GUIA PARA CREAR Y ENTRENAR UNA RED NEURONAL ARTIFICIAL SENCILLA

Índice

Resumen

Se realiza una descripción del proceso de construcción y entrenamiento de una Red Neuronal de Aprendizaje Profundo (Deep Learning). Para llevar a cabo este procedimiento se utiliza lenguaje python, el cual nos permite utilizar librerías desarrolladas específicamente para la creación y entrenamiento de redes neuronales. El conjunto de datos seleccionado es el conocido "mnist", utilizado a menudo como conjunto base para ejemplos de redes neuronales orientadas al aprendizaje profundo.

Introducción

El Aprendizaje Automático es una rama de la Inteligencia Artificial, el cual tiene como objetivo desarrollar técnicas que permitan que las computadoras aprendan. Dentro de este conjunto de técnicas se destacan aquellas que poseen una gran cantidad de transformaciones aplicadas a la señal mientras se propaga desde la capa de entrada a la capa de salida. A estas últimas se las denomina Aprendizaje Profundo.

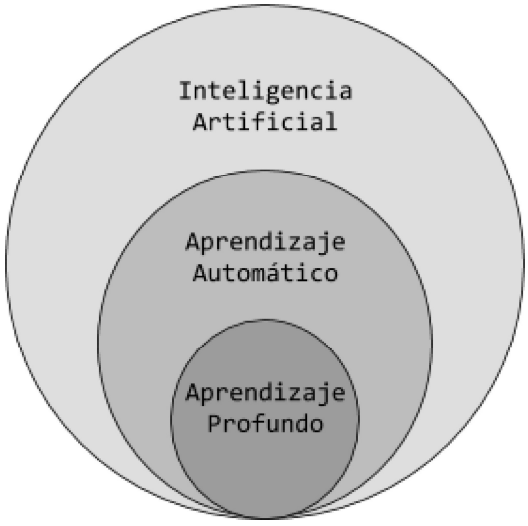


***Fig.* 1**. Aprendizaje Profundo dentro de la Inteligencia Artificial.

El aprendizaje profundo está presente en cosas cómo, las traducciones, los asistentes virtuales, los chatbots, reconocimiento facial, autos sin conductor, compras y entretenimiento personalizado, etc.

Hoy en día se trabaja exhaustivamente en esta temática, y se están logrando avances a pasos agigantados, tal es así, que la idea de incorporar robots a la sociedad no parece tan descabellada.

Las características que destacan a las Redes Neuronales son: el procesamiento de la información proveniente del entorno en tiempo real, la robustez y tolerancia a fallas. la capacidad de adaptación, el manejo de información difusa, con ruido e inconsistente y el procesamiento paralelo.

1. Herramientas

El trabajo es realizado sobre lenguaje **Python**, ya que es uno de los lenguajes del momento utilizado para temas del aprendizaje automático, que además de ser ágil y sencillo con una curva de aprendizaje muy corta, también se destaca por contar con una gran comunidad que lo respalda.

Las librerías que se utilizan son:

- **TensorFlow**: es una biblioteca de código abierto para aprendizaje automático desarrollada por google.
- **Keras**: es una biblioteca de Redes Neuronales de Código Abierto escrita en Python. Es capaz de ejecutarse sobre TensorFlow. Está especialmente diseñada para posibilitar la experimentación en más o menos poco tiempo con redes de Aprendizaje Profundo
- **Numpy**: es una extensión de Python, que le agrega mayor soporte para vectores y matrices, constituyendo una biblioteca de funciones matemáticas de alto nivel para operar con esos vectores o matrices.
- **Pandas**: es una biblioteca de software escrita como extensión de NumPy para manipulación y análisis de datos para el lenguaje de programación Python. En particular, ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales.
- **Matplotlib**: es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

Cómo workspace, se utiliza **Google Collaboratory**, o "Colab" para abreviar, un producto de Google Research. Colab permite que todos puedan escribir y ejecutar código arbitrario de Python en el navegador. Ideal para aplicarlo en proyectos de aprendizaje automático, análisis de datos y educación. Más técnicamente, Colab es un servicio de notebook alojado de Jupyter que no requiere configuración para usarlo y brinda acceso gratuito a recursos computacionales, incluidas GPU.

2. Preprocesamiento

En primer lugar se debe contar con un conjunto de datos, el cual se utiliza para entrenar y testear la red neuronal que deseamos construir. En este caso se toma el conjunto “mnist”. El MNIST es un conjunto de datos desarrollado por Yann LeCun, Corinna Cortes y Christopher Burges para la evaluación de modelos de aprendizaje de máquinas sobre el problema de la clasificación de los dígitos escritos a mano. Este conjunto se

encuentra dentro de la biblioteca de Keras por lo cual podremos importarlo directamente:

```
import keras

from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Salida:

```
Using TensorFlow backend.
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
```

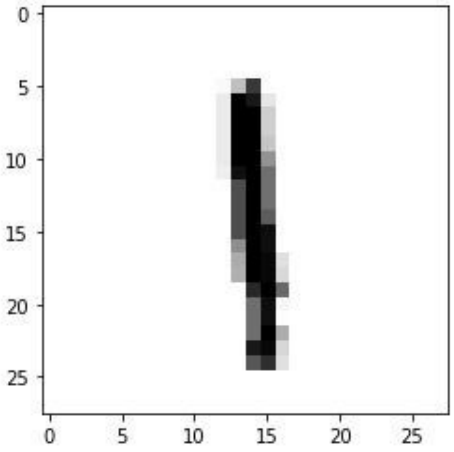
Cómo se aprecia en el código, verificar que la red neuronal predice de forma correcta el conjunto se divide en dos partes, cargando una parte en una matriz y la otra en la otra matriz. De esta manera, una de ellas se utiliza para entrenar la red neuronal y la otra para testear su comportamiento.

Para verificar los datos del conjunto se utiliza la librería “matplotlib” que permite graficar la imagen contenida de la siguiente forma:

```
import matplotlib.pyplot as plt
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

Salida:

```
<matplotlib.image.AxesImage at 0x7f4763581978>
```



Cada imagen viene acompañada por una etiqueta la cual expresa el valor representado en la imagen, en otras palabras, es la salida esperada. Este tipo de conjuntos que poseen una etiqueta, se utilizan para el "aprendizaje supervisado", una técnica para deducir una función a partir de datos de entrenamiento. Los datos de entrenamiento consisten de pares de objetos (normalmente vectores): una componente del par son los datos de entrada y el otro, los resultados deseados (para el caso **x** e **y**).

```
print(y_train[8])
```

Salida:

```
1
```

Para conocer el tipo de dato de la imagen es uint (entero sin signo), por lo cual, los pixeles de la matriz están representados entre 0 y 255 de acuerdo a la intensidad de cada pixel, a menor valor, menor intensidad viceversa caso contrario:

```
print(x_train.dtype)
```

Salida:

```
uint8
```

Si se quiere representar los pixeles entre 0 y 1 es necesario convertir el tipo de dato a flotante y dividirlo por 255:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

Luego para facilitar la entrada de datos a nuestra red neuronal debemos hacer una transformación del tensor (imagen) de 2 dimensiones (2D) a un vector de una dimensión (1D). Es decir, la matriz de 28×28 números se puede representar con un vector (array) de

784 números (concatenando fila a fila), que es el formato que acepta como entrada una red neuronal densamente conectada.

```
print(x_train.ndim)
print(x_train.shape)
```

Salida:

```
3
(60000, 28, 28)
```

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

print(x_train.shape)
print(x_test.shape)
```

Salida:

```
(60000, 784)
(10000, 784)
```

Como se sabe el resultado esperado oscila entre 0 y 9, osea 10 valores distintos:

```
print(y_train[0])
print(y_test[0])

print(y_train.shape)
print(y_test.shape)
```

Salida:

```
5
7
(60000,)
(10000,)
```

Para que la red pueda clasificar y estimar una salida es necesario, convertir estos valores en formato binario, como se aprecia a continuación:

```
from keras.utils import to_categorical
```

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

print(y_train[0])
print(y_test[0])
print(y_train.shape)
print(y_test.shape)
```

Salida:

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
(60000, 10)
(10000, 10)
```

De esta forma, el valor se ve representado por la posición en que se encuentra el 1 en un arreglo de 10 elementos.

3. Procesamiento

Una vez realizado el preprocesamiento de los datos, pasamos a la parte de la construcción de la red neuronal. Para esto se utiliza la librería **Keras**, que viene incorporada en el paquete **Tensorflow**.

Cómo primer paso se crea una red neuronal del tipo **Perceptrón Multicapa** y un **modelo secuencial**, donde la predicción realizada por una capa para un intervalo de tiempo actúa a su vez como variable explicativa o de entrada en otra capa dedicada a la predicción de otro intervalo anterior o posterior. Para terminar de entender este concepto, primero se debe entender la estructura de una red neuronal Perceptrón Multicapa.

Este tipo de redes neuronales, tienen una capa de entrada (input layer), una o más capas compuestas por perceptrones, llamadas capas ocultas (hidden layers) y una capa final con varios perceptrones llamada la capa de salida (output layer). En general se refiere a Deep Learning cuando el modelo basado en redes neuronales está compuesto por múltiples capas ocultas. Visualmente se puede presentar con el siguiente esquema:

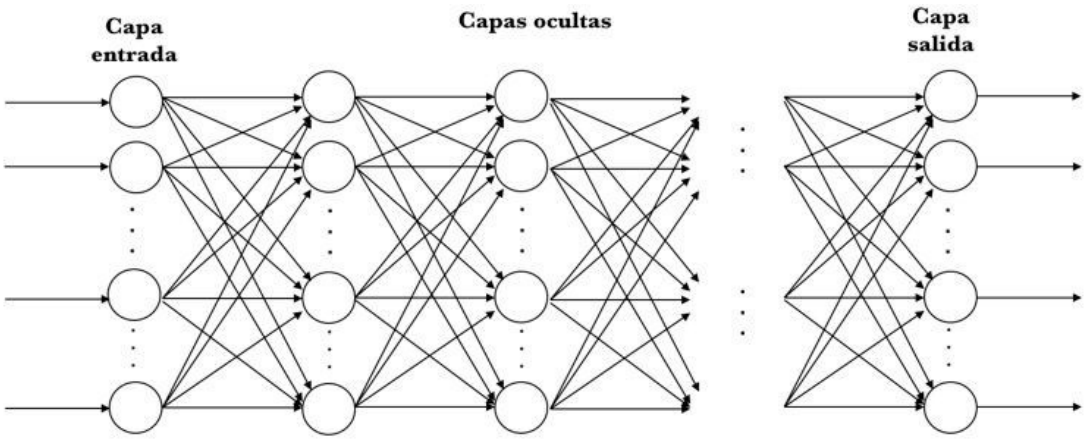


***Fig.* 2**. Red Neuronal Perceptrón Multicapa.

Hasta ahora se explicó cómo es la estructura de la red perceptrón multicapa, pero qué es un **perceptrón** ?. "El Perceptrón es la versión más simple de red neuronal porque consta de una sola capa que contiene una sola neurona artificial".

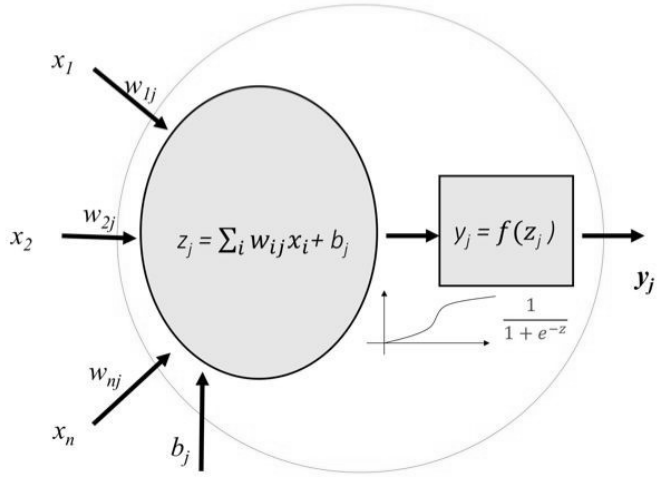


***Fig.* 3**. Perceptrón.

Pero entonces, ¿qué es una **neurona artificial** ? Bueno, las Redes Neuronales Artificiales o simplemente Redes Neuronales, buscan emular el comportamiento del cerebro humano. El cerebro humano procesa información imprecisa rápidamente, aprende sin instrucciones explícitas, crea representaciones internas que permiten estas habilidades. Entonces, si consideramos que las redes neuronales artificiales intentan

emular el comportamiento del cerebro humano, una neurona artificial, intenta emular el comportamiento de una neurona biológica.
Gráficamente se puede representar la comparación entre una neurona biológica y una neurona artificial de la siguiente manera (Fig. 4):

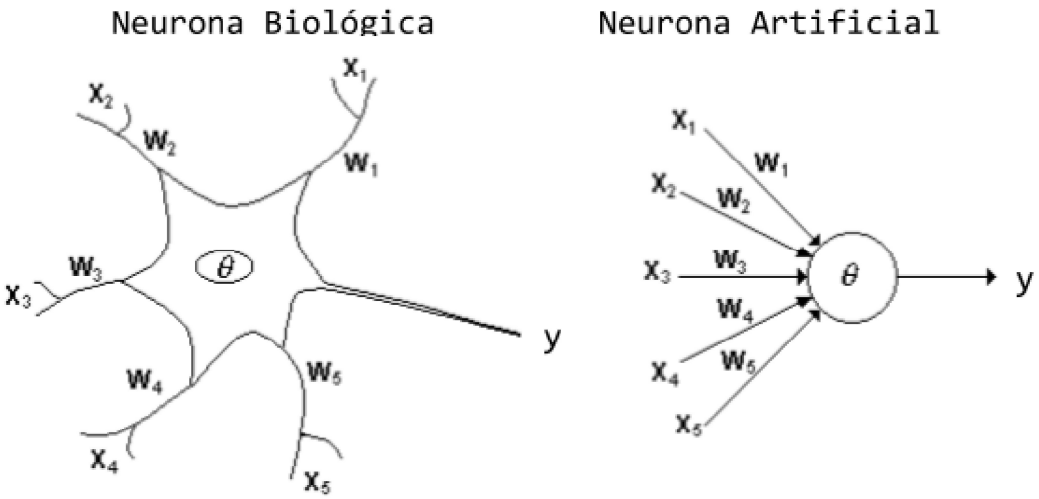


Fig. 4. Similitud entre una neurona biológica y una neurona artificial.

En el caso de la neurona biológica, las variables que la representan son las siguientes:

- Las entradas **Xi** representan las señales que provienen de otras neuronas y que son capturadas por las dendritas.
- Los pesos **Wi** son la intensidad de la sinapsis que conecta dos neuronas; tanto Xi como Wi son valores reales.
- ***θ*** es el umbral que la neurona debe sobrepasar para activarse; este proceso ocurre biológicamente en el cuerpo de la célula.

Las variables mencionadas también se encuentran presentes en las neuronas artificiales, pero hay que tener en cuenta algunas diferencias. A continuación se explican la composición y puntos más importantes de una neurona artificial:

- Las señales de entrada a una neurona artificial X1 , X2 ,.., Xn son variables continuas en lugar de pulsos discretos, como se presentan en una neurona biológica.

- Cada señal de entrada pasa a través de una ganancia o peso (W1, W2, ..., Wn), llamado peso sináptico o fortaleza de la conexión cuya función es análoga a la de la función sináptica de la neurona biológica. Los pesos pueden ser positivos (excitatorios), o negativos (inhibitorios).
- El nodo sumatorio acumula todas las señales de entrada multiplicadas o ponderadas por los pesos y las pasa a la salida a través de una función umbral o función de transferencia (algunas de las cuales pueden incluir $\boldsymbol{\theta}$ explícitamente y otras implícitamente) (Ver Apéndice).
    - Cada entrada es multiplicada por el peso de arco correspondiente.
    - Cada neurona calcula su entrada neta como: $neta_j = \sum_{i=1}^{n} x_i * w_i = X * W$
    - El valor de salida (único) se obtiene como: $y = f(neta)$

3.1 Forward Propagation

Es la manera en la cual las redes neuronales crean las predicciones. En un principio la red neuronal tiene valores de W y b aleatorios en cada neurona, los datos de entrenamiento pasan por estas neuronas hasta llegar a la capa de salida, en esta capa la red neuronal predice la clase a la cual pertenecen los datos de entrenamiento, estas predicciones las usa la función de pérdida para medir que tan buena es la red neuronal, este ciclo se repite varias veces según indiquemos y en cada ciclo se ejecuta el algoritmo de backpropagation (ver Apéndice) para actualizar los valores de W y b.

3.2 Creación y compilación de la red

La API de Keras tiene dos modos de construir redes neuronales. El más simple es el modelo secuencial que solo permite que las capas se agreguen en secuencia. (clase "Sequential")

```
from tensorflow.python.keras import Sequential
model = Sequential()
```

En este caso, el modelo en Keras se considera como una secuencia de capas donde cada una de ellas retorna los datos de entrada de la siguiente hasta obtener la salida deseada. En Keras podemos encontrar todos los tipos de capas requeridas y se pueden agregar fácilmente al modelo mediante el método add( ).

```
from tensorflow.python.keras.layers import Dense, Activation

model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))

model.summary()
```

Salida:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 10)                7850
_________________________________________________________________
dense_1 (Dense)              (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

En la primera capa por cada neurona i (entre 0 y 9) requerimos 784 parámetros para los pesos w_{ij} y por tanto 10 x 784 parámetros para almacenar los pesos de las 10 neuronas. Además de los 10 parámetros adicionales para los 10 sesgos b_j correspondientes a cada una de ellas. En la segunda capa, al ser una función softmax, se requiere conectar todos sus 10 nodos con los 10 nodos de la capa anterior, y por tanto se requieren 10×10 parámetros w_i además de los correspondientes 10 sesgos b_j correspondientes a cada nodo.

La red neuronal ahora se ha definido y debe completarse agregando una función de coste, un optimizador y las métricas de rendimiento. Esto se llama modelo de "compilación" en Keras, y su función es configurar el modelo para el entrenamiento. Los parámetros principales son:

- Función de pérdida (“loss”): La función de pérdida, también conocida como función de costo, es la función que nos dice que tan buena es la red neuronal, un resultado alto indica que la red neuronal tiene un desempeño pobre y un

resultado bajo indica que la red neuronal está haciendo un buen trabajo. (Ver Apéndice)

- Optimizador ("optimizer"): es el algoritmo que permite optimizar la función de pérdida, haciendo que dicha función, durante el proceso de entrenamiento, vaya mejorando sus resultados. (ver Apéndice)
- Métrica de rendimiento ("metrics"): para comparar el rendimiento del sistema actual.

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

Para un problema de clasificación como el del ejemplo (MNIST) que tiene 10 clases posibles, necesitamos usar la función de pérdida llamada "categorical_crossentropy" (ver Apéndice). La métrica de rendimiento que nos interesa es la precisión de la clasificación ("accuracy").

3.2 Entrenamiento de la red

El entrenamiento de la red consiste en aplicar el conjunto de entrenamiento seleccionado (x_train), para luego ser comparado con sus correspondientes salidas (y_train). En dicho proceso es necesario corregir los pesos asociados a cada neurona para que la predicción sea lo más precisa posible. Esta corrección se aplica de forma regresiva, es decir, desde la capa de salida hacia las capas ocultas hasta llegar a la capa de entrada. Además, para mejorar los resultados es preciso realizar varias iteraciones de este proceso, es por ello que se establece un Número de épocas para entrenar el modelo. Una época es una iteración sobre la totalidad del conjunto y los datos de salida proporcionados. (ver Apéndice)

```
model.fit(x_train, y_train, epochs=5) #entrenamiento
```

Salida:

```
Epoch 1/5
1875/1875 [==============================] - 2s 1ms/step - loss: 1.9403 - accuracy:
0.4803
Epoch 2/5
1875/1875 [==============================] - 2s 944us/step - loss: 1.3665 - accuracy:
```

```
0.6743
Epoch 3/5
1875/1875 [==============================] - 2s 952us/step - loss: 1.0206 - accuracy:
0.7696
Epoch 4/5
1875/1875 [==============================] - 2s 945us/step - loss: 0.8168 - accuracy:
0.8200
Epoch 5/5
1875/1875 [==============================] - 2s 939us/step - loss: 0.6929 - accuracy:
0.8447
<tensorflow.python.keras.callbacks.History at 0x7f9526d52b38>
```

En el entrenamiento de la red se ve claramente como la performance aumenta en cada etapa.

3.3 Testeo de la red

Ahora se quiere evaluar la performance de la red, para esto se utilizan el resto de los datos que no se utilizaron para el entrenamiento.

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

Salida:

```
313/313 [==============================] - 0s 904us/step - loss:
0.2867 - accuracy: 0.9202
```

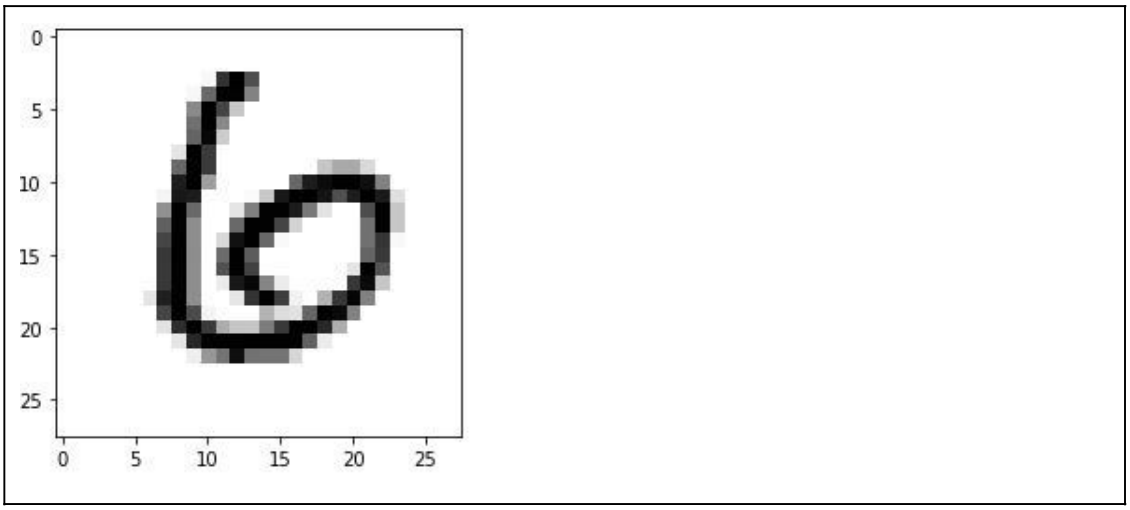
Ahora se puede observar que la performance sobre este conjunto es de un 92%, por lo cual, a priori se podría considerar que la red se comporta de buena forma sobre conjuntos desconocidos.

Se puede tomar un ejemplo para demostrar que efectivamente la red proporciona como resultado la salida esperada:

```
x_test = x_test.reshape(10000, 28,28) //
plt.imshow(x_test[11], cmap=plt.cm.binary)
```

Salida:

```
<matplotlib.image.AxesImage at 0x7fed2d9d4198>
```



```
x_test = x_test.reshape(10000, 784)
predictions = model.predict(x_test)
```

Una vez calculado el vector resultado de la predicción para este conjunto de datos podemos saber a qué clase le da más probabilidad de pertenencia mediante la función argmax de Numpy, que retorna el índice de la posición que contiene el valor más alto del vector. En concreto, para el elemento 11:

```
import numpy as np
np.argmax(predictions[11])
```

Salida:

```
6
```

```
print(predictions[11])
print(predictions[11].sum())
```

Salida:

```
[0.01538657 0.07919672 0.18933249 0.00409848 0.00313633 0.0013336
 0.5686336  0.00248949 0.11423277 0.02216003]
1.0000001
```

```
y_true=np.argmax(y_test,axis=-1)
y_pred=np.argmax(model.predict(x_test), axis=-1)
```

```
print(y_true)
print(y_pred)
```

Salida:

```
[7 2 1 ... 4 5 6]
[7 2 1 ... 4 5 6]
```

Conclusión

Cómo conclusión se puede decir que se obtuvo un resultado satisfactorio en todo el proceso realizado y se pudo demostrar con un ejemplo la utilidad que proporciona el uso de las redes neuronales orientadas al aprendizaje automático profundo. Este simple ejemplo es una muestra, y sirve como punto de partida para ideas originales por parte de los alumnos y/o profesores de Facultad de Informática e Ingeniería con el fin de encarar futuros proyectos.

Bibliografía

- Historia del Deep Learning (3): Hitos. Alejandro Arranz. 14 enero 2019. Link: https://www.datahack.es/historia-del-deep-learning-3/
- Trabajo Final: Redes Neuronales Artificiales. Ariel E. Repetur. Tandil, Abril de 2019. Link: https://www.ridaa.unicen.edu.ar/xmlui/bitstream/handle/123456789/2058/Trabajo%20Final%20Repetur.pdf?sequence=1&isAllowed=y
- Conceptos básicos sobre redes neuronales. Vicente Rodriguez. 30 de Octubre de 2018. Link: https://vincentblog.xyz/posts/conceptos-basicos-sobre-redes-neuronales#:~:text=La%20funci%C3%B3n%20de%20perdida%2C%20tambi%C3%A9n,esta%20haciendo%20un%20buen%20trabajo.
- Deep Learning Introducción a la práctica con Keras. Jordi Torres. Link: https://torres.ai/deep-learning-inteligencia-artificial-keras/
- Función de Activación. Wikipedia. 25 ene 2020. Link: https://es.wikipedia.org/wiki/Funci%C3%B3n_de_activaci%C3%B3n
- Función de pérdida. Wikipedia. 16 feb 2020. Link: https://es.wikipedia.org/wiki/Funci%C3%B3n_de_p%C3%A9rdida
- Funciones de Coste - Redes Neuronales. Diego Calvo. Dic 10, 2018. Link: https://www.diegocalvo.es/funcion-de-coste-redes-neuronales/
- Categorical crossentropy. Link: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy
- Tensorflow para pincipiantes(IV): Uso de API Keras. JC Gonzalez. 2 de Febrero de 2018. Link: https://www.apsl.net/blog/2018/02/02/tensor-flow-para-principiantes-vi-uso-de-la-api-keras/

- TensorFlow Core v2.3.0. 2020-09-05 UTC. Link: https://www.tensorflow.org/api_docs
- Keras. Link: https://keras.io/
- Curso intensivo de Aprendizaje Automático. Development Google. Link: https://developers.google.com/machine-learning?hl=es-419
- sklearn.metrics.confusion_matrix. scikit-learn. Link: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

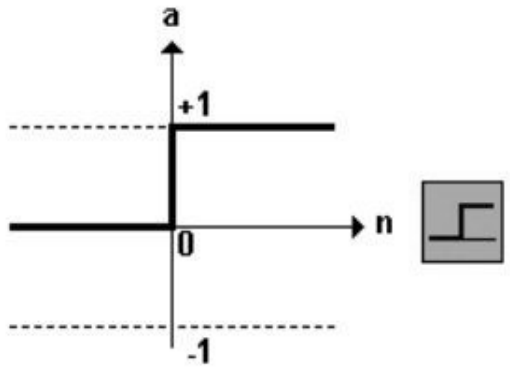
Apéndice

Funciones de Transferencia

Existen distintos tipos de funciones de transferencia, pero todas cumplen la misma función que es, ni más ni menos, que determinar la salida de acuerdo a la acumulación realizada en la entrada de la neurona. Algunos ejemplos son:

Función umbral binaria

Esta función crea neuronas que clasifican las entradas en dos categorías diferentes.

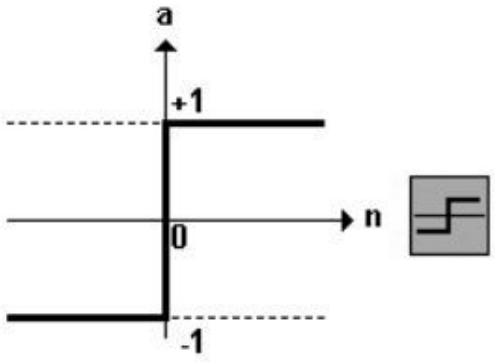


$$a = \begin{cases} 1 & si \quad n \geq 0 \\ 0 & si \quad n < 0 \end{cases}$$

***Fig.* A1.** Función umbral binaria.

Función umbral bipolar

Al igual que la función binaria, esta función crea neuronas que clasifican las entradas en dos categorías diferentes.

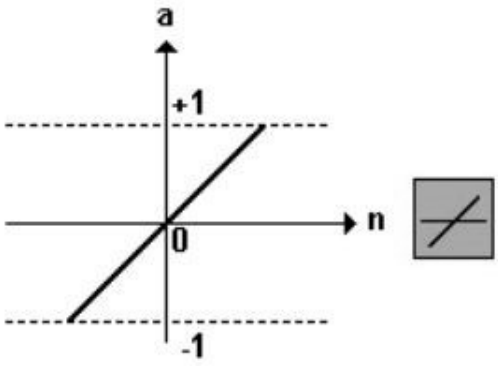


$$a = \begin{cases} 1 & si \quad n \geq 0 \\ -1 & si \quad n < 0 \end{cases}$$

***Fig.* A2.** Función umbral bipolar.

Función lineal

A diferencia de las anteriores, esta función no es acotada.

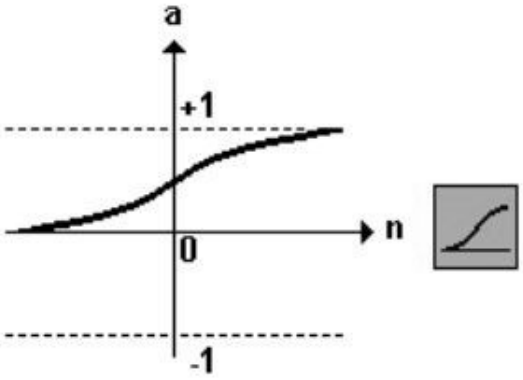


$$a = n$$

***Fig.* A3.** Función lineal.

Función sigmoide

Esta función es acotada y derivable.



$$a = \frac{1}{1+e^{-n}}$$

***Fig.* A4.** Función sigmoide.

Backpropagation

Es importante conocer este concepto a detalle, generalmente se conoce el algoritmo de backpropagation como el encargado de optimizar la función de pérdida para mejorar las predicciones de una red neuronal. Este algoritmo se encarga de calcular las derivadas (o gradientes) de los parámetros W y b para saber cómo estos parámetros afectan al resultado de la función de pérdida, esta es una definición que puede ser usada para explicar el algoritmo, pero para ser más precisos, la optimización de una red neuronal se divide en dos partes, la primera es el algoritmo de backpropagation, como mencione este algoritmo se encarga de ver como los valores de W y de b afectan al resultado de la función de pérdida y la segunda parte es el algoritmo de optimización, este se encarga de optimizar la red neuronal y cambiar los valores de W y de b conforme pasan los

ciclos (o epochs). Existen diferentes algoritmos de optimización, unos son mejores que otros, aunque depende del tipo de problema se esté resolviendo. Este algoritmo es un parámetro de la red neuronal llamado optimizer. Los algoritmos tienen un comportamiento parecido que varía en algunos cálculos pero la idea general es encontrar el global minimum o el mínimo global de la función de pérdida, lo que hace el algoritmo es descender por la función hasta llegar a este punto donde la función se encuentra optimizada, esto se logra con ayuda de las derivadas que indican qué camino se tiene que seguir.

Optimizador

Existen diferentes tipos de algoritmos de optimización el más famoso se llama gradient descent y el más usado se llama Adam. Este es considerado un hiperparametro.
Este hiperparametro indica qué tan largo será el camino que tome el algoritmo de optimización. Si el valor es muy pequeño la actualización puede quedar atrapada en un mínimo local y los valores de W y b no cambiarían correctamente, igualmente la red neuronal tardará mucho más tiempo en optimizar. Por otro lado, si los valores son muy altos la actualización puede pasarse del punto perfecto que sería el mínimo global y nunca encontrarlo y aunque el aprendizaje sea más rápido nunca llegará al mínimo global.

Función de Pérdida

La función de coste trata de determinar el error entre el valor estimado y el valor real, con el fin de optimizar los parámetros de la red neuronal.
En particular, la entropía cruzada categórica (categorical_crossentropy) es una función de pérdida que se utiliza en tareas de clasificación de clases múltiples. Estas son tareas en las que un ejemplo solo puede pertenecer a una de las muchas categorías posibles, y el modelo debe decidir cuál.
Formalmente, está diseñado para cuantificar la diferencia entre dos distribuciones de probabilidad.

La función de pérdida de entropía cruzada categórica calcula la pérdida de un ejemplo calculando la siguiente suma:

$$Loss = -\sum_{i=1}^{n} y_{yo} * log(\widehat{y}_{yo})$$

dónde $\widehat{y}_{yo}$ es el yo-ésimo valor escalar en la salida del modelo, y_{yo} es el valor objetivo correspondiente, y n es el número de valores escalares en la salida del modelo.

Esta pérdida es una muy buena medida de cuán distinguibles son dos distribuciones de probabilidad discretas entre sí. En este contexto, y_{yo} es la probabilidad de que el evento yo ocurre y la suma de todos los y_{yo} es 1, lo que significa que puede ocurrir exactamente un evento.

El signo menos asegura que la pérdida se reduce cuando las distribuciones se acercan entre sí.