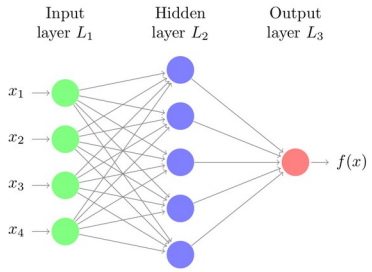
REDES NEURONALES:

Las redes neuronales son modelos creados al ordenar operaciones matemáticas siguiendo una determinada estructura. La forma más común de representar la estructura de una red neuronal es mediante el uso de capas (*layers*), formadas a su vez por neuronas (unidades, *units* o *neurons*). Cada neurona, realiza una operación sencilla y está conectada a las neuronas de la capa anterior y de la capa siguiente mediante pesos, cuya función es regular la información que se propaga de una neurona a otra.

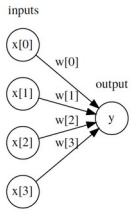


Representación de una red neuronal feed-forward (single-layer perceptron). Fuente: Computer Age Statistical Inference 2016

La primera capa de la red neuronal (color verde) se conoce como capa de entrada o *input layer* y recibe los datos en bruto, es decir, el valor de los predictores. La capa intermedia (color azul), conocida como capa oculta o *hidden layer*, recibe los valores la capa de entrada, ponderados por los pesos (flechas grises). La última capa, llamada *output layer*, combina los valores que salen de la capa intermedia para generar la predicción.

Para facilitar la comprensión de la estructura de las redes, es útil representar una red equivalente a un modelo de regresión lineal.

$$y=w_1x_1+...+w_dx_d+b$$



Representación de una red neuronal equivalente a un modelo lineal con 4 predictores. Fuente: COMS W4995 Applied Machine Learning

Cada neurona de la capa de entrada representa el valor de uno de los predictores. Las flechas representan los coeficientes de regresión, que en términos de redes se llaman pesos, y la neurona de salida representa el valor predicho. Para que esta representación equivalga a la ecuación de un modelo lineal, faltan dos cosas:

- El bias del modelo.
- Las operaciones de multiplicación y suma que combinan el valor de los predictores con los pesos del modelo.

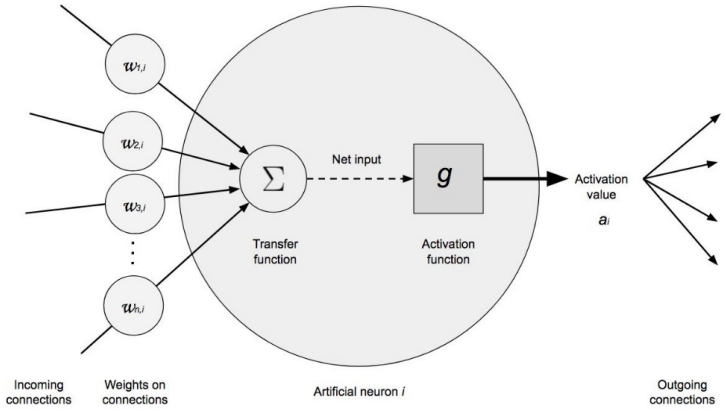
La capa intermedia de una red tiene un valor de bias, pero suele omitirse en las representaciones gráficas. En cuanto a las operaciones matemáticas, es el elemento clave que ocurre dentro de las neuronas y conviene verlo con detalle.

La neurona (unidad)

La neurona es la unidad funcional de los modelos de redes. Dentro de cada neurona, ocurren simplemente dos operaciones: la suma ponderada de sus entradas y la aplicación de una función de activación.

En la primera parte, se multiplica cada valor de entrada x_i por su peso asociado w_i y se suman junto con el bías. Este es el valor neto de entrada a la neurona. A continuación, este valor se pasa por una función, conocida como función de activación, que transforma el valor neto de entrada en un valor de salida.

Si bien el valor que llega a la neurona, multiplicación de los pesos por las entradas, siempre es una combinación lineal, gracias a la función de activación, se pueden generar salidas muy diversas. Es en la función de activación donde reside el potencial de los modelos de redes para aprender relaciones no lineales.



Representación de una neurona. Fuente: Deep Learning A Practitioner's Approach by Josh Patterson and Adam Gibson

El valor neto de entrada a una neurona es la suma de los valores que le llegan, ponderados por el peso de las conexiones, más el bias.

$$entrada = \sum_{i=1}^{n} x_i w_i + b$$

En lugar de utilizando el sumatorio, esta operación suele representarse como el producto matricial, donde $\mathbf{X}$ representa el vector de los valores de entrada y $\mathbf{W}$ el vector de pesos.

$$entrada = \boldsymbol{XW} + b$$

A este valor se le aplica una función de activación (g) que lo transforma en lo que se conoce como valor de activación (a), que es lo que finalmente sale de la neurona.

$$a = g(entrada) = g(\boldsymbol{XW} + b)$$

Para la capa de entrada, donde únicamente se quiere incorporar el valor de los predictores, la función de activación es la unidad, es decir, sale lo mismo que entra. En la capa de salida, la función de activación utilizada suele ser la identidad para problemas de regresión y *soft max* para clasificación.

Función de activación

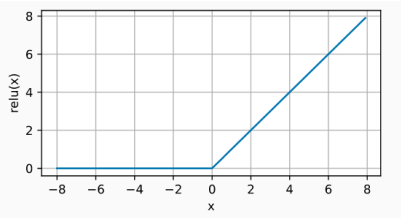
Las funciones de activación controlan en gran medida que información se propaga desde una capa a la siguiente (*forward propagation*). Estas funciones convierten el valor neto de entrada a la neuronal, combinación de los *input*, pesos y bías, en un nuevo valor. Es gracias combinar funciones de activación no lineales con múltiples capas, que los modelos de redes son capaces de aprender relaciones no lineales.

La gran mayoría de funciones de activación convierten el valor de entrada neto de la neurona en un valor dentro del rango (0, 1) o (-1, 1). Cuando el valor de activación de una neurona (salida de su función de activación) es cero, se dice que la neurona está inactiva, ya que no pasa ningún tipo de información a las siguientes neuronas. A continuación, se describen las funciones de activación más empleadas.

Rectified linear unit (ReLU)
La función de activación *ReLu* aplica una transformación no lineal muy simple, activa la neurona solo si el *input* está por encima de cero. Mientras el valor de entrada está por debajo de cero, el valor de salida es cero, pero cuando es superior de cero, el valor de salida aumenta de forma lineal con el de entrada.

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

De esta forma, la función de activación retiene únicamente los valores positivos y descarta los negativos dándoles una activación de cero.

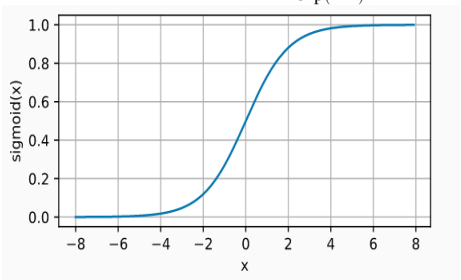


Representación función activación ReLU.

ReLU es con diferencia la función de activación más empleada, por sus buenos resultados en aplicaciones diversas. La razón de esto reside en el comportamiento de su derivada (gradiente), que es cero o constante, evitando así un problema conocido como vanishing gradients que limita la capacidad de aprendizaje de los modelos de redes.

Sigmoide
La función sigmoide transforma valores en el rango de (-inf, +inf) a valores en el rango (0, 1).

$$sigmoid(x) = \frac{1}{1+\exp(-x)}$$



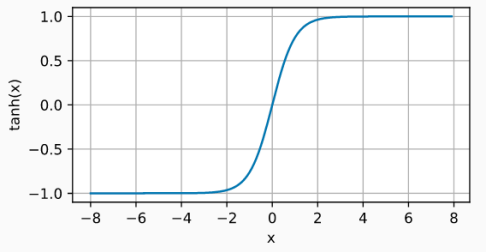
Representación función activación sigmoide.

Aunque la función de activación sigmoide se utilizó mucho en los inicios de los modelos de redes, en la actualidad, suele preferirse la función ReLU.

Un caso en el que la función de activación sigmoide sigue siendo la función utilizada por defecto es en las neuronas de la capa de salida de los modelos de clasificación binaria, ya que su salida puede interpretarse como probabilidades.

Tangente hiperbólica (Tanh)
La función de activación Tanh, se comporta de forma similar a la función sigmoide, pero su salida está acotada en el rango (-1, 1).

$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$$



Sin las funciones de activación, las redes neuronales solo pueden aprender relaciones lineales.

Función de coste (loss function)

La función de coste (l), también llamada función de pérdida, *loss function* o *cost function*, es la encargada de cuantificar la distancia entre el valor real y el valor predicho por la red, en otras palabras, mide cuánto se equivoca la red al realizar predicciones. En la mayoría de casos, la función de coste devuelve valores positivos. Cuanto más próximo a cero es el valor de coste, mejor son las predicciones de la red (menor error), siendo cero cuando las predicciones se corresponden exactamente con el valor real.

La función de coste puede calcularse para una única observación o para un conjunto de datos (normalmente promediando el valor de todas las observaciones). El segundo caso, es el que se utiliza para dirigir el entrenamiento de los modelos.

Dependiendo del tipo de problema, regresión o clasificación, es necesario utilizar una función de coste u otra. En problemas de regresión, las más utilizadas son el error cuadrático medio y el error absoluto medio. En problemas de clasificación suele emplearse la función *log loss*, también llamada *logistic loss* o *cross-entropy loss*.

Error cuadrático medio

El error cuadrático medio (*mean squared error*, *MSE*) es con diferencia la función de coste más utilizada en problemas de regresión. Para una determinada observación i , el error cuadrático se calcula como la diferencia al cuadrado entre el valor predicho $\hat{y}$ y el valor real y .

$$l^{(i)}(\boldsymbol{w},b)=\left(\hat{y}^{(i)}-y^{(i)}\right)^2$$

Las funciones de coste suelen escribirse con la notación $l(\boldsymbol{w},b)$ para hacer referencia a que su valor depende de los pesos y bias del modelo, ya que son estos los que determinan el valor de las predicciones $y^{(i)}$.

Con frecuencia, esta función de coste se encuentra multiplicada por ½ , esto es simplemente por conveniencia matemática para simplificar el cálculo de su derivada.

$$l^{(i)}(\boldsymbol{w},b)=\frac{1}{2}\left(\hat{y}^{(i)}-y^{(i)}\right)^2$$

Para cuantificar el error que comete el modelo todo un conjunto de datos, por ejemplo los de entrenamiento, simplemente se promedia el error de todas las N observaciones.

$$L(\boldsymbol{w},b)=\frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\boldsymbol{w},b)=\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}^{(i)}-y^{(i)}\right)^2$$

Cuando un modelo se entrena utilizando el error cuadrático medio como función de coste, está aprendiendo a predecir la media de la variable respuesta.

Error medio absoluto

El error medio absoluto (*mean absolute error*, *MAE*) consiste en promediar el error absoluto de las predicciones.

$$L(\boldsymbol{w},b)=\frac{1}{n}\sum_{i=1}^{n}\left|\hat{y}^{(i)}-y^{(i)}\right|$$

El error medio absoluto es más robusto frente a *outliers* que el error cuadrático medio. Esto significa que el entrenamiento del modelo se ve menos influenciado por datos anómalos que pueda haber en el conjunto de entrenamiento. Cuando un modelo se entrena utilizando el error absoluto medio como función de coste, está aprendiendo a predecir la mediana de la variable respuesta.

Log loss, logistic loss o *cross-entropy loss*

En problemas de clasificación, la capa de salida utiliza como función de activación la función *softmax*. Gracias a esta función, la red devuelve una serie de valores que pueden interpretarse como la probabilidad de que la observación predicha pertenezca a cada una de las posibles clases.

Cuando la clasificación es de tipo binaria, donde la variable respuesta es 1 o 0, y p=Pr(y=1) , la función de coste *log-likelihood* se define como:

$$L_{\log}(y,p) = -\log Pr(y|p) = -(y\log(p) + (1-y)\log(1-p))$$

Para problemas de clasificación con más de dos clases, esta fórmula se generaliza a:

$$L_{\log}(Y,P) = -\log Pr(Y|P) = \frac{-1}{N}\sum_{i=0}^{N-1}\sum_{k=0}^{K-1} y_{i,k}\log p_{i,k}$$

En ambos casos, minimizar esta la función equivale a que la probabilidad predicha para la clase correcta tienda a 1, y a 0 en las demás clases.

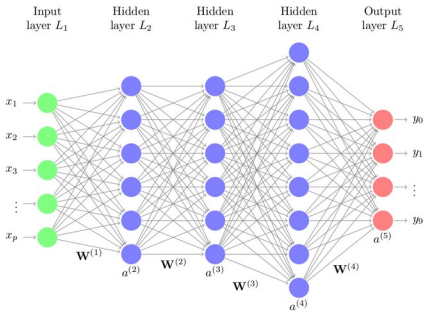
Dado que esta función se ha utilizado en campos diversos, se le conoce por nombres distintos: *Log loss*, *logistic loss* o *cross-entropy loss*, pero todos hacen referencia a lo mismo. Puede encontrarse una explicación más detallada de esta función de coste aquí.

Múltiples capas

El modelo de red neuronal con una única capa (*single-layer perceptron*), aunque supuso un gran avance en el campo del *machine learning*, solo es capaz de aprender patrones sencillos. Para superar esta limitación, los investigadores descubrieron que, combinando múltiples capas ocultas, la red puede aprender relaciones mucho más complejas entre los predictores y la variable respuesta. A esta estructura se le conoce como perceptrón multicapa o *multilayer perceptron* (*MLP*), y puede considerarse como el primer modelo de deep learning.

La estructura de un perceptón multicapa consta de varias capas de neuronas ocultas. Cada neurona está conectada a todas las neuronas de la capa anterior y a las de la capa posterior. Aunque no es estrictamente necesario, todas las neuronas que forman parte de una misma capa suelen emplear la misma función de activación.

Combinando múltiples capas ocultas y funciones de activación no lineales, los modelos de redes pueden aprender prácticamente cualquier patrón. De hecho, está demostrado que, con suficientes neuronas, un MLP es un aproximador universal para cualquier función.



Representación de una red neuronal feed-forward (multi-layer perceptron). Fuente: Computer Age Statistical Inference 2016

Entrenamiento

El proceso de entrenamiento de una red neuronal consiste en ajustar el valor de los pesos y bías de tal forma que, las predicciones que se generen, tengan el menor error posible. Gracias a esto, el modelo es capaz de identificar qué predictores tienen mayor influencia y de qué forma están relacionados entre ellos y con la variable respuesta.

La idea intuitiva de cómo entrenar una red neuronal es la siguiente:

- Iniciar la red con valores aleatorios de los pesos y bias.
- Para cada observación de entrenamiento ($\boldsymbol{X}$, y), calcular el error que comete la red al hacer sus predicciones. Promediar los errores de todas las observaciones.
- Identificar la responsabilidad que ha tenido cada peso y bias en el error de la predicción.

- Modificar ligeramente los pesos y bias de la red (de forma proporcional a su responsabilidad en el error) en la dirección correcta para que se reduzca el error.
- Repetir los pasos 2, 3, 4 y 5 hasta que la red sea suficientemente buena.

Si bien la idea parece sencilla, alcanzar una forma de implementarla ha requerido la combinación de múltiples métodos matemáticos, en concreto, el algoritmo de retropropagación (*backpropagation*) y la optimización por descenso de gradiente (*gradient descent*).

Backpropagation

Backpropagation es el algoritmo que permite cuantificar la influencia que tiene cada peso y bias de la red en sus predicciones. Para conseguirlo, hace uso de la regla de la cadena (*chain rule*) para calcular el gradiente, que no es más que es el vector formado por las derivadas parciales de una función.

En el caso de las redes, la derivada parcial del error respecto a un parámetro (peso o bias) mide cuanta "responsabilidad" ha tenido ese parámetro en el error cometido. Gracias a esto, se puede identificar qué pesos de la red hay que modificar para mejorarla. El siguiente paso necesario, es determinar cuánto y cómo modificarlos (optimización).

Descenso de gradiente

Descenso de gradiente (*gradient descent*) es un algoritmo de optimización que permite minimizar una función haciendo actualizaciones de sus parámetros en la dirección del valor negativo de su gradiente. Aplicado a las redes neuronales, el descenso de gradiente permite ir actualizando los pesos y bías del modelo para reducir su error.

Dado que, calcular el error del modelo para todas las observaciones de entrenamiento, en cada iteración, puede ser computacionalmente muy costoso, existe una alternativa al método de descenso de gradiente llamada gradiente estocástico (*stochastic gradient descent*, *SGD*). Este método consiste en dividir el conjunto de entrenamiento en lotes (*minibatch* o *batch*) y actualizar los parámetros de la red con cada uno. De esta forma, en lugar de esperar a evaluar todas las observaciones para actualizar los parámetros, se pueden ir actualizando de forma progresiva. Una ronda completa de iteraciones sobre todos los *batch* se llama época. El número de épocas con las que se entrena una red equivale al número de veces que la red ve cada ejemplo de entrenamiento.

Preprocesado

A la hora de entrenar modelos basados en redes neuronales, es necesario realizar, al menos, dos tipos de transformaciones de los datos.

Binarización (*One hot ecoding*) de las variables categóricas

La binarización (*one-hot-encoding*) consiste en crear nuevas variables *dummy* con cada uno de los niveles de las variables cualitativas. Por ejemplo, una variable llamada color que contenga los niveles rojo, verde y azul, se convertirá en tres nuevas variables (color_rojo, color_verde, color_azul), todas con el valor 0 excepto la que coincide con la observación, que toma el valor 1.

Estandarización y escalado de variables numéricas

Cuando los predictores son numéricos, la escala en la que se miden, así como la magnitud de su varianza pueden influir en gran medida en el modelo. Si no se igualan de alguna forma los predictores, aquellos que se midan en una escala mayor o que tengan más varianza dominarán el modelo aunque no sean los que más relación tienen con la variable respuesta. Existen principalmente 2 estrategias para evitarlo:

- Centrado: consiste en restarle a cada valor la media del predictor al que pertenece. Si los datos están almacenados en un dataframe, el centrado se consigue restándole a cada valor la media de la columna en la que se encuentra. Como resultado de esta transformación, todos

los predictores pasan a tener una media de cero, es decir, los valores se centran en torno al origen.

- Normalización (estandarización): consiste en transformar los datos de forma que todos los predictores estén aproximadamente en la misma escala.
  - Normalización Z-score (StandardScaler): dividir cada predictor entre su desviación típica después de haber sido centrado, de esta forma, los datos pasan a tener una distribución normal.
  - Estandarización max-min (MinMaxScaler): transformar los datos de forma que estén dentro del rango [0, 1].

Nunca se deben estandarizar las variables después de ser binarizadas.

Hiperparámetros

La gran "flexibilidad" que tienen las redes neuronales es un arma de doble filo. Por un lado, son capaces de generar modelos que aprenden relaciones muy complejas, sin embargo, sufren fácilmente el problema de sobreajuste (*overfitting*) lo que los incapacita al tratar de predecir nuevas observaciones. La forma de minimizar este problema y conseguir modelos útiles, pasa por configurar de forma adecuada sus hiperparámetros. Algunos de los más importantes son:

Número y tamaño de capas

La arquitectura de una red, el número de capas y el número de neuronas que forman parte de cada capa, determinan en gran medida la complejidad del modelo y con ello su potencial capacidad de aprendizaje.

La capa de entrada y salida son sencillas de establecer. La capa de entrada tiene tantas neuronas como predictores y la capa de salida tiene una neurona en problemas de regresión y tantas como clases en problemas de clasificación. En la mayoría de implementaciones, estos valores se establecen automáticamente en función del conjunto de entrenamiento. El usuario suele especificar únicamente el número de capas intermedias (ocultas) y el tamaño de las mismas.

Cuantas más neuronas y capas, mayor la complejidad de las relaciones que puede aprender el modelo. Sin embargo, dado que en cada neurona está conectada por pesos al resto de neuronas de las capas adyacentes, el número de parámetros a aprender aumenta y con ello el tiempo de entrenamiento.

Learning rate

El *learning rate* o ratio de aprendizaje establece cómo de rápido pueden cambiar los parámetros de un modelo a medida que se optimiza (aprende). Este hiperparámetro es uno de los más complicados de establecer, ya que depende mucho de los datos e interacciona con el resto de hiperparámetros. Si el *learning rate* es muy grande, el proceso de optimización puede ir saltando de una región a otra sin que el modelo sea capaz de aprender. Si por el contrario, el *learning rate* es muy pequeño, el proceso de entrenamiento puede tardar demasiado y no llegar a completarse. Algunas de las recomendaciones heurísticas basadas en prueba y error son:

- Utilizar un *learning rate* lo más pequeño posible siempre y cuando el tiempo de entrenamiento no supere las limitaciones temporales disponibles.
- No utilizar un valor constante de *learning rate* durante todo el proceso de entrenamiento. Por lo general, utilizar valores mayores al inicio y pequeños al final.

Algoritmo de optimización

El descenso de gradiente y el descenso de gradiente estocástico fueron de los primeros métodos de optimización utilizados para entrenar modelos de redes neuronales. Ambos utilizan directamente el gradiente para dirigir la optimización. Pronto se vio que esto genera problemas a medida que las

redes aumentan de tamaño (neuronas y capas). En muchas regiones del espacio de búsqueda, el gradiente es muy proximo a cero, lo que hace que la optimización quede estancada en estas regiones. Para evitar este problema, se han desarrollado modificaciones del descenso de gradiente capaces de adaptar el *learning rate* en función del gradiente y subgradiente. De esta forma, el proceso de aprendizaje se ralentiza o acelera dependiendo de las características de la región del espacio de búsqueda en el que se encuentren. Aunque existen multitud de adaptaciones, suele recomendarse:

- Para conjuntos de datos pequeños: l-bfgs
- Para conjuntos de datos grandes: adam o rmsprop

La elección del algoritmo de optimización puede tener un impacto muy grande en el aprendizaje de los modelos, sobretodo en *deep learning*. Puede encontrarse una excelente descripción más detallada en el libro gratuito Dive into Deep Learning.

Regularización

Los métodos de regularización se utilizan con el objetivo de reducir el sobreajuste (*overfitting*) de los modelos. Un modelo con sobreajuste memoriza los datos de entrenamiento pero es incapaz de predecir correctamente nuevas observaciones.

Los modelos de redes neuronales pueden considerarse como modelos sobre parametrizados, por lo tanto, las estrategias de regularización son fundamentales. De entre las muchas que existen, destacan la regularización L1 y L2 (*weight decay*), y el *dropout*.

Penalización L1 y L2

El objetivo de la penalización L1 y L2, esta última también conocida como *weight decay*, es evitar que los pesos tomen valores excesivamente elevados. De esta forma se evita que unas pocas neuronas dominen el comportamiento de la red y se fuerza a que las características poco informativas (ruido) tengan pesos próximos o iguales a cero.

- Regularización L1: La complejidad se mide como la media del valor absoluto de los pesos del modelo. Esta técnica es de utilidad cuando sospechamos que algunas de las características de entrada son irrelevantes. Se favorece que los coeficientes acaben valiendo 0, permitiendo descubrir las variables de entrada relevantes (selección de características). Funciona mejor cuando las variables no están muy correlacionadas entre sí.
- Regularización L2: La complejidad se mide como la media del cuadrado de los pesos del modelo. Esta técnica es de utilidad cuando sospechamos que algunas de las variables de entrada están correlacionadas entre sí. Hace que los coeficientes acaben siendo pequeños, minimizando la correlación entre las variables.

Dropout

El proceso consiste en de desactivar aleatoriamente una serie de neuronas durante el proceso de entrenamiento. En concreto, durante cada iteración del entrenamiento, se ponen a cero los pesos de una fracción aleatoria de neuronas por capa. El método de *dropout*, descrito por Srivastava et al. en 2014, se ha convertido en un estándar para entrenar redes neuronales. El porcentaje de neuronas que suele desactivarse por capa (*dropout rate*) suele ser un valor entre 0.2 y 0.5.

Earyly stopping

Detener el entrenamiento antes de que el modelo empiece a sobreajustar. (Se aplica con un callback. Función que se ejecuta cuando termina una época)

Bibliografía

The Elements of Statistical Learning (2nd edition)
Hastie, Tibshirani and Friedman (2009). Springer-Verlag.
Applied Predictive Modeling by Max Kuhn and Kjell Johnson
Deep Learning by Josh Patterson, Adam Gibson
Python Machine Learning 3rd Edition by Sebastian Raschka
Hands-On Computer Vision with TensorFlow 2 by Benjamin Planche and Eliot Andres
sklearn neural network

Documento de donde se saca la información: