 1])
```

Existe una alta integración con NumPy, por lo que podemos transformar **ndarrays** a **Tensor** y viceversa:

```
cn= np.array([1, 2, 3])
```

```
ct= tf.convert_to_tensor(cn)
```

```
cn2= ct.numpy()
```

También existen casi todas las opciones de NumPy para generar arrays:

```
Z= tf.zeros(shape=(3, 2), dtype=tf.float32)
```

```
O= tf.ones(shape=(3), dtype=tf.float32)
```

```
E= tf.eye(num_rows=3)
```

El hecho de que los tensores en TensorFlow sean constantes quiere decir que su **contenido** no puede cambiar. No obstante, su forma sí puede adaptarse para facilitar el encadenamiento de operaciones sobre tensores:

```
E_o= tf.reshape(e, shape(1, 9))
```

```
E_o_sq= tf.squeeze(E_o)
```

Ejemplo: Creación de tensores

Ver ficheros Python asociados: [Codigo1.py](#)

Operaciones básicas con tensores

Las operaciones con tensores son fundamentalmente operaciones matemáticas cuyo funcionamiento se asemeja bastante a NumPy, como por ejemplo:

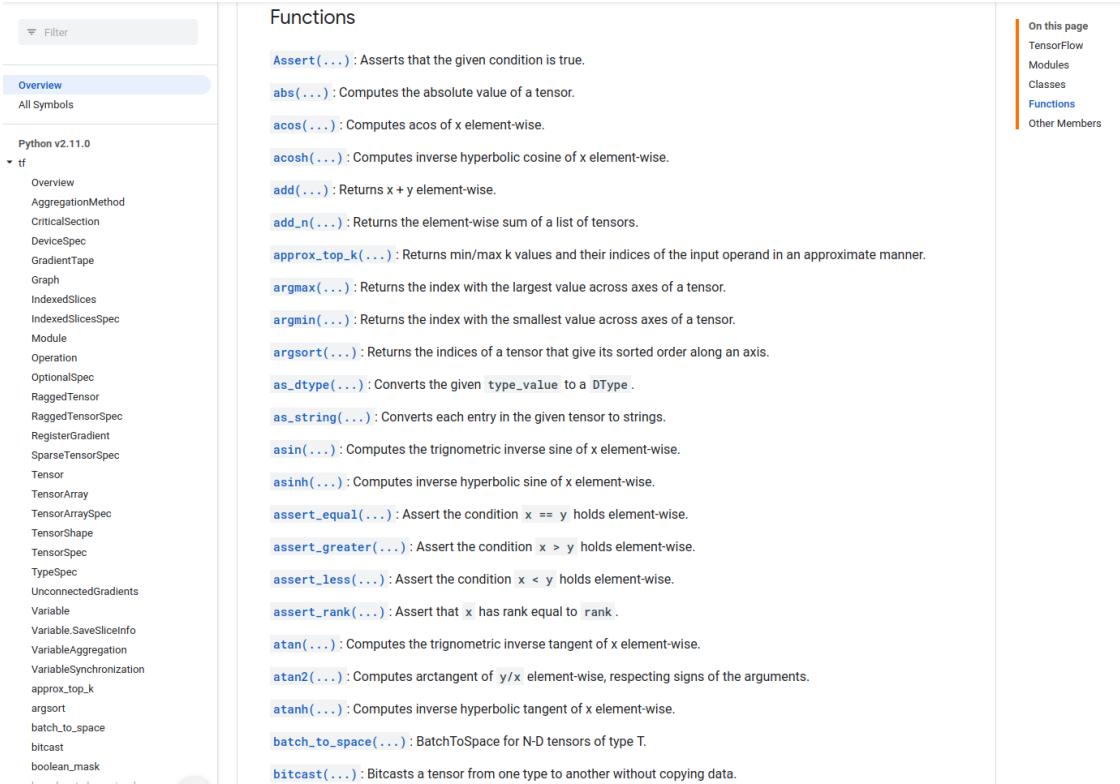
- | **Tf.add(x,y)** o **x+y**: Suma de tensores de misma dimensión.
- | **Tf.add(x,-y)** o **x-y**: Resta de tensores de misma dimensión.
- | **Tf.multiply(x,y)** o **x*y**: Multiplicación elemento a elemento de tensores.
- | **Tf.linalg.matmul(x,y)**: Multiplicación matricial de tensores.
- | **Tf.random.uniform(shape=(x,y))**: Generador de números aleatorios de distribución uniforme. Genera tensor de tamaño (x,y).
- | **Tf.random.normal(shape=(x,y))**: Generador de números aleatorios de distribución normal. Genera tensor de tamaño (x,y).

| **Tf.reduce_mean(x)**: Calcula la media de los elementos de un tensor x.

| **Tf.reduce_sum(x)**: Calcula la suma de los elementos de un tensor x.

| Etc.

El total de operaciones sobre tensores se puede visualizar en la API (https://www.tensorflow.org/api_docs/python/tf):



The screenshot shows the TensorFlow API documentation for Python v2.11.0. The left sidebar has a 'Functions' tab selected under the 'All Symbols' category. The main content area lists various functions with their descriptions. A vertical bar on the right contains links to 'On this page' sections: TensorFlow, Modules, Classes, Functions, and Other Members.

Function	Description
<code>Assert(...)</code>	Asserts that the given condition is true.
<code>abs(...)</code>	Computes the absolute value of a tensor.
<code>acos(...)</code>	Computes acos of x element-wise.
<code>acosh(...)</code>	Computes inverse hyperbolic cosine of x element-wise.
<code>add(...)</code>	Returns x + y element-wise.
<code>add_n(...)</code>	Returns the element-wise sum of a list of tensors.
<code>approx_top_k(...)</code>	Returns min/max k values and their indices of the input operand in an approximate manner.
<code>argmax(...)</code>	Returns the index with the largest value across axes of a tensor.
<code>argmin(...)</code>	Returns the index with the smallest value across axes of a tensor.
<code>argsort(...)</code>	Returns the indices of a tensor that give its sorted order along an axis.
<code>as_dtype(...)</code>	Converts the given <code>type_value</code> to a <code>DType</code> .
<code>as_string(...)</code>	Converts each entry in the given tensor to strings.
<code>asin(...)</code>	Computes the trigonometric inverse sine of x element-wise.
<code>asinh(...)</code>	Computes inverse hyperbolic sine of x element-wise.
<code>assert_equal(...)</code>	Assert the condition <code>x == y</code> holds element-wise.
<code>assert_greater(...)</code>	Assert the condition <code>x > y</code> holds element-wise.
<code>assert_less(...)</code>	Assert the condition <code>x < y</code> holds element-wise.
<code>assert_rank(...)</code>	Assert that <code>x</code> has rank equal to <code>rank</code> .
<code>atan(...)</code>	Computes the trigonometric inverse tangent of x element-wise.
<code>atan2(...)</code>	Computes arctangent of <code>y/x</code> element-wise, respecting signs of the arguments.
<code>atanh(...)</code>	Computes inverse hyperbolic tangent of x element-wise.
<code>batch_to_space(...)</code>	BatchToSpace for N-D tensors of type T.
<code>bitcast(...)</code>	Bitcasts a tensor from one type to another without copying data.

A continuación, se muestra un ejemplo de algunas operaciones:

Ejemplo: Operaciones con tensores

Ver ficheros Python asociados: **Codigo2.py**

Operaciones de indexación y concatenación

Tensorflow y Numpy también guardan una estrecha relación en cuanto a las facilidades de indexación y concatenación de ndarrays/tensores. El siguiente ejemplo muestra el uso de diferentes mecanismos de indexación:

- | Indexación con el operador dos puntos ":".
- | Indexación booleana.
- | División de tensor en partes uniformes con **Split**.
- | Concatenación con **concat**.
- | Apilación con **stack**.
- | Selección con **gather/gather_nd**.

Ejemplo: Indexación y concatenación de tensores

Ver ficheros Python asociados: **Codigo3.py**

Diferenciación automática

Una de las principales y más utilizadas capacidades de TensorFlow reside en la simplicidad para elaboración de algoritmos de optimización. En particular, la capacidad de elaborar automáticamente el cálculo de derivadas necesarias para la actualización del gradiente. Para ello, TensorFlow elabora una **cinta** donde se almacena el grafo de cómputo de un proceso implementado, a través del cual genera automáticamente los valores de la derivada en un punto.

Por ejemplo, sea la función $y=x^2$, y el punto **x=3.0**. La derivada en el punto **x** se calcula como **$y'=2x=6.0$** . Este cálculo se puede realizar de forma automatizada en TensorFlow **sin necesidad de conocer la expresión analítica de la derivada**:

```
X= tf.Variable(3.0)
```

```
With tf.GradientTape() as tape:
```

```
Y=X**2
```

```
# Cálculo de  $y'= dy/dx$ 
```

```
Derivada= tape.gradient(Y, X)
```

Este ejemplo se puede extender a operaciones mucho más complejas, lo que facilita la creación de diferentes modelos y su aprendizaje de forma completamente automatizada:

Ejemplo: GradientTape

Ver ficheros Python asociados: **Codigo4.py**

Conjuntos de datos

En TensorFlow es fácil gestionar grandes conjuntos de datos, representados como tensores, mediante la API de **Dataset**:

- | **Tf.data.Dataset.from_tensor_slices(x)**: Crea un conjunto de datos con los datos dados en el tensor **x**.
- | **Tf.data.Dataset.zip((x,y))** Crea un conjunto de datos de pares **(x,y)**

- | **Dataset.shuffle()**: Mezcla los patrones del dataset de forma aleatoria.
- | **Dataset.batch()**: Organiza el dataset en conjuntos disjuntos más pequeños (batches)

Además, Tensorflow facilita el acceso online a algunos datasets, mediante la instalación del paquete **tensorflow_datasets**:

pip install tensorflow_datasets

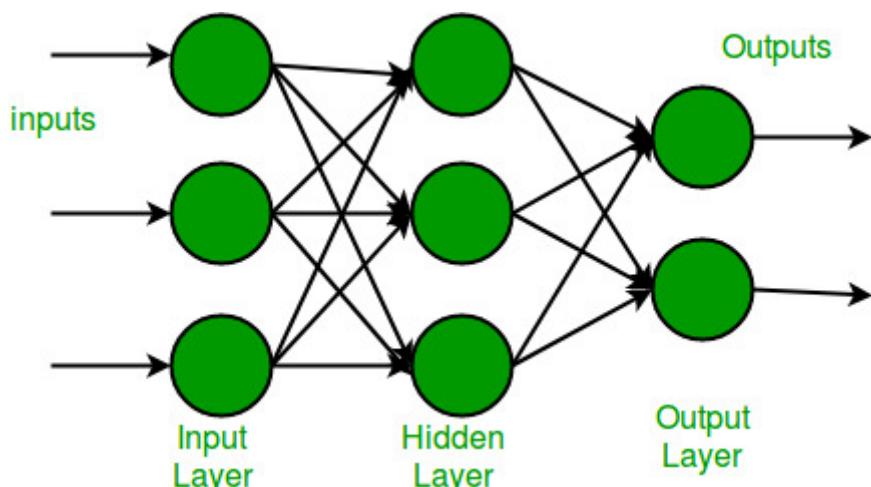
Estos datasets pueden cargarse directamente desde la API. El siguiente código fuente muestra un ejemplo de estas funcionalidades:

Ejemplo: Conjuntos de datos en TensorFlow

Ver ficheros Python asociados: [Codigo5.py](#)

3. Redes *feedforward*

Una conexión ***feedforward*** o ***hacia adelante*** es aquella en la que el flujo de datos sigue una misma dirección lineal, desde una entrada hacia una salida. Existen múltiples modelos de redes feedforward: Perceptrón Multicapa, redes de base radial, redes FIR, etc. No obstante, en la actualidad comúnmente nos referimos a redes de tipo Perceptrón cuando hablamos de redes ***feedforward***:



El perceptrón multicapa (MultiLayer Perceptron, MLP) es el modelo de red más conocido y utilizado en la historia de las redes neuronales artificiales. Consta de:

- | **Una capa de entrada**, donde se proporcionan los datos de entrada a la red.
- | **Ninguna, una o múltiples capas intermedias (ocultas)**, dedicadas a hacer procesamiento intermedio de los datos de entrada.
- | **Una capa de salida**, dedicada a agregar los datos de cómputo de la última capa intermedia y proporcionar las salidas requeridas.

Una característica del MLP es que todas las neuronas de una capa **L** reciben como entrada únicamente las salidas de la capa anterior **L-1**, y dan su propia salida como entrada a la capa siguiente **L+1**. Aunque el modelo teórico permite que cada neurona tenga una función de activación personalizada, lo usual es que todas las neuronas de una misma **capa l** compartan la misma función de activación f^l (por ejemplo: sigmoide, tangente hiperbólica, ReLU...). Llamaremos \mathbf{o}_j^l a la salida de la j -ésima neurona de la capa l , asumiendo que $\mathbf{o}_j^0 = \mathbf{x}_i$, es decir, la "capa cero" es la de entrada, y la neurona de entrada proporciona directamente el j -ésimo atributo del patrón de datos de entrada $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$. También notaremos como N^l al número de neuronas existentes en la **capa l**, a w_{ij}^l al peso que une la salida de la i -ésima neurona de la **capa l-1** con la j -ésima neurona de la **capa l**, y a b_j^l al **bias** de la j -ésima neurona de la **capa l**.

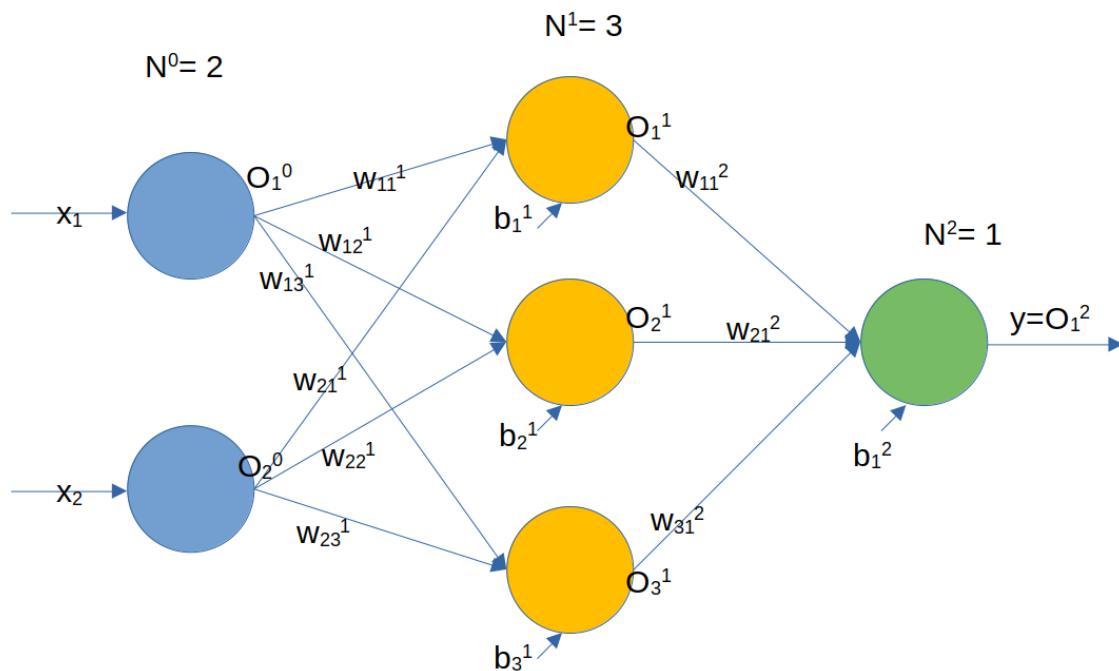
El modelo de funcionamiento del Perceptrón multicapa es el siguiente:

$$\begin{aligned} o_j^l &= f^l(z_j^l) \\ z_j^l &= \sum_{i=1}^{N^{l-1}} w_{ij}^l o_i^{l-1} + b_j^l \end{aligned}$$

Es habitual utilizar notación matricial para describir el funcionamiento de una red neuronal. En el caso anterior sería:

$$\mathbf{o}^l = f^l(\mathbf{W}^l \mathbf{o}^{l-1} + \mathbf{B}^l)$$

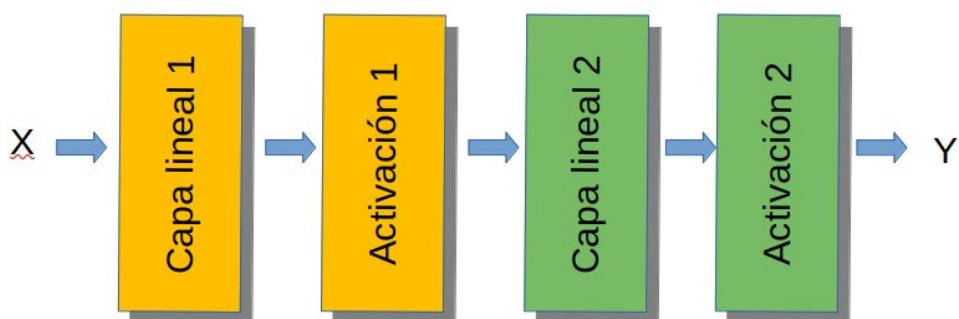
Donde $O^l = [o_1^l, o_2^l, \dots, o_{N^l}^l]^t$; $B^l = [b_1^l, b_2^l, \dots, b_{N^l}^l]^t$, y W^l es una matriz de tamaño $N^{l-1} \times N^l$ que contiene los pesos de las conexiones entre neuronas, entre la capa $l-1$ y la capa l .



La construcción de modelos en TensorFlow se realiza a través de la especialización de dos clases fundamentales:

- | **Tf.keras.Layer:** Representa un modelo de capa; es decir, el comportamiento parcial que una capa debería tener dentro de un modelo.
- | **Tf.keras.Model:** Representa un modelo de comportamiento. Los modelos pueden estar formados por capas o por otros modelos. Especialmente relevante es la funcionalidad **tf.keras.Sequential**, capaz de crear un modelo como secuencia de modelos o capas.

Vamos a diferenciar aquí entre operaciones de agregación (cómo se usan los pesos de cada capa) con la función de activación. Así, la creación de un modelo equivalente al de la figura anterior en TensorFlow sería:



El modelo se construiría como la concatenación de todas las cuatro capas anteriores. Modelaremos cada tipo de capa (lineal/activación) mediante **Layer** y concatenaremos con **Sequential**. La función de activación de ejemplo que implementaremos es la función logística/sigmoide:

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

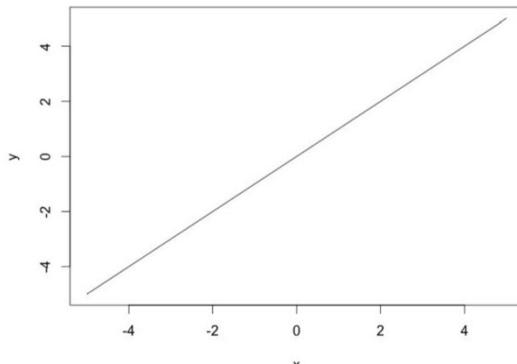
Ejemplo: Implementación manual de un MLP

Ver ficheros Python asociados: **Codigo6.py**

No obstante, se puede usar cualquier función de activación (siempre que sea derivable). Algunas de las más comunes son:

| **Lineal:** También conocida como función identidad:

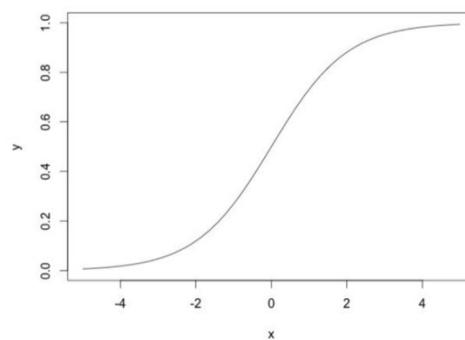
$$f(x) = x$$



| **Logís-**

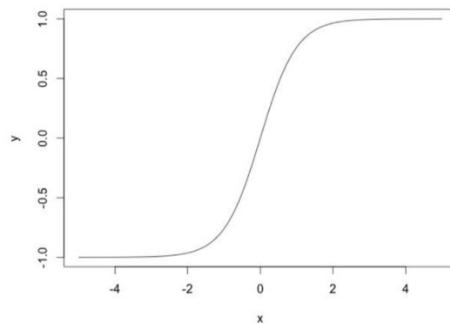
tica/Sigmoide: Devuelve valores en el intervalo [0,1]

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



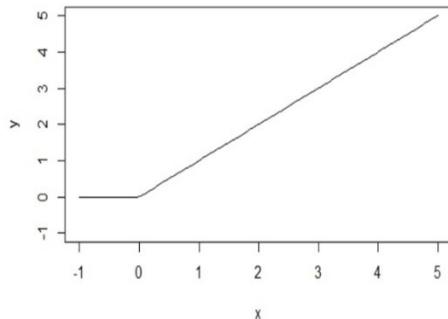
| **Tangente Hiperbólica (tanh):** Devuelve valores en el rango [-1,1].

$$f(x) = \tanh(x)$$



| **Rectified Linear Units (ReLU)**: Lineal a partir de 0, valor constante 0 para datos de entrada negativos:

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



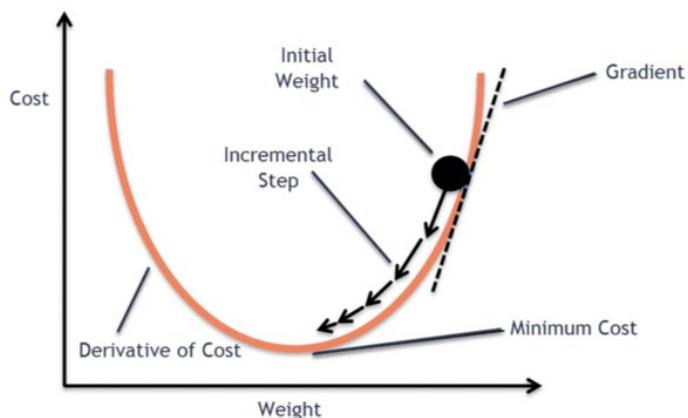
| **SoftMax**: Transforma los datos de salida de la red (que deben ser 2 ó más) en una distribución de probabilidad, mediante la fórmula:

$$\text{softmax}(x) = \left\{ \frac{e^{x_i}}{\sum_j e^{x_j}} \right\}$$

Aprendizaje: El algoritmo de retropropagación de errores

El algoritmo de retropropagación de errores (o **backpropagation**) es el método más conocido para entrenamiento de redes neuronales artificiales de tipo **feedforward**. Está basado en el **descenso del gradiente**, el cual actualiza iterativamente los valores de los parámetros del modelo **w** según la información de la primera derivada de una **función de error o función de coste**.

Notamos como la función $\mathbf{y}' = \mathbf{M}(\mathbf{x}, \mathbf{w})$ a la salida \mathbf{y}' del modelo \mathbf{M} , para las entradas \mathbf{x} y los parámetros \mathbf{w} del modelo. Llamamos también $J(\mathbf{y}, \mathbf{y}')$ (**también notada como $J(\mathbf{w})$**) a la función de error o coste entre la salida esperada del modelo \mathbf{y} , y la salida proporcionada por el modelo \mathbf{y}' . La idea perseguida por **BackPropagation**, al igual que en el descenso del gradiente, es encontrar el valor óptimo para \mathbf{w} que haga que la función de coste sea mínima:



Esto se lleva a cabo a través de una actualización iterativa de los pesos o parámetros del modelo, con una *fuerza* o *cantidad de aprendizaje* dada por un parámetro denominado **tasa de aprendizaje** λ :

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

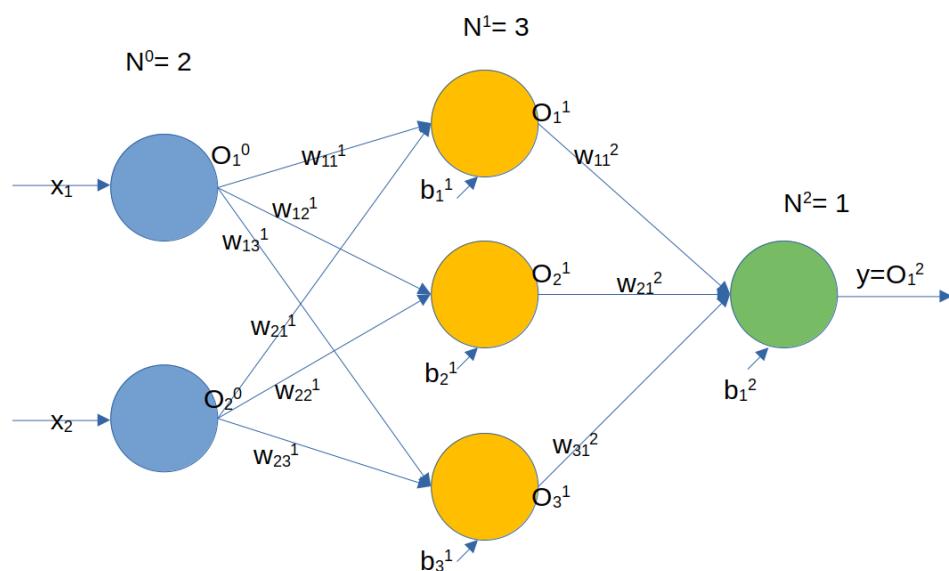
$$\Delta\mathbf{w} = -\lambda \nabla J(\mathbf{w})$$

El cálculo del gradiente de \mathbf{J} se realiza a través de la primera derivada de la función, con respecto a cada parámetro:

$$\nabla J(\mathbf{w}) = \frac{d J}{d w_j}$$

No obstante, en redes neuronales artificiales de al menos una capa oculta, el cálculo del gradiente para los pesos de las primeras capas sólo puede realizarse gracias a la **regla de la cadena**. Por ejemplo, para la siguiente figura, el cálculo del gradiente para el peso w_{11}^1 sería:

$$\frac{d J}{d w_{11}^1} = \frac{d J}{d O_1^2} \frac{d O_1^2}{d w_{11}^1} = \frac{d J}{d O_1^2} \frac{d O_1^2}{d z_1^2} \frac{d z_1^2}{d O_1^1} \frac{d O_1^1}{d z_1^1} \frac{d z_1^1}{d w_{11}^1}$$



Como podemos observar, el cálculo del gradiente de cada peso puede ser bastante complejo en el caso de que haya un número de capas elevado. Además, este hecho se amplifica si consideramos que hay múltiples funciones de pérdida posibles, y múltiples funciones de activación posibles para cada neurona de cada capa. Por este motivo, la implementación tradicional de redes neuronales artificiales resultaba compleja, hasta que se crearon frameworks como **TensorFlow** o **Pytorch** que incorporan el cálculo de gradiente automatizado, permitiendo abstraer estos cálculos y centrarnos en el diseño del modelo más que en el método de aprendizaje y su implementación particularizada para cada caso.

En TensorFlow, un paso de entrenamiento puede ser bastante sencillo una vez que se ha definido la función de pérdida, como se ilustra en el siguiente ejemplo:

Ejemplo: Implementación manual del entrenamiento en un MLP

Ver ficheros Python asociados: **Codigo7.py**

Funciones de activación en Tensorflow

Existen diferentes funciones de activación en Tensorflow. En particular, en el sub-paquete **tf.keras.activations** (ver enlace <https://keras.io/api/layers/activations>) para Deep Learning encontramos:

- | Relu
- | Sigmoid
- | Softmax
- | Softsign
- | Selu
- | Elu
- | Exponential
- | Tanh

K Keras

[Star](#)
[About Keras](#)
[Getting started](#)
[Developer guides](#)
[Keras API reference](#)
[Models API](#)
[Layers API](#)
[The base Layer class](#)
[Layer activations](#)
[Layer weight initializers](#)
[Layer weight regularizers](#)
[Layer weight constraints](#)
[Core layers](#)
[Convolution layers](#)
[Pooling layers](#)
[Recurrent layers](#)
[Preprocessing layers](#)
[Normalization layers](#)
[Regularization layers](#)
[Attention layers](#)
[Reshaping layers](#)
[Merging layers](#)
[Locally-connected layers](#)
[Activation layers](#)
[Callbacks API](#)
[Optimizers](#)
[Metrics](#)

Search Keras documentation...


[» Keras API reference / Keras layers API](#)

Keras layers API

Layers are the basic building blocks of neural networks in Keras. A layer consists of a tensor-in tensor-out computation function (the layer's `call` method) and some state, held in TensorFlow variables (the layer's `weights`).

A Layer instance is callable, much like a function:

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation='relu')
inputs = tf.random.uniform(shape=(10, 20))
outputs = layer(inputs)
```

Unlike a function, though, layers maintain a state, updated when the layer receives data during training, and stored in `layer.weights`:

```
>>> layer.weights
[tf.Variable 'dense/kernel:0' shape=(20, 32) dtype=float32>,
 <tf.Variable 'dense/bias:0' shape=(32,) dtype=float32>]
```

Keras layers API

- ◆ Creating custom layers

- ◆ Layers API overview

- The base Layer class

- Layer activations

- Layer weight initializers

- Layer weight regularizers

- Layer weight constraints

- Core layers

- Convolution layers

- Pooling layers

- Recurrent layers

- Preprocessing layers

- Normalization layers

- Regularization layers

- Attention layers

- Reshaping layers

- Merging layers

- Locally-connected layers

- Activation layers

Creating custom layers

While Keras offers a wide range of built-in layers, they don't cover every possible use case. Creating custom layers is very common, and very easy.

See the guide [Making new layers and models via subclassing](#) for an extensive overview, and refer to the documentation for [the base Layer class](#).

Layers API overview

The base Layer class

- [Layer class](#)
- [weights property](#)

Funciones de pérdida en Tensorflow

Al igual que las funciones de activación de neuronas, también existen algunas funciones de pérdida pre-construidas (subpaquete **tf.keras.losses**), cuya API podemos estudiar en la URL https://www.tensorflow.org/api_docs/python/tf/keras/losses:

- | BinaryCrossentropy (clasificación binaria)
- | CategoricalCrossentropy (clasificación multi-clase)
- | CosineSimilarity (clasificación multi-clase)
- | MeanAbsoluteError (clasificación + regresión)
- | MeanSquaredError (clasificación + regresión)
- | Etc.

TensorFlow

- Install
- Learn
- API
- Resources
- Community
- Why TensorFlow
- Search
-
- GitHub
- Accede!

Filter

Classes

```

class BinaryCrossentropy: Computes the cross-entropy loss between true labels and predicted labels.
class BinaryFocalCrossentropy: Computes the focal cross-entropy loss between true labels and predictions.
class CategoricalCrossentropy: Computes the crossentropy loss between the labels and predictions.
class CategoricalHinge: Computes the categorical hinge loss between y_true & y_pred.
class CosineSimilarity: Computes the cosine similarity between labels and predictions.
class Hinge: Computes the hinge loss between y_true & y_pred.
class Huber: Computes the Huber loss between y_true & y_pred.
class KLDivergence: Computes Kullback-Leibler divergence loss between y_true & y_pred.
class LogCosh: Computes the logarithm of the hyperbolic cosine of the prediction error.
class Loss: Loss base class.
class MeanAbsoluteError: Computes the mean of absolute difference between labels and predictions.
class MeanAbsolutePercentageError: Computes the mean absolute percentage error between y_true & y_pred.
class MeanSquaredError: Computes the mean of squares of errors between labels and predictions.
class MeanSquaredLogarithmicError: Computes the mean squared logarithmic error between y_true & y_pred.
class Poisson: Computes the Poisson loss between y_true & y_pred.
class Reduction: Types of loss reduction.
class SparseCategoricalCrossentropy: Computes the crossentropy loss between the labels and predictions.
class SquaredHinge: Computes the squared hinge loss between y_true & y_pred.

```

Functions

```

KLD(...): Computes Kullback-Leibler divergence loss between y_true & y_pred.
MAE(...): Computes the mean absolute error between labels and predictions.

```

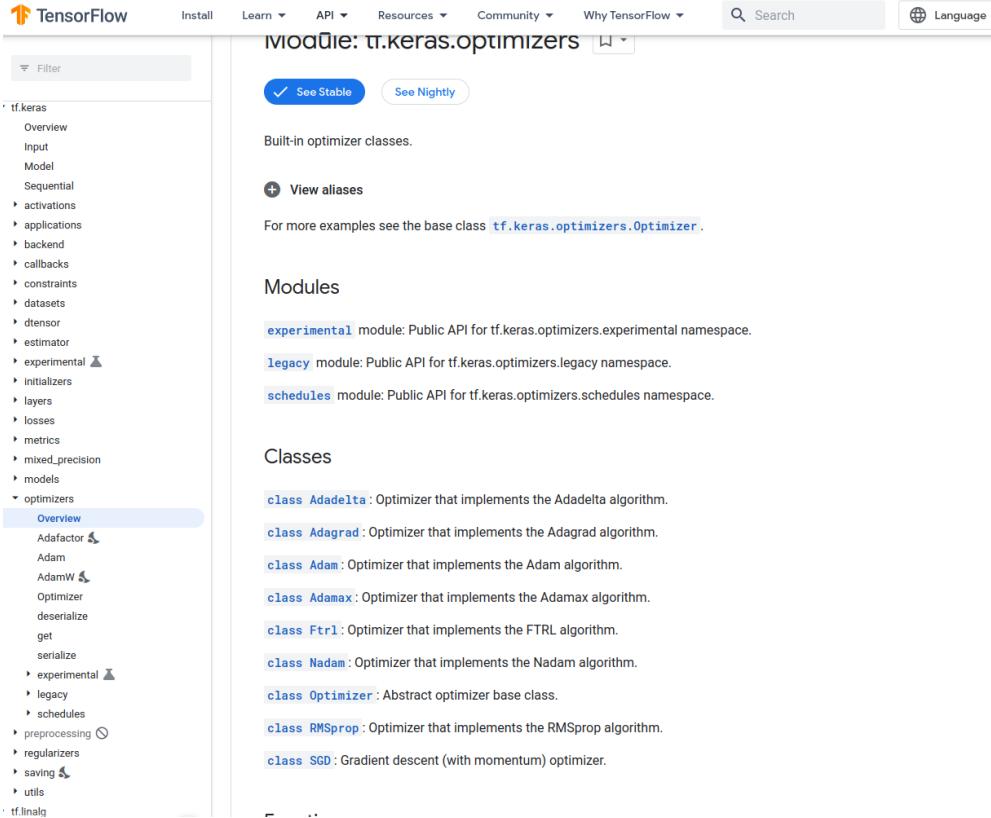
En esta página

- Classes
- Functions

Algoritmos de aprendizaje en TensorFlow

Para facilitar aún más la tarea de optimizar un modelo, Tensorflow incluye una batería de algoritmos de optimización que podemos usar en nuestros proyectos (URL de la API: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers). Algunos son:

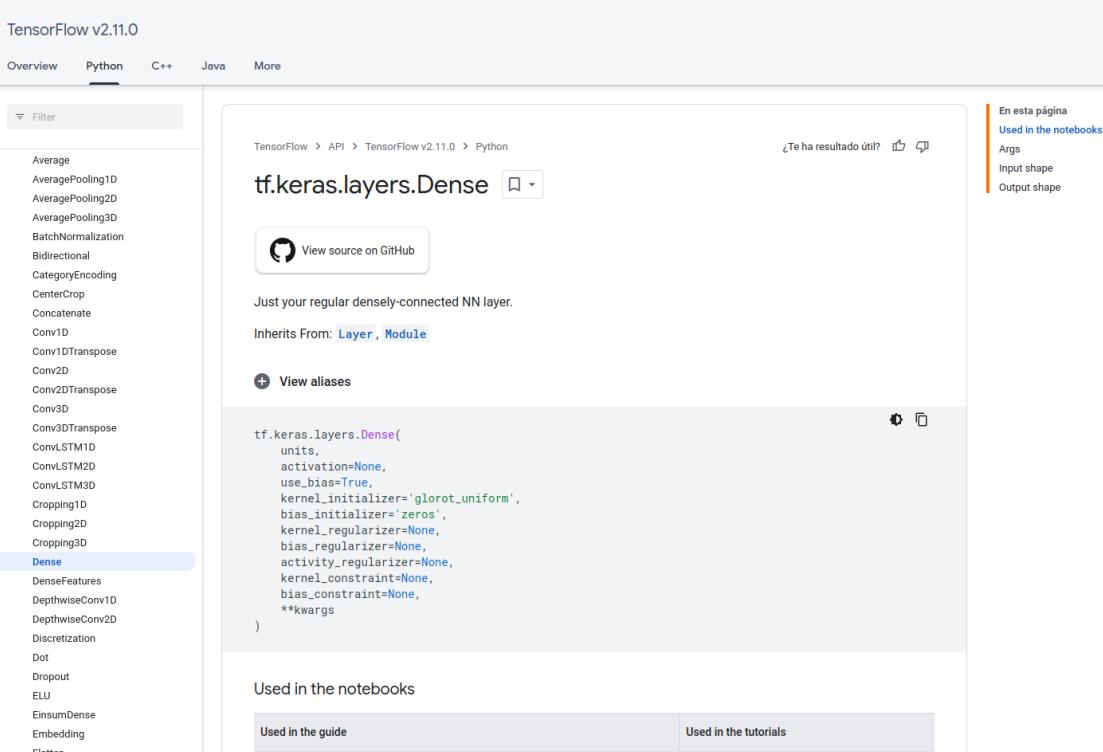
- | Regla Ada-Delta
- | Algoritmo Adam
- | Algoritmo Stochastic Gradient Descent (SGD)
- | Algoritmo Root Mean Square Propagation (RMSProp)
- | Etc.



The screenshot shows the TensorFlow API documentation for the `tf.keras.optimizers` module. The left sidebar has a tree view of the TensorFlow API, with `tf.keras` expanded and `optimizers` selected. The main content area displays the `tf.keras.optimizers` module documentation, which includes sections for `experimental`, `legacy`, and `schedules` modules, and lists various optimizer classes like `Adadelta`, `Adagrad`, `Adam`, `Adamax`, `Ftrl`, `Nadam`, `Optimizer`, `RMSprop`, and `SGD`. A sidebar on the right titled "En esta página" lists "Modules", "Classes", and "Functions".

Redes feedforward en Tensorflow

Aunque es posible definirse clases personalizadas para construir capas y modelos, tal y como hemos hecho en los apartados anteriores, lo habitual es hacer uso de la API de TensorFlow para construcción de modelos. En particular, las capas **lineales** se implementan como una capa **Keras** denominada **Dense**:



The screenshot shows the TensorFlow API documentation for the `tf.keras.layers.Dense` class. The main content area displays the class definition:

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Below the code, there are sections for "Used in the notebooks", "Used in the guide", and "Used in the tutorials". A sidebar on the left lists other layer classes like Average, AveragePooling1D, etc., with `Dense` being highlighted. Another sidebar on the right provides links to "Used in the notebooks", "Args", "Input shape", and "Output shape".

La clase **Model** también dispone de varias utilidades, tales como:

- | La función **predict**, para obtener las salidas del modelo a partir de unas entradas.
- | La función **compile**, para optimizar la ejecución de un modelo con un algoritmo de aprendizaje.
- | La función **fit**, para entrenar un modelo.
- | Etc.

El siguiente ejemplo ilustra cómo combinar todas estas características en un caso de uso para predecir el tipo de flor de Iris:

Ejemplo: Red feedforward con TensorFlow

Ver ficheros Python asociados: **Código8.py**

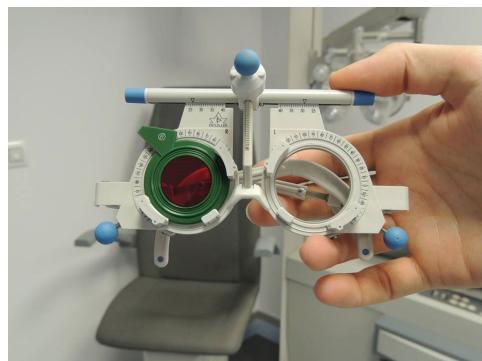
4. Redes Neuronales Convolucionales

El modelo teórico de red neuronal convolucional nació a finales de la década de los 90 del s. XX, aunque no fue hasta la primera década del s. XXI cuando estos modelos se afianzaron y encontraron aplicaciones prácticas reales que solucionar. Esto fue debido, en gran medida, al avance de algoritmos de optimización, el aumento de la potencia del hardware de cómputo, las estrategias de paralelización de algoritmos y la concepción del Deep Learning.

Las redes convolucionales tienen su origen práctico en el tratamiento de imágenes y, más concretamente, en el desarrollo o aprendizaje de filtros sobre imágenes para extracción de características de las mismas. No obstante, tienen su fundamento en el funcionamiento biológico de la percepción visual en la corteza visual primaria.

Filtros convolucionales

En ingeniería, un filtro es un elemento o función que ayuda a obtener características de los datos, dados como señales, a una determinada frecuencia. Por ejemplo, a nivel visual un filtro "rojo" elimina las frecuencias de luz percibidas salvo las que estén en el espectro del color rojo:



Un **filtro convolucional** es un filtro que aplica una función de **convolución** sobre la entrada. El filtro convolucional puede ser de 1D, 2D o múltiples dimensiones. En particular, dada una secuencia de valores medidos en el tiempo, $\mathbf{X}=(x_1, x_2, \dots, x_t)$, como por ejemplo la evolución de temperatura medida cada hora, y un filtro convolucional \mathbf{Y} , la operación de convolución se nota comúnmente como \otimes , como \odot y también con el símbolo asterisco “ $*$ ”. Por simplicidad, en este módulo usaremos el “ $*$ ” como operador. La convolución 1D de \mathbf{X} con el filtro \mathbf{Y} en el instante de tiempo t se calcula como:

$$S(t) = (X * Y)(t) = \int_{a=-\infty}^{\infty} X(a)Y(t-a)$$

En el caso discreto:

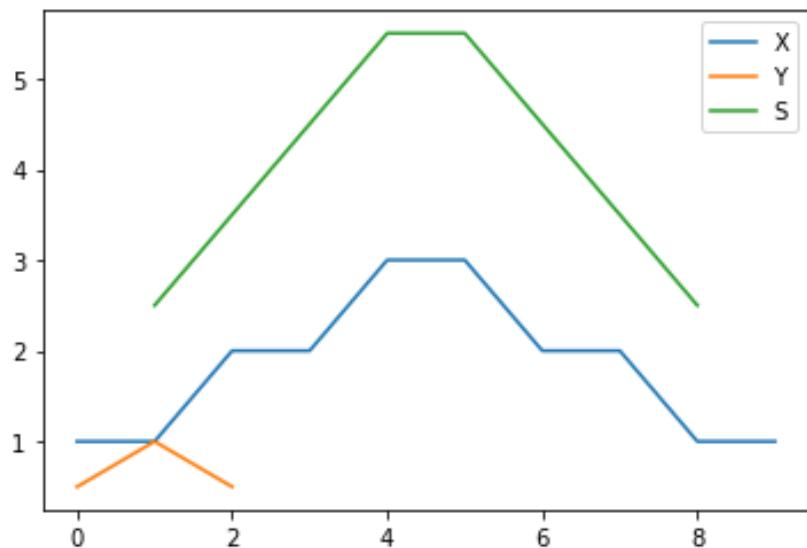
$$S(t) = (X * Y)(t) = \sum_a X(a)Y(t-a)$$

Es fácil implementar la función de convolución en Python, asumiendo que los valores de ambos arrays serían o fuera del rango de definición de los mismos:

```
def conv1D(X, Y):
    Conv_len = len(X) - len(Y) + 1

    Y_inv = Y[::-1].copy()
    result = np.zeros(Conv_len, dtype=np.float64)
    for i in range(Conv_len):
        result[i] = np.dot(X[i:i + len(Y)], Y_inv)
    return result
```

Veamos un ejemplo con los siguientes datos: $X=[1, 1, 2, 2, 3, 3, 2, 2, 1, 1]$, e $Y=[0.5, 1, 0.5]$:



En el caso de la convolución 2D, ampliamente utilizada en el tratamiento de imágenes, se asume que las señales X, Y son matrices 2D, y la convolución se define como:

$$S(m, n) = (X * Y)(m, n) = \sum_j \sum_i X(i, j)Y(m - i, n - j)$$

El cálculo de la convolución 2D se realiza para cada casilla (m,n) de la matriz de señal X . Es fácil de implementar en Python con el siguiente código:

```
# Convolución 2D
def conv2D(X, Y):

    Y_height, Y_width = Y.shape
    X_height, X_width = X.shape
    S_height = (X_height - Y_height)+1 #// stride + 1
    S_width = (X_width - Y_width)+1 #// stride + 1

    S= np.zeros((S_height, S_width), dtype=np.float64)

    for y in range(0, S_height):
        for x in range(0, S_width):
            S[y, x] = np.sum(X[y:y+Y_height, x:x + Y_width]*Y)
    return S
```

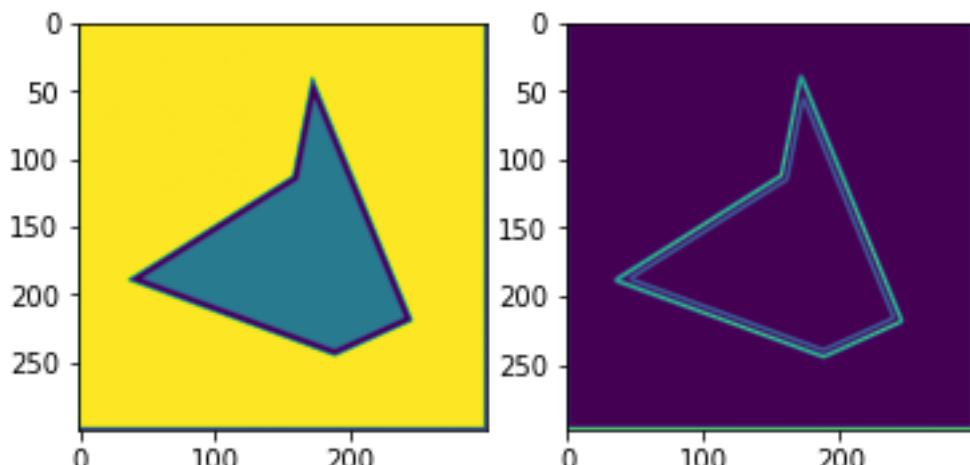
Como ejemplo, aplicaremos el conocido filtro de Sobel para encontrar los bordes de una imagen \mathbf{I} . En particular, Sobel aplica dos filtros sobre una imagen (horizontal y vertical):

$$hF = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -2 \end{pmatrix}; vF = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -2 & -2 & -1 \end{pmatrix}$$

Dando como lugar dos imágenes de salida $\mathbf{hS} = (\mathbf{I} * hF)(x,y)$, $\mathbf{vS} = (\mathbf{I} * vF)(x,y)$. La imagen filtrada se calcula con la fórmula siguiente:

$$S = \sqrt{hS^2 + vS^2}$$

Podemos ver su resultado en la siguiente figura:



El código fuente que incorpora los ejemplos anteriores se encuentra en:

Ejemplo: Operador de Convolución

Ver ficheros Python asociados: [Codigo9.py](#)

Conviene destacar que uno de los efectos del operador de convolución es que reduce en un mínimo de 2 elementos la señal de salida, por cada dimensión. En el ejemplo de la convolución 1D, el array X convolucionado con un filtro de 3 componentes da como resultado una señal convolucionada de 8 componentes. En el caso de la imagen, se reduce de 300×300 a 298×298 .

Este efecto se produce porque el filtro sólo se aplica sobre los elementos de la señal. Si el filtro tiene tamaño (impar) \mathbf{Ny} y la señal tiene longitud \mathbf{Nx} , la señal convolucionada tendrá un tamaño de $\mathbf{Ns} = \mathbf{Nx-Ny+1}$. Esto supone una limitación, sobre todo cuando el tamaño del filtro es elevado. No obstante, existen soluciones para paliar este problema, conocidas como **padding**.

Otro problema adicional reside en la alta necesidad de cómputo para realizar convoluciones en imágenes o volúmenes. En este caso, para reducir el tamaño de la imagen convolucionada (también el número de operaciones) podríamos desear no realizar la convolución sobre todos los píxeles de la imagen original: En lugar de ir pixel a pixel, podríamos ir de dos en dos y así reducir el tamaño de la imagen de salida a la mitad. A esta técnica se le conoce como **stride**.

Padding

El **padding** consiste en ampliar los bordes de la señal original un número de componentes dado, a fin de poder conseguir una señal de salida del tamaño deseado:

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
 0 & 3 & 3 & 4 & 4 & 7 & 0 & 0 \\ \hline
 0 & 9 & 7 & 6 & 5 & 8 & 2 & 0 \\ \hline
 0 & 6 & 5 & 5 & 6 & 9 & 2 & 0 \\ \hline
 0 & 7 & 1 & 3 & 2 & 7 & 8 & 0 \\ \hline
 0 & 0 & 3 & 7 & 1 & 8 & 3 & 0 \\ \hline
 0 & 4 & 0 & 4 & 3 & 2 & 2 & 0 \\ \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
 \end{array} * \begin{array}{|c|c|c|} \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 \end{array} = \begin{array}{|c|c|c|c|c|c|c|c|} \hline
 -10 & -13 & 1 & & & & & \\ \hline
 -9 & 3 & 0 & & & & & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
 \end{array}$$

$6 \times 6 \rightarrow 8 \times 8$

No obstante, existe aún el problema de con cuáles valores se rellenan los nuevos datos añadidos a la señal. Existen varias alternativas:

- | Rellenar con ceros: Los nuevos bordes contendrían valor igual a cero y la convolución se realizaría de la forma habitual.
- | Rellenar con los mismos valores existentes en el borde de la imagen.
- | Rellenar con función espejo:

3	5	1
3	6	1
4	7	9

No padding

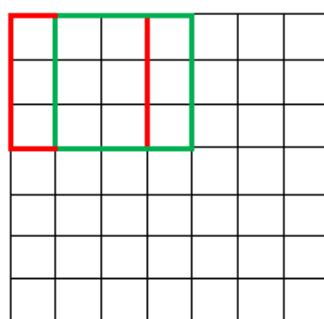
1	6	3	6	1	6	3
1	5	3	5	1	5	3
1	6	3	6	1	6	3
9	7	4	7	9	7	4
1	6	3	6	1	6	3

(1, 2) reflection padding

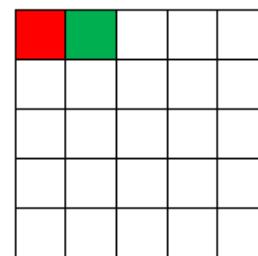
Stride

El **stride** consiste en aplicar selectivamente la convolución sobre un subconjunto de elementos de la señal original. Por ejemplo, en lugar de aplicar el filtro elemento a elemento, se trataría de aplicarlo cada 2 elementos, cada 3, etc. (según el valor del stride deseado). Las dos imágenes siguientes muestran el ejemplo de usar un stride 1 (primera imagen) o un stride 2 (segunda imagen):

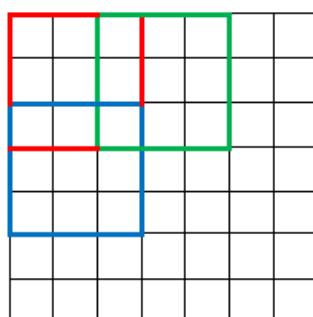
7 x 7 Input Volume



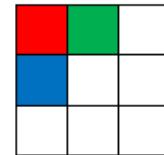
5 x 5 Output Volume



7 x 7 Input Volume



3 x 3 Output Volume



Tanto el **padding** como el **stride** son elementos fundamentales de las redes convolucionales.

Ejemplo: Operador de Convolución con padding y stride

Ver ficheros Python asociados: **Codigo10.py**

Redes convolucionales en Tensorflow

El operador de convolución se implementa dentro de TensorFlow a través de capas (**tipo Layer**). En particular, encontramos los siguientes tipos de capas convolucionales:

- | **Capa convolucional 1D.** Aplica tantos filtros como se indique a los datos de entrada. Se incluye la restricción de que todos los filtros deben tener el mismo tamaño, con el mismo tipo de **padding** y de **stride**. En particular, Tensorflow únicamente habilita 2 tipos de padding:
 - | **'valid'**: Tipo de convolución tradicional, sin modificar la señal de entrada.
 - | **'same'**: Incluye un padding de tipo 0's del tamaño necesario para que la señal de salida tenga exactamente el mismo tamaño que la de entrada.

Adicionalmente, se puede incluir una función de activación que se aplicará para cada valor de la señal resultante.

TensorFlow > API > TensorFlow v2.11.0 > Python

¿Te ha resultado útil?  

`tf.keras.layers.Conv1D` 

 [View source on GitHub](#)

1D convolution layer (e.g. temporal convolution).

Inherits From: [Layer](#), [Module](#)

 [View aliases](#)

```
tf.keras.layers.Conv1D(  
    filters,  
    kernel_size,  
    strides=1,  
    padding='valid',  
    data_format='channels_last',  
    dilation_rate=1,  
    groups=1,  
    activation=None,  
    use_bias=True,  
    kernel_initializer='glorot_uniform',  
    bias_initializer='zeros',  
    kernel_regularizer=None,  
    bias_regularizer=None,  
    activity_regularizer=None,  
    kernel_constraint=None,  
    bias_constraint=None,  
    **kwargs  
)
```

Capa convolucional 2D. Es la más utilizada para el tratamiento de imágenes. Contempla las mismas opciones que la capa convolucional 1D.

TensorFlow > API > TensorFlow v2.11.0 > Python

¿Te ha resultado útil?  

tf.keras.layers.Conv2D

 View source on GitHub

2D convolution layer (e.g. spatial convolution over images).

Inherits From: [Layer](#), [Module](#)

 [View aliases](#)

Main aliases

[tf.keras.layers.Convolution2D](#)

Compat aliases for migration

See [Migration guide](#) for more details.

[tf.compat.v1.keras.layers.Conv2D](#), [tf.compat.v1.keras.layers.Convolution2D](#)

```
tf.keras.layers.Conv2D(  
    filters,  
    kernel_size,  
    strides=(1, 1),  
    padding='valid',  
    data_format=None,  
    dilation_rate=(1, 1),  
    groups=1,  
    activation=None,  
    use_bias=True
```

| **Capa convolucional 3D.** Realiza convolución sobre volúmenes. Al igual que la anterior, comparte parámetros con la capa convolucional 1D.

TensorFlow > API > TensorFlow v2.11.0 > Python

¿Te ha resultado útil?  

tf.keras.layers.Conv3D



[View source on GitHub](#)

3D convolution layer (e.g. spatial convolution over volumes).

Inherits From: [Layer](#), [Module](#)

 [View aliases](#)

```
tf.keras.layers.Conv3D(  
    filters,  
    kernel_size,  
    strides=(1, 1, 1),  
    padding='valid',  
    data_format=None,  
    dilation_rate=(1, 1, 1),  
    groups=1,  
    activation=None,  
    use_bias=True,  
    kernel_initializer='glorot_uniform',  
    bias_initializer='zeros',  
    kernel_regularizer=None,  
    bias_regularizer=None,  
    activity_regularizer=None,  
    kernel_constraint=None,  
    bias_constraint=None,  
    **kwargs  
)
```

Otras opciones de Tensorflow son las convoluciones **Conv1DTranspose**, **Conv2DTranspose**, **Conv3DTranspose**, que realizan la convolución traspuesta (operación opuesta a la convolución, también conocida como **deconvolución**), o las capas recurrentes convolucionales como **ConvLSTM1D**, **ConvLSTM2D**, **ConvLSTM3D**, por citar algunas.

IMPORTANTE: En el caso de tratamiento de imágenes, normalmente estas suelen proporcionarse mediante una matriz 3D (**Y, X, C**), donde **Y,X** representan el alto y el ancho de la imagen, y **C** el **número de canales**. Las imágenes deben proporcionarse en este formato a las redes convolucionales de TensorFlow.

El siguiente ejemplo muestra el uso de la capa Conv2D para tratamiento de imágenes. En particular, el conjunto de datos **Fashion MNIST** de identificación de prendas de ropa.



Ejemplo: Capas convolucionales

Ver ficheros Python asociados: **Codigo11.py**

Normalmente, las capas convolucionales conviven dentro de un modelo con otro tipo de capas (por ejemplo, lineales o **capas densas**). No obstante, las capas densas (tipo Perceptrón) no pueden obtener como entrada datos multidimensionales como imágenes. Se deben incorporar, por tanto, otras capas de transformación para preparar los datos al procesamiento de una capa a otra. En el caso de capas convolucionales a capas densas, se utiliza la función **flatten**:

Ejemplo: Capas convolucionales y lineales

Ver ficheros Python asociados: **Codigo12.py**

Podemos observar cómo los parámetros a aprender por una red neuronal con capas convolucionales son elevados. Por este motivo, se suelen aplicar capas intermedias de resumen de información, de modo que se reduzcan los cálculos y se aprenda realmente las características abstractas de la información.

Pooling

El **pooling** es una técnica ampliamente utilizada en Deep Learning para reducir el tamaño de los datos procesados por una red (normalmente convolucional). Consiste en seleccionar un subconjunto de datos de una capa convolucional mediante un operador de resúmenes o **pooling**.

El **pooling**, a efectos de ejecución, se realiza elemento a elemento sobre la señal de entrada, aplicando la operación de resumen de datos sobre una máscara de tamaño predeterminado.

En Tensorflow/Keras disponemos de varios operadores de pooling:

AveragePooling (1D/2D/3D): Capas **AveragePooling1D**, **AveragePooling2D**, **AveragePooling3D**. Reduce la entrada de la capa realizando la media de los elementos dentro del filtro:

TensorFlow > API > TensorFlow v2.11.0 > Python

¿Te ha resultado

tf.keras.layers.AveragePooling2D



[View source on GitHub](#)

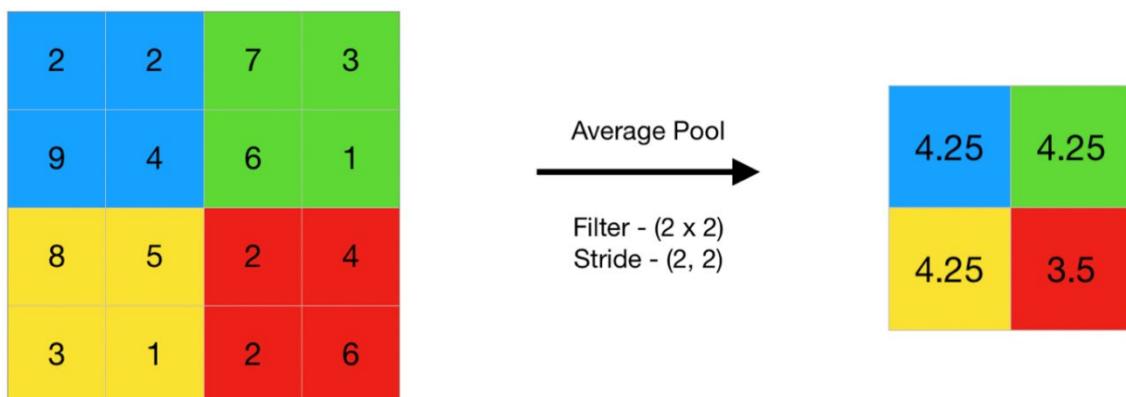
Average pooling operation for spatial data.

Inherits From: [Layer](#), [Module](#)

[+ View aliases](#)

```
tf.keras.layers.AveragePooling2D(  
    pool_size=(2, 2),  
    strides=None,  
    padding='valid',  
    data_format=None,  
    **kwargs  
)
```

Un ejemplo de Pooling por promedio sobre una matriz 2D, con una máscara de tamaño 2x2 y un **stride** de tamaño (2,2) sería el siguiente:



| MaxPooling (1D/2D/3D), Capas **MaxPool1D**, **MaxPool2D**, **MaxPool3D**. Reduce la entrada de la capa calculando el máximo de los elementos dentro del filtro:

TensorFlow > API > TensorFlow v2.11.0 > Python

¿Te ha resultado

tf.keras.layers.MaxPool2D

View source on GitHub

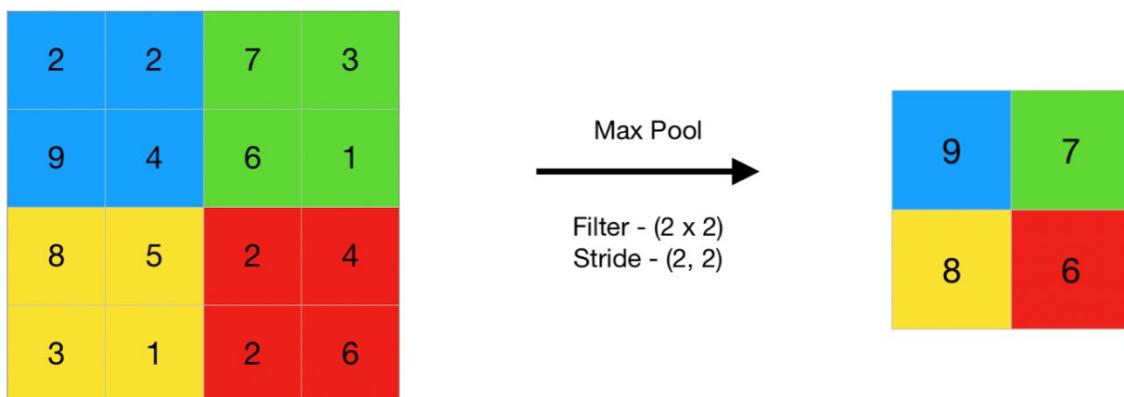
Max pooling operation for 2D spatial data.

Inherits From: [Layer](#), [Module](#)

View aliases

```
tf.keras.layers.MaxPool2D(  
    pool_size=(2, 2),  
    strides=None,  
    padding='valid',  
    data_format=None,  
    **kwargs  
)
```

Un ejemplo de Pooling por máximo utilizando una máscara de 2x2 con un **stride** de (2,2) sería el siguiente:



A continuación, se muestra un ejemplo de uso de las capas de **pooling** para entrenar una red neuronal convolucional profunda en el conjunto de datos de **Fashion MNIST**:

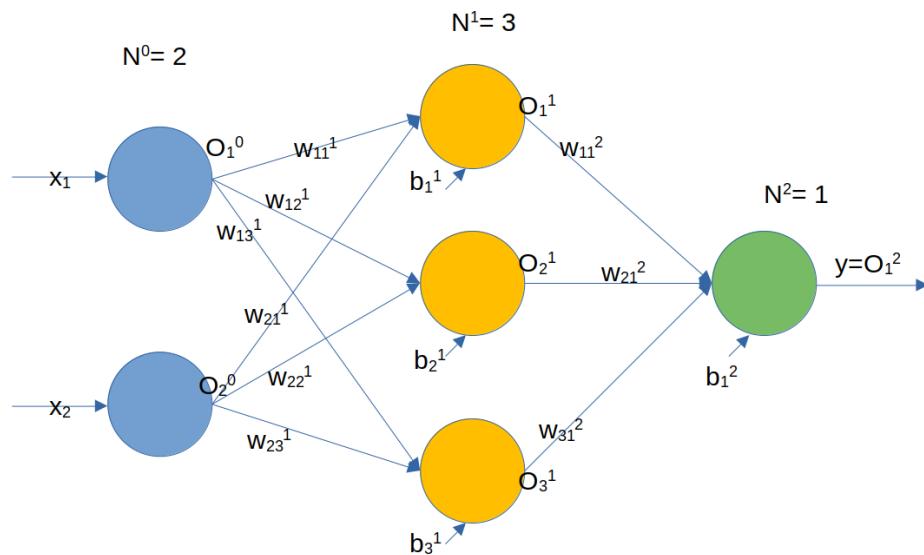
Ejemplo: Red neuronal convolucional

Ver ficheros Python asociados: **Codigo13.py**

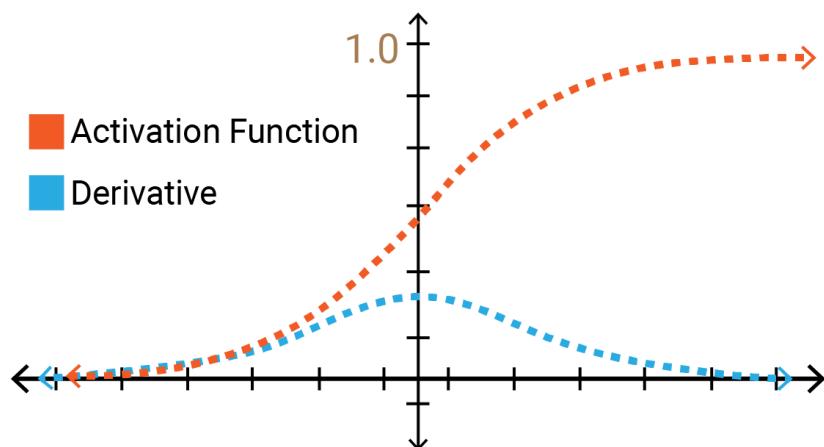
Redes neuronales (muy) profundas: Normalización y ResNet

Usualmente, las redes neuronales profundas toman su nombre de la existencia de múltiples capas entre la entrada y la salida. En este sentido, cabe recordar la regla de la cadena sobre la que se basan sus algoritmos de aprendizaje:

$$\frac{d J}{d w_{11}^1} = \frac{d J}{d O_1^2} \frac{d O_1^2}{d w_{11}^1} = \frac{d J}{d O_1^2} \frac{d O_1^2}{d z_1^2} \frac{d z_1^2}{d O_1^1} \frac{d O_1^1}{d z_1^1} \frac{d z_1^1}{d w_{11}^1}$$

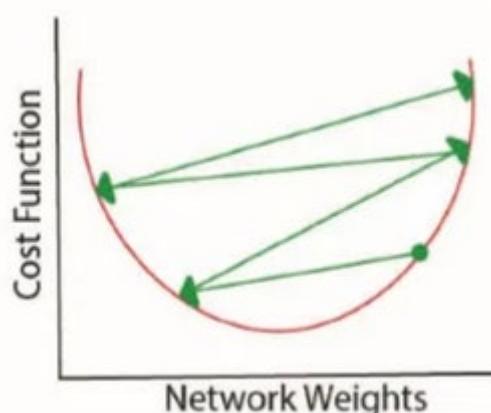


Cuando existen múltiples capas en secuencia, puede ocurrir el conocido como **problema del desvanecimiento del gradiente**, que se produce cuando el gradiente va tomando valores más pequeños al calcular el diferencial de un peso a modificar por el algoritmo de aprendizaje:



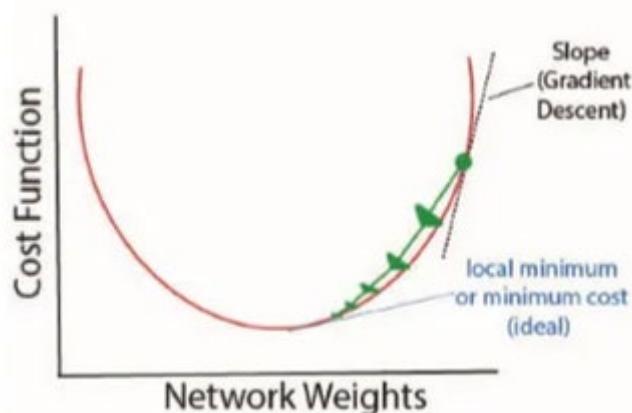
En redes profundas este problema se vuelve más acuciantante, tal como se puede deducir a partir de la regla de la cadena.

Por otra parte, otra situación que puede darse es justo la contraria, y que se conoce como **problema de la explosión del gradiente**. En este caso, la derivada de la función de pérdida con respecto a uno o más pesos puede ir creciendo indefinidamente, produciendo cambios bruscos en los pesos de la red y desfavoreciendo así el aprendizaje:



El problema de la explosión del gradiente puede atajarse a través de dos posibilidades:

- | **Modificación de la tasa de aprendizaje:** La reducción de la tasa de aprendizaje puede mejorar el problema en la mayoría de los casos:



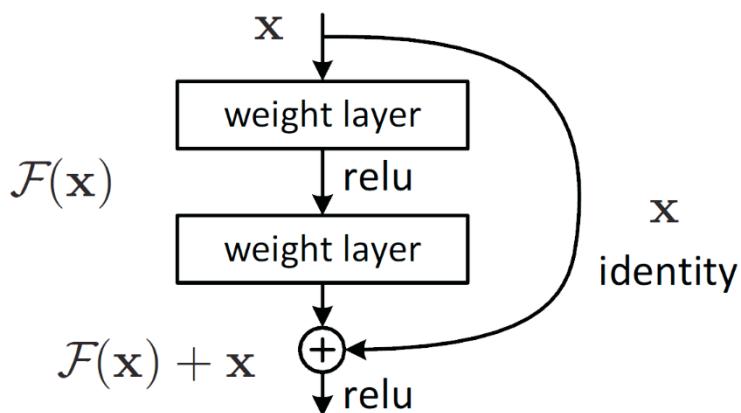
- | **Normalización:** Cuando las entradas a la capa tienen valores muy elevados (en valor absoluto), es necesario realizar una normalización de los mismos para evitar el cálculo de

gradientes con valores también muy elevados. En este caso, se suele normalizar de modo que los datos tengan media 0 y desviación típica 1. La normalización, no obstante, puede conducir a un mayor problema de desvanecimiento del gradiente.

La normalización está implementada en TensorFlow y Keras, a través de diversas opciones:

- | Normalización Batch: normalización para cada subconjunto de entrada que tiene la red. Implementada en la capa **keras.layers.BatchNormalization**.
- | Normalización de Capa: Normalización a valores [-1, 1] dentro de los atributos de cada patrón de entrada. La diferencia con la normalización batch reside en que la Batch actúa sobre el conjunto de datos batch introducido, mientras que la de capa actúa sobre cada patrón de forma individual. Implementada en la capa **keras.layers.LayerNormalization**.
- | Normalización de unidad: Normaliza cada patrón de entrada para que su norma sea igual a 1. Implementada en la capa **keras.layers.UnitNormalization**.

Tradicionalmente, el problema del desvanecimiento o degradado del gradiente ha sido el principal enemigo de las redes neuronales artificiales (en especial de las recurrentes). Las **redes de residuos (Residual networks, ResNet)** son una estructura de red profunda que permite abordar el problema del desvanecimiento del gradiente, reforzando capas más profundas de la red con salidas de capas previas:



Las redes de residuos pueden construirse con cualquier arquitectura siguiendo este principio, aunque al hablar de redes **ResNet** se asume normalmente una estructura similar a la siguiente:

- Entradas a capa ResNet **X**.
- Capa de convolución 1.
- Capa de normalización Batch.
- Capa de función de activación 1.
- Capa de convolución 2 para dar salida **F(x)**.
- Capa de agregación **F(x)+x**
- Capa de función de activación 2.

El siguiente ejemplo muestra cómo usar una ResNet-50 (profundidad de 50 capas ResNet) para resolver el problema de reconocimiento de fotografías (**ImageNet**):



Ejemplo: Red ResNet

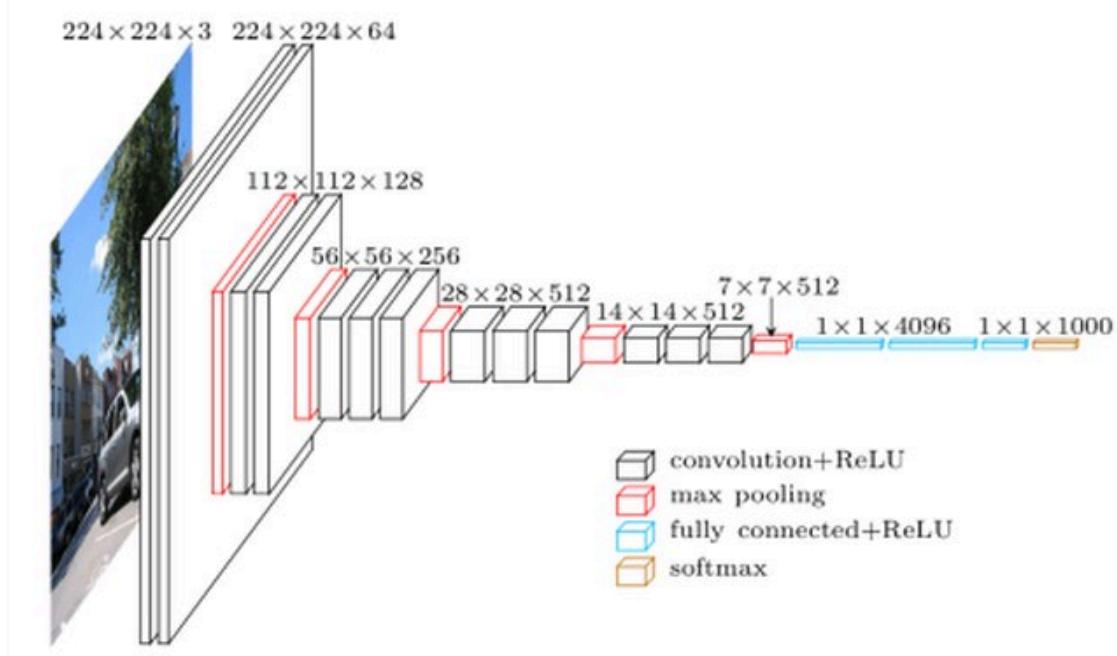
Ver ficheros Python asociados: **Codigo14.py**

Otros modelos preconstruidos

Tensorflow y Keras proporcionan un gran conjunto de modelos de redes profundas pre-entrenadas, que pueden utilizarse (o adaptarse) para realizar tareas principalmente de clasificación de imágenes. La red **ResNet50** que hemos utilizado en el apartado anterior es un ejemplo, aunque existen bastantes más:

Losses	Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Data loading	Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
Built-In small datasets	VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
Keras Applications	VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
Xception	ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
EfficientNet B0 to B7	ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
EfficientNetV2 B0 to B3 and S, M, L	ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ConvNeXt Tiny, Small, Base, Large, XLarge	ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
VGG16 and VGG19	ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet and ResNetV2	ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
MobileNet, MobileNetV2, and MobileNetV3	InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
DenseNet	InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
NasNetLarge and NasNetMobile	MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
InceptionV3	MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
InceptionResNetV2	DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
Mixed precision	DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
Utilities	DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
KerasTuner	NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
KerasCV	NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
KerasNLP	EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9
Code examples	EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
Why choose Keras?	EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
Community & governance	EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
Contributing to Keras	EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
KerasTuner	EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
KerasCV	EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
KerasNLP	EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6
	EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-	-
	EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-	-
	EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-	-
	EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-	-
	EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-	-

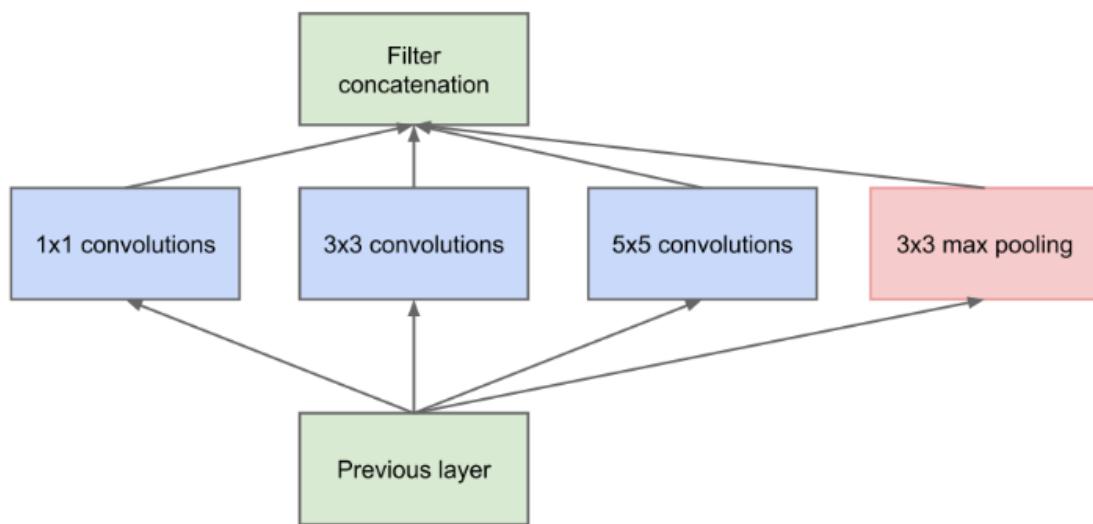
Algunos de ellos implementan de forma concreta algunas estrategias útiles a la hora de construir redes neuronales para reconocimiento y clasificación, como es el caso de las **Inception Networks** o **VGG**. El caso de **VGG** es una red pre-entrenada para clasificación. En el caso de **VGG-16**, consta de 16 capas:



El caso de las **Inception Networks**, aunque los modelos existentes en Keras son redes profundas concretas entrenadas para clasificación, se corresponden con una estrategia de reconocimiento más general. En particular, se aborda el problema de que varias imágenes que contengan el mismo contenido pueden tener una profundidad diferente:



La idea perseguida por las **Inception Networks** es utilizar filtros de diferente tamaño en la misma profundidad de la red, para luego agregar los resultados y obtener una clasificación más precisa:

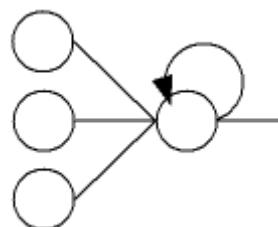


Otros modelos preconstruidos se centran en la optimización del tamaño de la red (que usualmente es muy elevado), para poder ser utilizados en dispositivos de bajas prestaciones (como teléfonos móviles, por ejemplo). Es el caso de **MobileNet**. En términos generales, podemos indicar que existen multitud de modelos y estrategias para resolver problemas de aprendizaje automático, tanto de clasificación como de regresión y que, cada uno de ellos, puede tomar una enorme cantidad de tiempo de estudio para llegar a ser un Maestro en la resolución de un problema concreto.

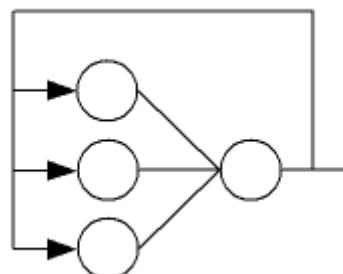
5. Redes Neuronales Recurrentes

Las redes neuronales recurrentes son un tipo especial de redes neuronales artificiales, orientadas al procesamiento de información temporal, donde la estructura de interconexión de las neuronas de la red contiene alguna conexión de retroalimentación (bien simétrica o bien asimétrica), como por ejemplo: Mapa autoorganizativo de Kohonen, red recurrente dinámica de Elman, red LSTM, etc. Podemos distinguir tres tipos elementales de conexiones recurrentes:

- | **Recurrencia local.** La salida de una neurona es retroalimentada a la propia neurona.



- | **Recurrencia global.** La salida de una neurona es entrada a otras neuronas de capas anteriores.



- | **Recurrencia mixta.** Cuando los dos tipos de recurrencia anteriores tienen lugar concurrentemente.

La recurrencia local es un caso relativamente fácil de tratar. Sin embargo, la recurrencia global tiene una serie de implicaciones mucho más complejas, las cuales están íntimamente relacionadas con las características y dificultades de los métodos de entrenamiento. Dependiendo del funcionamiento interno de una red neuronal recurrente, estas pueden ser clasificadas a su vez de la siguiente forma:

- | **Redes neuronales recurrentes estacionarias.** En este tipo de modelos, las conexiones recurrentes se utilizan de modo que la red pueda ser utilizada como una memoria asociativa. Algunos ejemplos de redes recurrentes estacionarias son el modelo de Hopfield o los mapas autoorganizativos de Kohonen.
- | **Redes neuronales recurrentes dinámicas.** En este tipo de modelos, las conexiones recurrentes sirven como memoria de almacenamiento de los patrones de entrada a lo largo del tiempo. Algunos ejemplos de redes neuronales recurrentes dinámicas son el modelo de Elman, las redes recurrentes completas o las Long-Short Term Memory (LSTM)

Las redes neuronales recurrentes se caracterizan porque, ante una misma entrada de datos, pueden proporcionar distinta salida dependiendo de una **información de contexto**, almacenada en los pesos de las conexiones recurrentes de la red. Por este motivo, son especialmente adecuadas para el procesamiento de información temporal, como puede ser la evolución de condiciones meteorológicas, el análisis de textos, modelado de mercados de valores, predicción del consumo energético, etc. Podemos decir que la información aprendida por una red neuronal se basa en el análisis de series temporales.

Series Temporales y redes neuronales recurrentes

Una serie temporal $X(t) = \{x(1), x(2), x(3), \dots, x(t)\}$ es una secuencia de observaciones de un fenómeno dado, muestreadas **periódicamente** e **indexadas** en el tiempo. Ejemplos de series temporales son la evolución de los precios de los mercados de valores, históricos del uso de la electricidad, históricos meteorológicos, la realización de gestos o comportamientos, la señal de un electrocardiograma, etc. Desde el punto de vista de un sistema de Aprendizaje Automático, existen varias aplicaciones tradicionales de series temporales donde los modelos pueden aportar algún valor añadido :

- | **Clasificación de series temporales:** Consiste en asignar un valor de clase a una secuencia de datos. Por ejemplo: Detección de comentarios positivos o negativos, identificación de ADN de especies, reconocimiento de gestos, etc.
- | **Predicción de series temporales:** Consiste en, dada una secuencia de valores numéricos históricos muestreados sobre un fenómeno, tratar de aproximar los valores futuros de la serie. Por ejemplo: Predicción meteorológica, predicción de ventas de un producto, predicción del consumo energético, etc.
- | **Detección de anomalías:** Es un sub-conjunto del problema de clasificación. Consiste en detectar anormalidades en una secuencia de datos. Por ejemplo: Detección de fallos de sensores en un complejo industrial, detección de fraudes en el uso de tarjetas de crédito, etc.

En el procesamiento de series temporales con redes neuronales artificiales, existen varios tipos de patrones (x,y) de entrada/salida a la red, dependiendo del problema a tratar:

- | **La serie temporal al completo es un patrón de entrada.** Es un caso bastante común en el problema de clasificación donde, dada una secuencia de mediciones, debemos asignar una clase a la serie de datos. Por ejemplo: Reconocimiento de gestos, clasificación de perfiles de usuarios según sus visitas a una web, etc.
- | **Un subconjunto de la serie temporal es un patrón de entrada.** Es un caso común del problema de predicción, aunque también hay aplicaciones de uso en clasificación y detección de anomalías. En el caso del problema de predicción, se tratará de aproximar cuál será el valor futuro de la serie temporal, $x(t)$, dados valores anteriores de la serie $\{x(t-1), x(t-2), x(t-3), \dots\}$. Por tanto, la misma serie temporal juega el rol de entrada y de salida simultáneamente (aunque en instantes de tiempo distintos).

Funcionamiento general de una red neuronal recurrente dinámica

Al contrario que en las redes neuronales de tipo **feedforward**, donde todo el patrón de datos se le presenta a la red en un único **forward pass** como atributos de entrada al modelo, en redes neuronales recurrentes el patrón de datos se presenta a la red secuencialmente. El número de pasos o **forward pass** que hay que realizar depende de la longitud de la serie temporal de entrada. No obstante, hay unos cuantos elementos comunes que son extensivos a todos los modelos de redes neuronales recurrentes dinámicas :

- | En cada instante de tiempo **t**, La red mantiene un **estado interno $S(t)$** , que modela el **contexto** actual de la información, codificado a través de las conexiones recurrentes y sus pesos. En el instante de tiempo inicial **t=0**, el estado **$S(0)$** se encuentra **reseteados a un valor por defecto** (normalmente valor 0).
- | Para procesar una serie temporal **{ $x(0), x(1), x(2), \dots, x(T)$ }**, en cada instante de tiempo **t** la red toma como entrada el valor de la serie **$x(t)$** . Con este valor:
 - | Se calcula la salida de la red en ese instante, como **$O(t)=F(x(t), S(t))$** .
 - | Se actualiza el estado interno de la red para el instante de tiempo siguiente, como **$S(t+1)=G(x(t), S(t))$** .

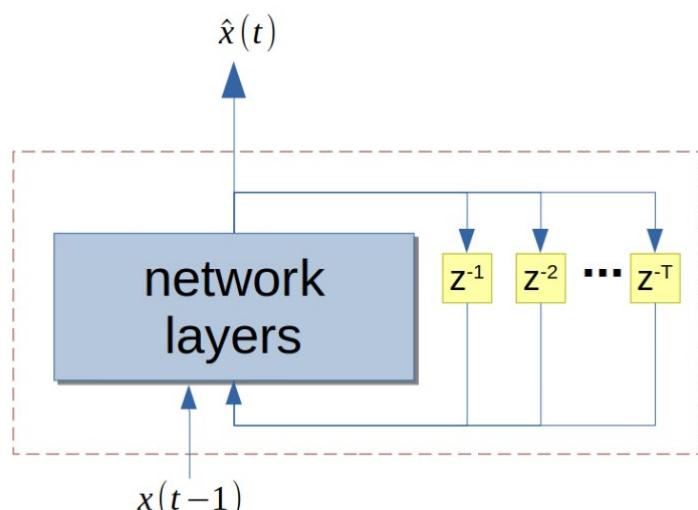
Por tanto, la salida de una red neuronal recurrente dinámica es también una serie temporal **{ $O(0), O(1), O(2), \dots, O(T)$ }**. La interpretación de estos valores depende de la aplicación o del problema a resolver en cuestión. En problemas de clasificación, es habitual obtener el valor **$O(T)$** de la salida en el último instante de tiempo como el valor de clase a asignar a la secuencia. Al contrario, en problemas de predicción se suele modelar la serie completa como la siguiente predicción:

$$x'(t + 1) = O(t) = F(x(t - 1), x(t - 2), x(t - 3), \dots, S(t))$$

Es decir, la salida de la red se corresponde con el siguiente valor de la serie a predecir.

Redes NAR (Non-linear Auto-Regressive neural network)

Las redes NAR son modelos específicos para abordar problemas de predicción de series temporales. Están fundamentados en el análisis clásico de series temporales con modelos lineales auto-regresivos (AR), y extienden su capacidad para poder abordar problemas donde los datos pueden tener dependencias no lineales entre sí. Es un modelo simple para abordar por primera vez el concepto de red neuronal recurrente, dado que se trata básicamente de una red **feedforward** con una conexión de retroalimentación desde la salida de la red hasta la capa de entrada :



La red **NAR** trata de aproximar el valor de **x(t)** en función de **T** valores anteriores de la serie. Es decir:

$$x'(t) = F(x(t-1), x(t-2), x(t-3), \dots, x(t-T))$$

Para ello, toma como entrada el valor de la serie temporal en el instante **t-1**, **x(t-1)**, y utiliza el contexto para "recordar" los valores pasados de la serie y así predecir el futuro valor que, de forma óptima, será **x(t)=x'(t)**.

Si la red está correctamente entrenada, entonces las aproximaciones de la salida de la red, x' , para los instantes de tiempo anteriores $x(t-2)=x'(t-2)$, $x(t-3)=x'(t-3)$, etc., tendrán un error lo suficientemente pequeño como para poder ser utilizadas en la predicción. Por tanto, la red almacena sus salidas y las retro-alimenta a la red como entradas en los siguientes instantes de tiempo, hasta un horizonte temporal T dado como hiperparámetro:

$$x'(t) = F(x(t-1), x'(t-2), x'(t-3), \dots, x'(t-T))$$

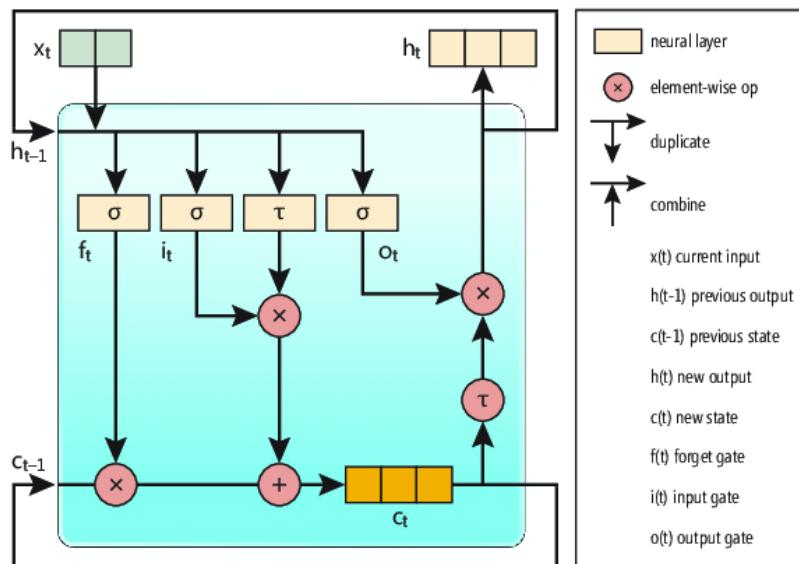
En el siguiente código fuente, se analiza la serie temporal **AirPassengers**, que contiene el número de pasajeros de avión (en miles) en EEUU desde 1949 hasta 1960. Como ejercicio, tomaremos la serie entre 1949 y 1959 y realizaremos la predicción para 1960.

Ejemplo: Red NAR

Ver ficheros Python asociados: [Codigo15.py](#)

Redes LSTM

Las redes **Long-Short Term Memory (LSTM)** son un tipo de redes recurrentes dinámicas con bastante popularidad hoy día. Su principal ventaja reside en el hecho de poder memorizar la serie temporal en el corto y en el largo plazo, algo que es necesario en muchas tareas tanto de clasificación como de predicción. El modelo de neurona LSTM consta de varias componentes :



- | **Input Gate $i(t)$.** Contiene la activación de entrada, dados los datos de la serie temporal en el instante actual $x(t)$ y el estado de la neurona $h(t-1)$
- | **Output Gate $o(t)$.** Contiene la activación de la salida de la neurona, dependiendo de la entrada actual y el estado interno de la neurona $c(t)$.
- | **Forget Gate $f(t)$.** Contiene la activación para el mecanismo de eliminación de información antigua de la neurona.

Las fórmulas básicas que rigen el comportamiento de una neurona LSTM son las siguientes :

$$f(t) = \sigma_g(W_f x(t) + U_f h(t-1) + b_f)$$

$$i(t) = \sigma_g(W_i x(t) + U_i h(t-1) + b_i)$$

$$o(t) = \sigma_g(W_o x(t) + U_o h(t-1) + b_o)$$

$$z(t) = \sigma_c(W_z x(t) + U_z h(t-1) + b_z)$$

$$c(t) = f(t) \odot c(t-1) + i(t) \odot z(t)$$

$$h(t) = o(t) \odot \sigma_h(c(t))$$

Donde los símbolos σ denotan funciones de activación, W y U denotan pesos de las conexiones, b denotan bias y \odot denota al producto de Hadamard (multiplicación elemento a elemento de arrays).

Al contrario que las redes NAR, y debido a su popularidad, las redes LSTM están implementadas nativamente dentro de Tensorflow/Keras. El siguiente ejemplo muestra el uso de este modelo de red para resolver el problema de **Air Passengers** previamente descrito en el apartado anterior.

Ejemplo: Red LSTM

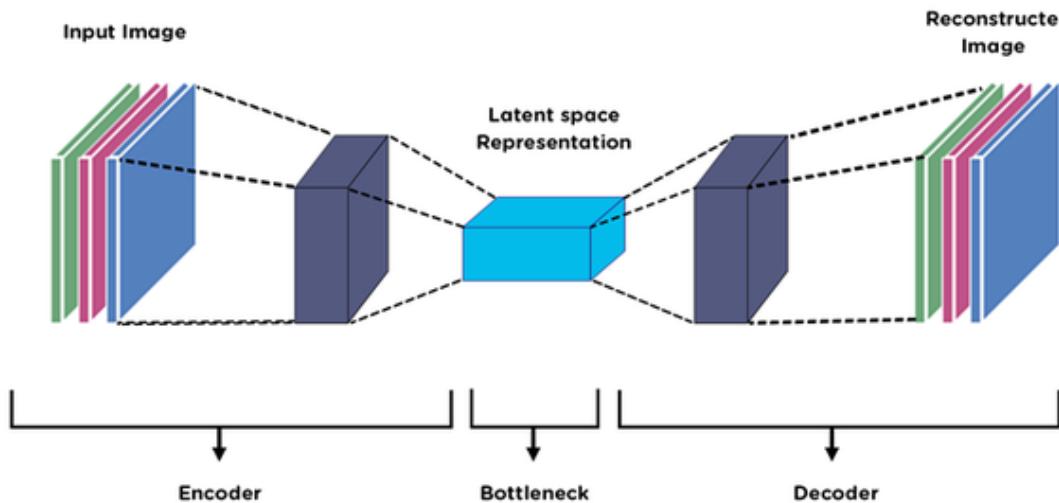
Ver ficheros Python asociados: [Codigo16.py](#)

6. Autoencoders

Las redes neuronales profundas denominadas **autoencoders** se utilizan principalmente para extracción de características o elementos relevantes de los patrones de un conjunto de datos. No obstante, se engloba dentro del **aprendizaje no supervisado**, dado que únicamente requieren de patrones de entrada. Un **autoencoder** tiene la tarea de reducir el espacio de entrada al denominado **espacio de características o espacio latente**, que represente fielmente (pero con un menor número de datos) cada patrón de entrada.

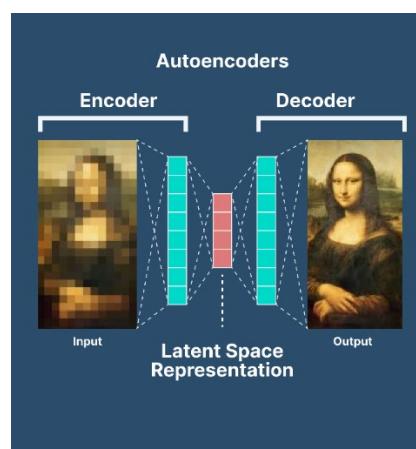
Aunque un **autoencoder** se englobe dentro del aprendizaje supervisado, su entrenamiento se realiza de forma **supervisadas**, dando como salida el mismo patrón de entrada; es decir, se presenta un patrón entrada/salida **(x,y), donde y=x**. La idea subyacente de los autoencoders es generar capas secuenciales en la red neuronal que cada vez tengan un menor número de parámetros hasta la capa de características conocida como **feature vector** para, seguidamente, realizar la operación inversa y tratar de reproducir la misma entrada utilizando únicamente el **feature vector** como dato inicial. Esto da a la red **autoencoder** una arquitectura fija con 3 componentes:

- | **Encoder:** Encargado de reducir la dimensionalidad de los datos de entrada.
- | **Feature vector:** Encargado de representar la información mínima necesaria para poder reconstruir la entrada.
- | **Decoder:** Encargado de reconstruir la entrada a partir del **feature vector**.



En Deep Learning, es común utilizar **autoencoders** con varios propósitos:

- | Utilizar la representación del espacio latente como entrada a otros modelos, que trabajen sobre una abstracción de los datos de entrada.
- | Utilizar la representación dada por el decoder para, por ejemplo, realizar tareas de filtrado de datos (eliminación de ruido en imágenes), reconstrucción (inferir objetos en una imagen partiendo de información parcial), etc.



Los **autoencoders** son, más que una estructura de red (como NAR o LSTM, que tienen arquitecturas fijas), una metodología de trabajo para reducción de la dimensionalidad de los datos. Es muy sencillo construir un **autoencoder** en TensorFlow/Keras, conociendo esta metodología de base. En particular, el siguiente ejemplo muestra cómo elaborar un autoencoder para representar el conjunto de datos de **Digits MNIST**:

Ejemplo: Red AutoEncoder

Ver ficheros Python asociados: [Codigo17.py](#)

Un tipo particular de autoencoder es el **Variational Autoencoder**. En particular, los **autoencoders** suelen tener un problema de sobre-aprendizaje, que puede mitigarse mediante la modificación del espacio latente con un ruido adicional:

Espacio Latente' = Espacio Latente + ruido

Con esta estrategia, se comprueba que aumenta la capacidad de generalización de los autoencoders pero que, además, puede tener aplicaciones comerciales adicionales, como por ejemplo la **generación automática de contenido**. Esta capacidad, no obstante, se analizará en el apartado "**Generative Adversarial Networks**" de este documento.

7. Aprendizaje por refuerzo profundo

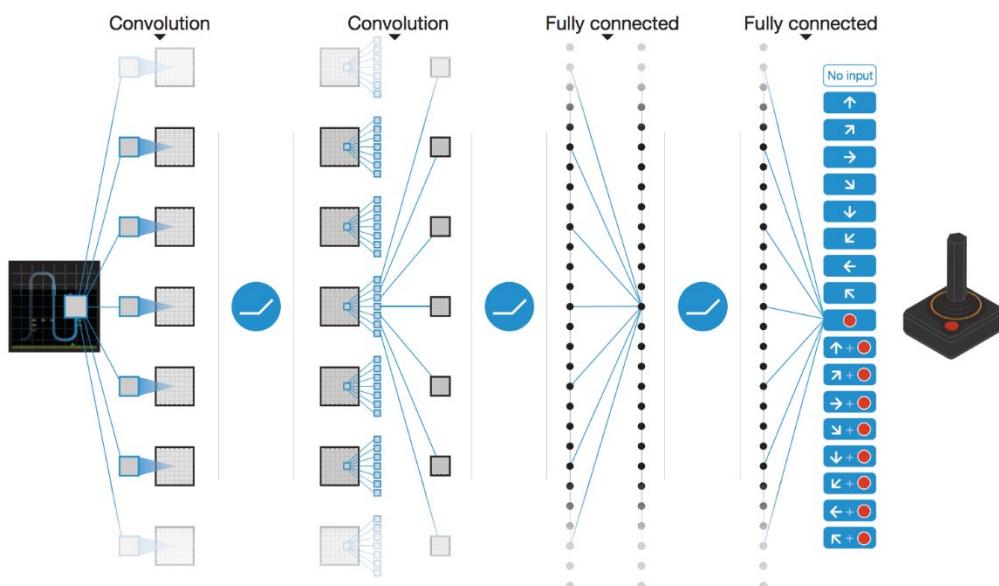
El aprendizaje por refuerzo profundo ha tenido una gran expansión desde el año 2014, cuando se consiguió adaptar el algoritmo de Q-Learning (visto en el módulo anterior) con redes neuronales convolucionales profundas para aprender a jugar a diversos videojuegos, teniendo como entrada las imágenes del videojuego y como salida las acciones a realizar.



Desde entonces, han surgido una gran cantidad de algoritmos de aprendizaje para el caso del aprendizaje por refuerzo, tanto basados en Q-Learning como en técnicas de Policy Gradient, entre los que destacamos las Deep Q-Networks, algoritmo REINFORCE, modelos Actor-Critic, algoritmo Proximal Policy Optimization (PPO), etc. En este apartado, y por dar continuación al algoritmo de Q-Learning estudiado en el módulo anterior, vamos a centrar nuestros esfuerzos en **Deep Q-Learning**.

Deep Q-Networks

Las redes **Deep Q-Networks (DQN)** se utilizan en aprendizaje por refuerzo profundo para aprender, a partir de la experiencia del agente, la tabla de la **función Q**. Suponiendo un conjunto de acciones **A** discreto, una Deep Q-Network tiene como entrada la percepción/observación del entorno en el instante actual, **s(t)**, y da como salida todos los valores Q del par **Q(s,a)** para las acciones **{a}** de **A**:



Las principales ventajas de las Deep Q-Networks frente al enfoque de Q-Learning clásico son numerosas :

- | Los estados se pueden representar como arrays multidimensionales.
- | Se permiten valores continuos para los atributos de un estado.

- | Diferentes tipos de características permitidas como variables de estado (continuo, discreto, categórico...).
- | El tamaño de la tabla Q queda supeditado al diseño de la red.
- | Existe una gran variedad de técnicas y algoritmos dentro del área de las redes neuronales.
- | Se facilita la adaptabilidad en la construcción de diferentes modelos.

El aprendizaje de una red Q-network persigue optimizar la regla de actualización de la función Q :

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

En particular, se trata de minimizar el error de diferencia temporal :

$$y^{target} = r + \gamma \max_{a'} Q(s', a')$$

$$y = Q(s, a)$$

$$MSE(y^{target}, y) = \frac{1}{N} \sum (y - y^{target})^2$$

No obstante, dado que en el aprendizaje por refuerzo se presentan los datos de aprendizaje como secuencias **{s(t), a(t), r(t), s(t+1), a(t+1), r(t+1), ...}**, existe una fuerte correlación en los datos de entrada ($s(t+1)$ depende de $s(t)$ y $a(t)$, por ejemplo), lo que dificulta el aprendizaje de la red. Además, el uso de la red para predecir los valores **y** e **y^{target}** puede también provocar un pobre aprendizaje debido a la correlación existente.

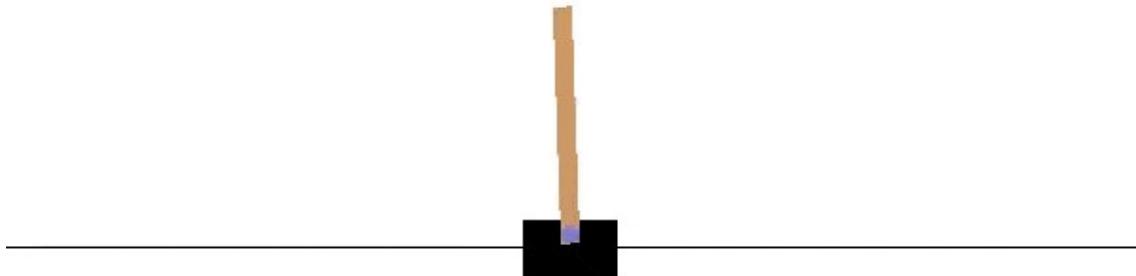
Para solucionar estos problemas, **Deep Q-Learning** aporta dos soluciones que se complementan entre sí :

- | El uso de un Buffer (denominado técnicamente como **replay buffer**) donde se van guardando experiencias. En cada paso de aprendizaje, se muestrea un batch de elementos del buffer aleatoriamente seleccionados (se rompe así la dependencia de los datos).
- | El uso de una red neuronal auxiliar o **target**, que consiste en «**una copia antigua**» de la red Q-Network a aprender, para predecir los valores y^{target} . (se rompe así la dependencia entre salidas de la red). Tras un número de iteraciones dado, la red target se actualizaría a las nuevas versiones de la red aprendida.

Con estas modificaciones, el algoritmo Q-Learning adaptado a Deep Q-Networks queda como sigue :

1. Inicializar entorno
2. Inicializar DQN con parámetros \mathbf{W} y target con $\mathbf{W}' = \mathbf{W}$.
3. Inicializar buffer con número mínimo de patrones desde DQN.
4. Mientras no se generen **MaxEpisodes** episodios, hacer:
 - 1) Realizar un paso en el entorno y guardar (s,a,r,s',done) en el buffer. Pasar $s=s'$ o reiniciar entorno si el actual termina.
 - 2) Muestrear batch de experiencias del buffer
 - 3) Calcular $\mathbf{Q}_{\mathbf{W}'}^{\text{target}}(\mathbf{s},\mathbf{a}) = r + \gamma \max_{\mathbf{a}'} \mathbf{Q}_{\mathbf{W}'}(\mathbf{s}',\mathbf{a}')$ en el batch
 - 4) Calcular $\mathbf{Q}_{\mathbf{W}}(\mathbf{s},\mathbf{a})$ en el batch
 - 5) Calcular J y **optimizar \mathbf{W}** .
 - 6) Sincronizar $\mathbf{W}' = \mathbf{W}$ si han pasado un número de iteraciones dado.
5. Devolver \mathbf{W}

El ejemplo siguiente muestra cómo utilizar las redes **DQN** para resolver un problema simple de OpenAI Gym: CartPole.



Ejemplo: Deep Q-networks

Ver ficheros Python asociados: **Codigo18.py**

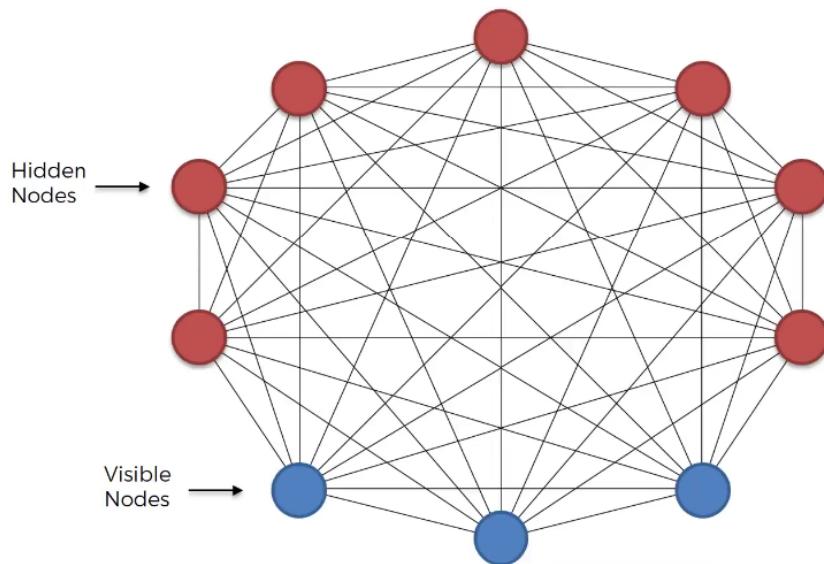
8. Máquinas de Boltzmann

Las Máquinas de Boltzmann son un tipo de redes neuronales cuyo funcionamiento está basado en el concepto físico de energía. En particular, se basan en el principio de distribución de Boltzmann, que se resume en que el estado de un sistema en un instante de tiempo dado depende de la energía que contiene y de la temperatura a la que opera. Por este motivo, las Máquinas de Boltzmann difieren sustancialmente de los modelos de redes neuronales estudiados hasta ahora dado que, en lugar de utilizar terminología estrictamente matemática, utilizan conceptos de física. En particular, la adaptación del modelo de Boltzmann a redes neuronales asume el símil de que la energía equivale a la diferencia entre la respuesta esperada del sistema y la respuesta dada por el mismo (tradicionalmente en el área de redes neuronales denominado como error de predicción). La probabilidad de que el sistema se encuentre en el estado i se calcula con la ecuación de la distribución de Boltzmann:

$$p_i = \frac{e^{-\epsilon_i/kT}}{\sum_{j=1}^M e^{-\epsilon_j/kT}}$$

Donde k es un término constante, T es la temperatura del sistema, y ϵ_i la energía del sistema.

Otra diferencia sustancial entre las redes neuronales estudiadas hasta ahora y las Máquinas de Boltzmann genéricas reside en que no existe el concepto de capa neuronal, ni tampoco de entradas ni salidas. Al contrario, se utiliza el concepto de **nodos visibles** y **nodos invisibles** u **ocultos**. Los **nodos visibles** se corresponderían con los nodos de recepción de entradas de la máquina de Boltzmann que, a su vez, también actuarían como salida.



Las Máquinas de Boltzmann se utilizaban generalmente en el aprendizaje no supervisado, para encontrar dependencias entre los datos de un conjunto binarizado. Las conexiones entre las neuronas son **simétricas**, y cada neurona únicamente puede contener un estado binario (0/1), que se aproxima a partir de una distribución de probabilidad de Bernouilli típicamente. Notaremos como \mathbf{h}_i a las neuronas ocultas, como \mathbf{v}_i a los vértices/nodos visibles, y como s_i a la salida de cada neurona. En cada instante de tiempo, cada neurona i calcula la probabilidad de cambiar de estado de acuerdo a la importancia de las conexiones con sus neuronas vecinas j , w_{ij} , y un bias b_i de la siguiente forma:

$$z_i = \sum_j s_j w_{ij} + b_i$$

$$p(s_i = 1) = \frac{1}{1 + e^{-z_i}}$$

Durante la evolución del sistema a lo largo del tiempo, la teoría indica que el sistema converge a una distribución de Boltmann donde la energía del estado final \mathbf{v} es relativa a la energía de todos los posibles estados \mathbf{u} :

$$P(\mathbf{v}) = \frac{e^{-E(\mathbf{v})}}{\sum_{\mathbf{u}} e^{-E(\mathbf{u})}}$$
$$E(\mathbf{v}) = - \sum_i s_i^v b_i - \sum_{i < j} s_i^v s_j^v w_{ij}$$

En la ecuación anterior de representación de la energía $E(\mathbf{v})$, el valor s_i^v es el valor/estado asignado a la neurona i en el estado del sistema \mathbf{v} .

En la práctica, la implementación de las Máquinas de Boltzmann a gran escala requiere de una gran cantidad de recursos de cómputo, por lo que se suelen emplear modelos reducidos denominados **Máquinas de Boltzmann Restringidas (Restricted Boltzmann Machines, RBM)**.

La única diferencia existente entre una Máquina de Boltzmann genérica y una **RBM** reside únicamente en las conexiones permitidas. En particular, las **RBM** únicamente permiten conexiones entre neuronas de distinto tipo (las ocultas no están conectadas entre sí, las visibles no están conectadas entre sí). En este caso, la energía de un estado del sistema \mathbf{v} viene determinada por la siguiente expresión:

$$E(\mathbf{v}, \mathbf{h}) = - \sum_i v_i a_i - \sum_j h_j b_j - \sum_{i,j} v_i h_j w_{ij}$$

En la ecuación anterior, los términos a_i , b_j se refieren a los valores de **bias** de las neuronas visibles y ocultas, respectivamente.

Los valores \mathbf{v}_i , \mathbf{h}_j se refieren a la activación de las neuronas visibles y ocultas, respectivamente. Utilizamos la doble terminología para diferenciar correctamente entre ambos tipos de neuronas.

También diferenciaremos entre los términos de probabilidad de activación de una neurona dado un estado del sistema \mathbf{v} como:

$$p(h_j = 1 | \mathbf{v}) = \frac{1}{1 + e^{-(b_j + \sum_i w_{ij} v_i)}}$$

$$p(v_i = 1 | \mathbf{v}) = \frac{1}{1 + e^{-(a_i + \sum_j w_{ij} h_j)}}$$

Debido a su estructura y funcionamiento, el proceso de aprendizaje de una RBM difiere sustancialmente del aprendizaje estudiado en apartados anteriores. En particular, el aprendizaje de RBM tiene dos pasos:

Muestreo de Gibbs: En primer lugar, se introducen los valores de entrada en las neuronas visibles y, en base a ellos, se calcula el estado de las neuronas ocultas. Esto da lugar al estado inicial del sistema \mathbf{v}_0 . A continuación, se recalculan los valores de las neuronas visibles en función de los nuevos valores de las neuronas ocultas. El proceso se debe repetir, en teoría, un número de iteraciones \mathbf{k} dado, aunque en la práctica se suele repetir un número reducido de veces (una o ninguna). El resultado es el estado final del sistema tras \mathbf{k} iteraciones, denominado \mathbf{v}_k . Es el equivalente al **forward pass** de las redes estudiadas anteriormente.

Contraste de divergencia: En este paso, se actualizan los pesos de las conexiones neuronales considerando los estados inicial \mathbf{v}_0 y final \mathbf{v}_k del sistema:

$$W = W + \lambda \Delta W$$

$$\Delta W = v_0 \otimes p(h_0 | v_0) - v_k \otimes p(h_k | v_k)$$

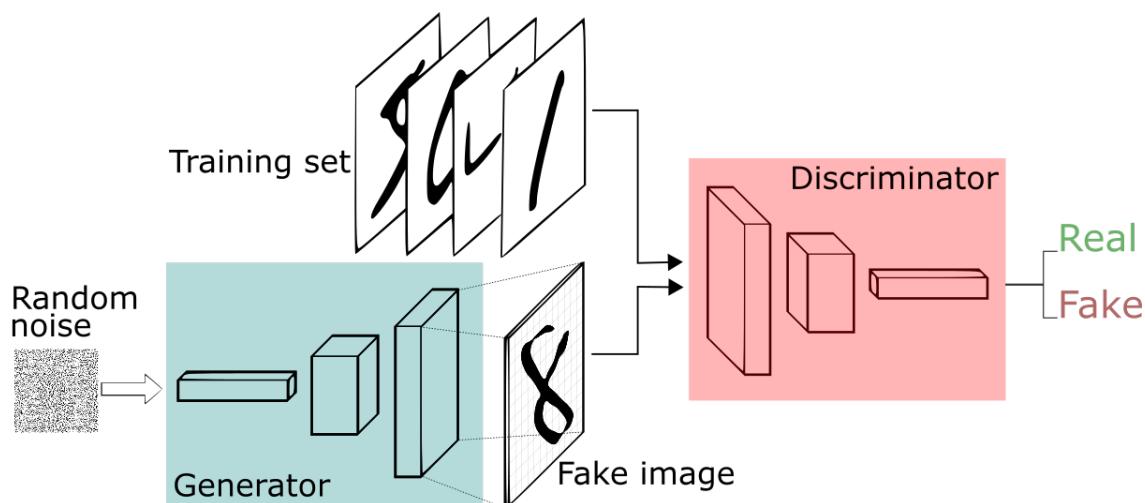
Actualmente, las Máquinas de Boltzmann se han visto desplazadas en el área de Deep Learning por las conocidas como **Deep Belief Networks** o **redes bayesianas profundas**.

9. Generative Adversarial Networks

Hasta el momento, en todo el curso hemos estudiado modelos discriminadores; es decir, clasificadores que aprenden a distinguir datos que pertenecen a una clase u otra de elementos. Otro tipo de aprendizaje se basa en **modelos generadores**. Este tipo de modelos se basan en la idea de que, en el aprendizaje no supervisado, podría entenderse que un modelo que ha aprendido a abstraer los datos de una clase quizás podría generar también instancias o datos de esa misma clase con el conocimiento adquirido. Por ejemplo, un modelo que ha aprendido a distinguir imágenes de rostros de personas dispone del suficiente conocimiento como para poder generar sintéticamente un nuevo rostro.

El estudio de las redes **Generative Adversarial Networks (GANs)** es una metodología basada en la idea de modelos generadores. En particular, las GANs constan de dos tipos de redes:

- | **Red generadora:** Encargada de generar una nueva instancia de los datos partiendo de información de características iniciales.
- | **Red discriminadora:** Se encarga de evaluar el dato generado por la red generadora de modo que pueda considerarse como "real" o como "falso".



Por tanto, en el modelo co-existen dos redes neuronales artificiales diferentes: Una que trata de generar contenido falso (generador) y "engaños" a un evaluador (discriminador), y un evaluador que trata de perfeccionar la técnica para distinguir entre contenido real y contenido falso.

El proceso de funcionamiento de un modelo GAN es como sigue:

- | Un vector de características inicial (normalmente ruido aleatorio) se introduce como entrada a un generador, que genera un contenido falso.
- | El contenido falso se mezcla con contenido real, y todo el conjunto de datos se proporciona como entrada al discriminador, que clasifica de forma binaria qué contenido es real y qué contenido es falso.

La salida del discriminador puede ayudar en el aprendizaje tanto del propio discriminador (fallos/aciertos al clasificar contenido falso) como al generador, para mejorar su competencia. Un ejemplo de su implementación y uso en Tensorflow para el problema **Digits MNIST** se proporciona en el siguiente código fuente:

Ejemplo: Generative Adversarial Networks

Ver ficheros Python asociados: **Codigo19.py**

En internet puede encontrarse múltiples cantidades de ejemplos de GAN con fines lúdicos: Generación de rostros de personas o en general imágenes, generación de texto en Twitter, etc. No obstante, el interés científico y tecnológico de las GAN va mucho más allá, pudiendo encontrar aplicaciones en campos como las técnicas de Data Augmentation, generación de personajes para dibujos animados, traducción, descripción textual de imágenes, detección de SPAM, etc.

10. Alternativas a TensorFlow. Pytorch

Pytorch se presenta como una gran alternativa a Tensorflow. Dispone de todas las características de Tensorflow y, además, una API estable a lo largo del tiempo, lo que hace que la migración entre versiones sea más sencilla. Además, Pytorch también requiere de una curva de aprendizaje similar a Tensorflow, aunque su orientación tradicional ha sido más científica, al contrario que Tensorflow, que dispone de una gran cantidad de herramientas para despliegue de modelos tanto en grandes servidores como en dispositivos empotrados.

El cambio de Tensorflow a Pytorch es relativamente sencillo, basta con conocer la API. Particularmente, la creación de un tensor en ambos sistemas sería:

En Tensorflow:

```
import tensorflow as tf

rank_2_tensor = tf.constant([[1, 2],
                           [3, 4],
                           [5, 6]], dtype=tf.int32)
```

En Pytorch:

```
import torch

rank_2_tensor = torch.tensor([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=torch.int32)
```

No obstante, la distinción realizada en Tensorflow sobre tensores constantes y variables no se considera una limitación en Pytorch:

En Tensorflow:

```
a = tf.constant([[1, 2],  
                 [3, 4]])  
b = tf.constant([[1, 1],  
                 [1, 1]])  
  
a = tf.Variable(a)  
b = tf.Variable(b)  
  
print(tf.add(a, b), "n")  
print(tf.multiply(a, b), "n")  
print(tf.matmul(a, b), "n")
```

En Pytorch:

```
a = torch.tensor([1, 2, 3], dtype=torch.float)  
b = torch.tensor([7, 8, 9], dtype=torch.float)  
  
print(torch.add(a, b))  
print(torch.subtract(b, a))  
print(a.mul(b))  
  
# Calculating the dot product  
print(a.dot(b))
```

Una ventaja de Pytorch frente a Tensorflow es el gran número de operaciones incluidas dentro de la clase que modela los tensores, lo que facilita la escritura y lectura de código. Mientras que operaciones como la media se implementan directamente dentro del paquete principal de tensorflow, en Pytorch son miembros de la clase:

En tensorflow:

```
A= tf.reduce_mean(a)
```

En Pytorch:

```
A= a.mean()
```

En cuanto a la definición de modelos de Machine Learning, ambas plataformas son similares. Un ejemplo de implementación de un perceptrón con dos capas se realizaría así en Pytorch:

```
# Constructing the model

class neural_network(nn.Module):
    def __init__(self, input_size, num_classes):
        super(neural_network, self).__init__()
        self.fc1 = nn.Linear(in_features=input_size, out_features=50)
        self.fc2 = nn.Linear(in_features=50, out_features=num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x
```

La eficiencia de TensorFlow vs Pytorch no resulta tampoco un condicionante a la hora de escoger una plataforma u otra. No obstante, a nivel práctico, sí que se observa que algunas operaciones sobre datasets son más rápidas en Pytorch y otras lo son más en Tensorflow. Sin embargo, Pytorch proporciona un mayor control al usuario sobre el flujo de control del grafo de cómputo, permitiendo escoger qué operaciones se realizan en CPU y cuáles en GPU (sin embargo, Tensorflow utiliza siempre por defecto GPU salvo que esta capacidad se desactive al comienzo del script).

En definitiva, la elección de una plataforma u otra depende en gran medida de los recursos y objetivos para los que se desarrollen los modelos de Deep Learning. Entre las ventajas de Pytorch podemos enumerar las siguientes:

- | Más intuitivo que Tensorflow para programadores de Python.
- | Existencia de buenos tutoriales y gran soporte de la comunidad.
- | Versiones estables y portables.
- | Grafos de cómputo dinámicos incluidos nativamente (Tensorflow sólo a partir de Tensorflow 2).

Sin embargo, los inconvenientes de la plataforma serían:

- | Falta de herramientas adicionales de desarrollo (similares a Google Colab o TensorBoard).
- | Escasos recursos orientados a sistemas en producción (servidores, dispositivos empotrados).

vuela

**PLATAFORMA
DIGITAL DE
ANDALUCÍA**

