3. INTRODUCCIÓN A LAS REDES NEURONALES

En el capítulo anterior se estableció que la función de pérdida se encarga de cuantificar la distancia entre el valor real y el valor estimado por la función objetivo. Más allá de evaluar este error, es importante en ciertos algoritmos de aprendizaje automático, como los de clasificación, interpretar la función de error de forma probabilística.

El objetivo aquí es mapear el vector de error entregado por el modelo hacia un vector de probabilidades. Por ejemplo, para un problema de clasificación, cada elemento de dicho vector contiene la probabilidad de pertenencia a cada una de las clases evaluadas. En cualquier caso, es importante garantizar que los valores de probabilidad estén normalizados entre 0 y 1 y que la suma de todos ellos sea 1. Esto se logra a través de la función *softmax*, que calcula una distribución de probabilidad sobre todas las clases (Zhang, 2021). Esta función se define como:

$$\hat{y} = softmax(o_i) = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{44}$$

Donde los elementos o_i corresponden a los valores entregados por el modelo.

Para ilustrar el funcionamiento de la operación *softmax*, se tiene un modelo cuyos pesos y *bias* ya han sido establecidos, y a partir de los cuales se calculará la salida estimada del modelo como $\hat{y} = \boldsymbol{w}x + b$.

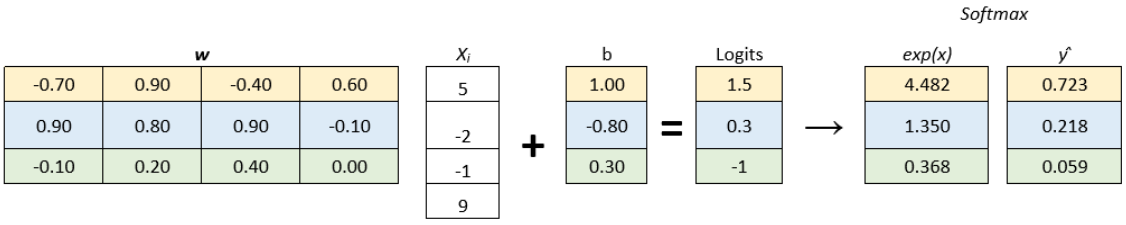


Figura 13. Ejemplo de operación softmax

El resultado de esta operación corresponde al vector de predicciones no normalizadas generado por el modelo de clasificación, y que se denomina en la

figura como el vector "*Logits*". Normalmente, este vector es la entrada a una función de normalización, que para el caso de un problema de clasificación multiclase suele ser la función *softmax*. De aquí que esta función genera el vector de probabilidades (normalizadas) con un valor para cada clase posible (vector $\hat{y}$ en la figura).

El resultado de la operación *softmax* también se usa en una de las funciones de pérdida más comunes en aprendizaje profundo, la pérdida de entropía cruzada (*cross entropy loss*). Esta función de pérdida evalúa la diferencia entre la probabilidad asignada por el modelo (dada por la operación *softmax*) y el valor esperado de la pérdida. Para un problema de clasificación multiclase, la pérdida de entropía cruzada está dada por:

$$L_{CE} = -\sum_{1}^{C} y_i \log(\hat{y}_i) \tag{45}$$

Donde *y* es el vector de etiquetas verdaderas (por ejemplo, con codificación *one-hot*), p_i es la probabilidad *softmax* para la *i*-ésima clase y *C* es el número de clases. Esto quiere decir que la entropía cruzada calcula el negativo de la función de verosimilitud logarítmica de la probabilidad predicha asignada a la etiqueta verdadera.

Para el caso binario, la función de pérdida se reduce a dos casos:

$$l = -\sum_{1}^{C} y_i \log(\hat{y}_i) \tag{46}$$

$$l = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{47}$$

La cual se puede generalizar para todos los ejemplos del conjunto de datos (N)[12]:

$$L = -\frac{1}{N}\sum_{1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{48}$$

[12] https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

PERCEPTRÓN MULTICAPA

La red neuronal más sencilla se puede construir apilando múltiples capas de neuronas, donde cada capa está totalmente conectada a la capa anterior. En esta arquitectura, por lo general, las primeras L-1 capas corresponden a la etapa de representación o extracción de características, y la última capa corresponde al predictor lineal. La incorporación de una o más capas ocultas está orientada a mejorar la generalización de los modelos, desde el punto de vista de linealidad. Este tipo de arquitectura se conoce como perceptrón multicapa (MLP), y fue inicialmente propuesto con una capa oculta por (Rosenblatt, 1958), a partir de lo cual evolucionó manteniéndose vigente en la actualidad. Esta arquitectura se muestra en la siguiente Figura (Saravia, 2021).

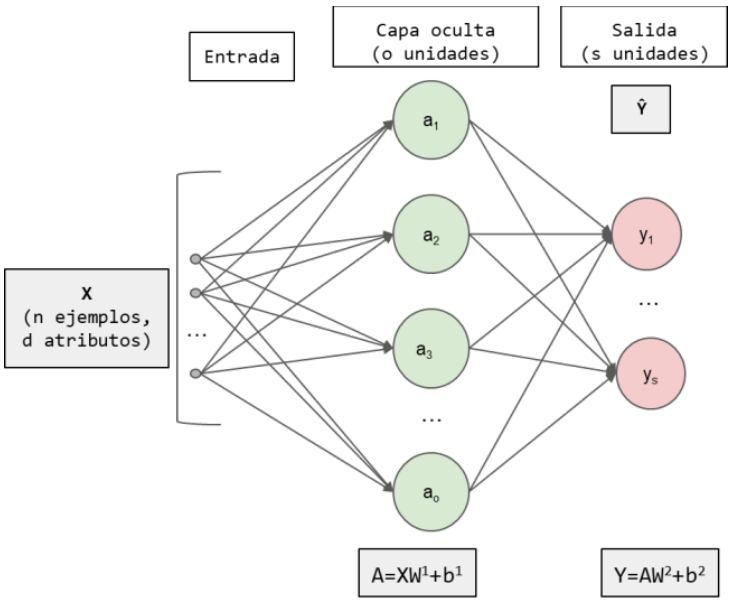


Figura 14. Ejemplo de estructura de red perceptrón.

En la arquitectura de la Figura se identifican tres capas: la de entrada, una capa oculta y la capa de salida.

Es común en este tipo de modelos, que, para definir la profundidad de la red o número de capas apiladas, no se considere la capa de entrada, por lo cual se dirá que este modelo tiene dos capas de procesamiento o profundidad. A continuación, se listan las características de los datos asociados a las tres capas mencionadas.

- Capa de entrada: corresponde al vector de datos de entrada (**x**), por lo cual el número de elementos o unidades de la capa corresponde al número de atributos de los datos (*d*). Para un conjunto de datos con *n* ejemplos y *d* atributos, el dataset se puede ver como una matriz **X** de *n* filas y *d* columnas.
- Capa oculta: esta capa involucra neuronas o unidades que se conectan a los datos de entrada por lo cual constituye una suma ponderada, donde los pesos ($\mathbf{w^1}$) y sesgo (b^1) de dicha suma son parámetros que aprende la red. El número de neuronas o unidades se puede definir al diseñar la red, y se denominará como *o*. En este sentido, tras la interconexión de la capa oculta con los datos de entrada, se tendrá una transformación del vector de datos de entrada (**x**) hacia un nuevo vector ***a***, dada por $\mathbf{a}=\mathbf{w}^1x+b^1$. Si se generaliza la operación de esta capa para todo el conjunto de datos de entrada, se tendrá una matriz de características **A**, dada por:

$$\boldsymbol{A} = \boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1 \tag{49}$$

  A será una matriz cuyo número de filas corresponde al número de ejemplos (*n*) del dataset y el número de columnas es el número de unidades de la capa (*o*).

  $\mathbf{W}^1$ y $\mathbf{b}^1$ corresponden a los parámetros (pesos y *bias*) de la primera capa. Las dimensiones de $\mathbf{W}^1$ son $d \times o$, mientras que $\mathbf{b}^1$ es un vector de *o* elementos.

- Capa de salida: esta capa involucra neuronas o unidades que se conectan a los datos ocultos (**A**) por lo cual constituye una suma ponderada, donde los pesos ($\mathbf{w^2}$) y sesgo (b^2) de dicha suma son parámetros que aprende la red. El número de neuronas o unidades se puede definir al diseñar la red, y se denominará *s*. En este sentido, tras la interconexión de la capa de salida con los datos de la capa oculta, se tendrá una transformación del vector (**a**) hacia un nuevo vector $\hat{\mathbf{y}}$, dado por $\hat{\mathbf{y}}=\mathbf{w}^2a+b^2$. Si se generaliza la operación de esta capa para todo el conjunto de datos de entrada, se tendrá un vector de salida $\hat{\mathbf{Y}}$, dado por:

$$\hat{\boldsymbol{Y}} = \boldsymbol{A}\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{50}$$

  Y será un vector cuyo número de elementos corresponde al número de unidades de la capa de salida (s).

$\mathbf{W}^2$ y $\mathbf{b}^2$ corresponden a los parámetros (pesos y *bias*) de la segunda capa. Las dimensiones de $\mathbf{W}^2$ son $o \times s$, mientras que $\mathbf{b}^2$ es un vector de s elementos.

Considerando el procesamiento de las dos capas mencionadas, la salida se puede calcular a partir de la entrada mediante (Zhang, 2021):

$$\widehat{\boldsymbol{Y}} = (\boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1)\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{51}$$

$$= \boldsymbol{X}\boldsymbol{W}^1\boldsymbol{W}^2 + \boldsymbol{b}^1\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{52}$$

Definiendo el producto $\boldsymbol{W}^1\boldsymbol{W}^2$ como una sola matriz $\boldsymbol{W}$ y $\boldsymbol{b}^1\boldsymbol{W}^2 + \boldsymbol{b}^2$ como un solo vector de sesgo, la salida se puede calcular como:

$$\widehat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b} \tag{53}$$

Lo anterior significa que, incluso agregando más capas al modelo, existe el riesgo de que el modelo se comporte de manera lineal, evitando obtener funciones más generales que puedan modelar de una mejor manera el comportamiento de los datos. Para evitar que la red presente un comportamiento lineal, los modelos han involucrado el uso de funciones no lineales, conocidas como funciones de activación.

FUNCIONES DE ACTIVACIÓN

A partir del valor de la suma ponderada más el sesgo en una capa, una función de activación decide si una neurona debe activarse o no. Es decir, al valor entregado por cada unidad oculta de una capa ($\boldsymbol{wx} + \boldsymbol{b}$), se le aplica una función no lineal ($f(x)$, función de activación). El valor resultante será la entrada para la siguiente capa. Para el caso de la arquitectura mostrada en la figura anterior, el cálculo de cada capa quedaría expresado como:

$$\boldsymbol{A} = f_1(\boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1) \tag{54}$$

$$\widehat{\boldsymbol{Y}} = f_2(\boldsymbol{A}\boldsymbol{W}^2 + \boldsymbol{b}^2) \tag{55}$$

Donde $f_1(x)$ y $f_2(x)$ corresponden a las funciones de activación de la primera y segunda capa respectivamente.

En la literatura se han propuesto una gran diversidad de funciones de activación, entre las cuales se resaltan las funciones sigmoide, tangente hiperbólica y ReLU (*Rectified Linear Unit*) por su amplia adopción. A continuación, se relacionan las principales características de estas funciones.

Función sigmoide

Función de activación que transforma el dominio de los datos de entrada hacia el intervalo (0, 1), Está definida como:

$$f_{sigmoide}(x) = \frac{1}{1+e^{-x}} \tag{56}$$

Esta función se utiliza comúnmente en las unidades de la capa de salida de clasificadores binarios, cuando se desea interpretar el valor de salida como una probabilidad (Zhang, 2021). Es decir, para problemas binarios realiza una tarea similar a la de la operación *softmax* en problemas multiclase. Esto se puede observar en la gráfica de la función, donde es fácil discriminar valores superiores/inferiores a 0.5.

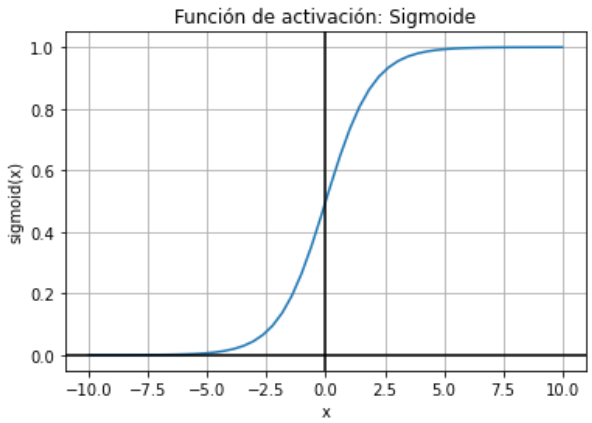


Figura 15. Función de activación sigmoide

Sin embargo, algunos inconvenientes de esta función están relacionados con el comportamiento asintótico que presenta la curva al aproximarse a 0 y a -1, y a que existen funciones más sencillas de implementar a nivel computacional. De aquí que, aunque inicialmente se consideró el uso de la función sigmoide para capas ocultas, actualmente esta ha sido reemplazada por la función de activación ReLU.

Función tangente hiperbólica

Función de activación similar a la *sigmoide*, pero que tiene la ventaja de estar centrada en cero. Esta función transforma el dominio de los datos de entrada hacia el intervalo $(-1, 1)$ y se define como:

$$f_{tanh}(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{57}$$

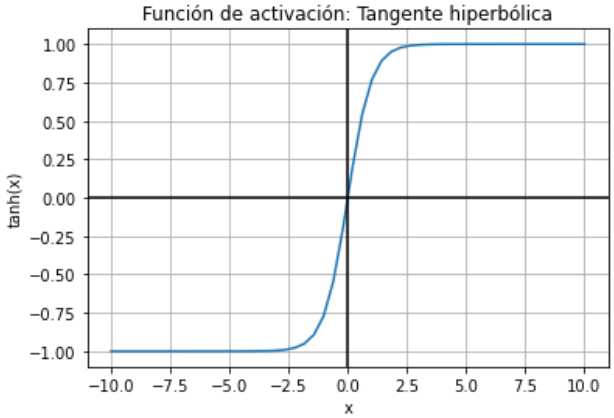


Figura 16. Función de activación tangente hiperbólica

La desventaja de esta función consiste en que sigue presentando un comportamiento asintótico en sus extremos, lo que ocasiona que prácticamente el gradiente de la función en esas zonas desaparezca.

Función ReLU (Rectified Linear Unit)

La función de activación ReLU (Rectified Linear Unit) se comporta como un rectificador de los valores de entrada, es decir que anula los valores negativos y mantiene los valores positivos. De aquí su sencillez de implementación, a la vez que se comporta como una función no lineal de bajo costo computacional. Esta función se define como:

$$f_{ReLU}(x) = max(0, x) \tag{58}$$

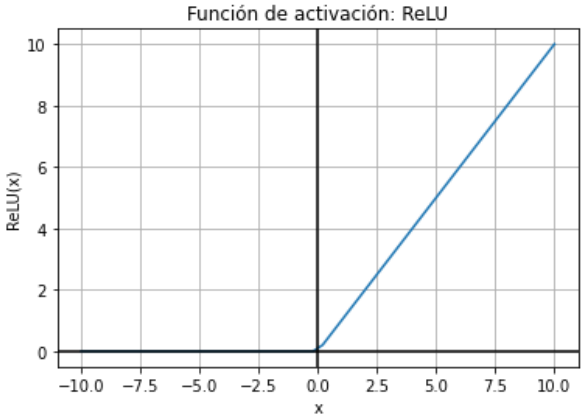


Figura 17. Función de activación ReLU

Ventajas adicionales de ReLU implican la no saturación en el semiplano derecho o una convergencia más rápida en comparación con las dos funciones descritas anteriormente. Como desventaja se tiene que la función no está centrada en cero y su comportamiento constante en el semiplano izquierdo.

Como variantes de la función ReLU se tiene LeakyReLU, la cual no anula la función cuando la unidad no está activa (semiplano izquierdo), sino que permite configurar una pequeña pendiente predefenida. Esto garantiza que la función no tenga saturación, mejora la eficiencia computacional y acelera la convergencia. Otra variante es PReLU (Parametric Rectified Linear Unit), la cual incluye también la inclusión de un pequeño gradiente en la parte negativa, que en este caso corresponde a un arreglo de las mismas dimensiones de la entrada y que es un parámetro que aprende la red.

Función ELU (Exponential Linear Unit)

Variante de la función ReLU definida por la siguiente ecuación:

$$f_{ELU}(x) = \begin{cases} x & si \quad x > 0 \\ \alpha(e^x - 1) & si \quad x < 0 \end{cases} \tag{59}$$

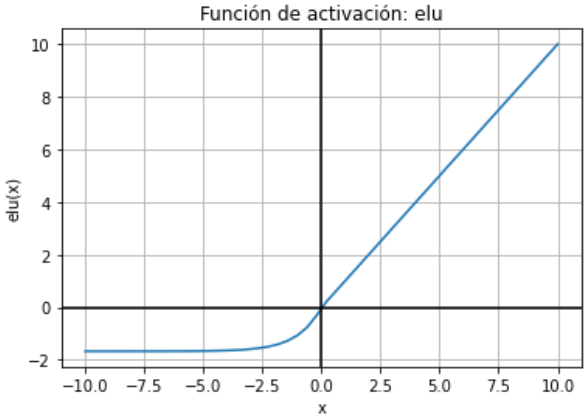


Figura 18. Función de activación ELU

Donde α es un hiperparámetro que controla el valor al que se satura la función para valores negativos, con el fin de disminuir el efecto del gradiente nulo o de fuga (Clevert, 2015). Esto es importante tenerlo en cuenta, ya que cuando hay saturación, la variación de la función y la información que se propaga a la siguiente capa disminuye. Asimismo, en ELU, al contar con valores negativos (en comparación con ReLU), la media de los valores es más cercana al gradiente natural (cero) lo que redunda en mayor velocidad de aprendizaje.

Función SELU (Scaled ELU)

Variante de la función ELU definida por la siguiente ecuación:

$$f_{SELU}(x) = \begin{cases} s * x & si \quad x > 0 \\ s * \alpha(e^x - 1) & si \quad x < 0 \end{cases} \tag{60}$$

Donde α y s (*scale*) son constantes predefinidas (α=1.67326324 y s=1.05070098 en Keras[13]). Estos valores deben elegirse de tal forma que la media y la varianza de las entradas se conserven entre dos capas consecutivas (Klambauer, 2017). La diferencia fundamental ente SELU y ELU consiste en la adición de un valor de escala que asegura una pendiente mayor a 1 en los valores positivos.

[13] https://keras.io/api/layers/activations/

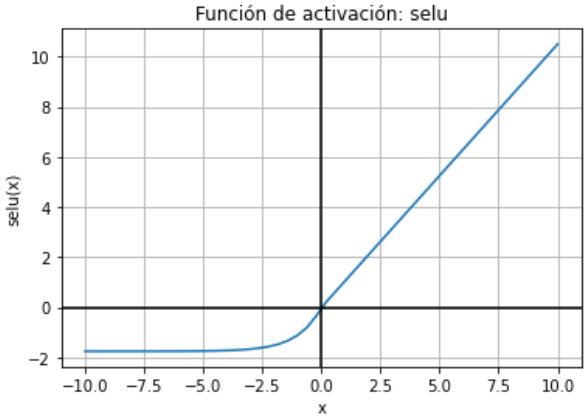


Figura 19. Función de activación SELU

Función softmax

Como se comentó al inicio de este capítulo, la función *softmax* convierte un vector de valores en una distribución de probabilidad, de tal forma que los elementos del vector de salida están en el rango (0, 1) y suman 1. La función *softmax* se utiliza comúnmente como la activación para la última capa de un modelo de clasificación multiclase, ya que su salida puede interpretarse como una distribución de probabilidad[14]. La función está definida tal como se explicó anteriormente.

$$f_{softmax}(x) = \frac{e^x}{\sum e^x} \tag{61}$$

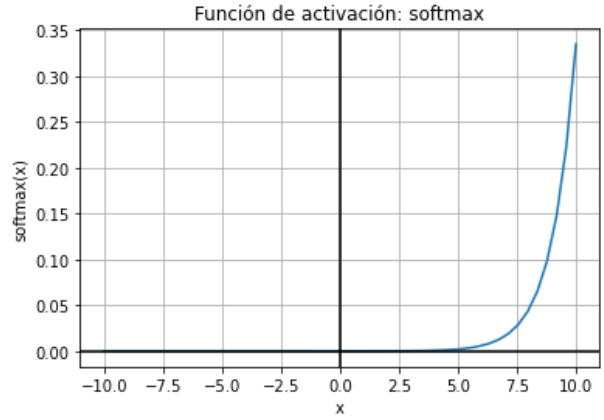


Figura 20. Función de activación Softmax

[14] https://keras.io/api/layers/activations/

PERCEPTRÓN MULTICAPA - EJEMPLO EN PYTHON

Una vez revisados los conceptos sobre redes neuronales y aspectos asociados al aprendizaje de estas, como la función de pérdida, los algoritmos de optimización, las funciones de activación y los hiperparámetros del modelo, se mostrará a continuación un ejemplo que involucra estos conceptos en un solo modelo programado en Python.

En primer lugar, será necesario importar las librerías necesarias para la ejecución del código. En este caso se trabajará con `numpy` para el manejo de datos, la selección aleatoria de imágenes mediante `random`, `tensorflow` para la creación del modelo y `matplotlib` para graficar el rendimiento en el entrenamiento del modelo.

```
# Carga de librerías
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
```

Adicional a las citadas librerías, se importará la clase `Sequential` para crear un modelo secuencial que contiene solamente capas tipo `Flatten` para convertir datos matriciales (imágenes) en vectores y capas totalmente interconectadas (`Dense`) como la base del perceptrón. Por su parte, las etiquetas se representarán en formato *one-hot*, utilizando `to_categorical`.

```
# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.utils import to_categorical
```

Conjunto de datos

Como conjunto de datos del ejemplo se utilizará uno de los más tradicionales en el área de aprendizaje automático con imágenes: MNIST. Este dataset constituido a partir del trabajo de Yann LeCun en el reconocimiento con redes neuronales de dígitos escritos a mano en el código postal de EE. UU. (LeCun Y. B., 1989). MNIST cuenta con 70000 imágenes en escala de gris con una dimensión de 28×28 y

datos en formato de 8 bits sin signo (uint8), etiquetados del 0 al 9. Debido a su gran difusión y utilización, este dataset está disponible en *frameworks* como Keras/tensorflow, donde es posible cargarlo con una sola línea de código. En este caso, la carga involucra una separación de los datos con 60000 imágenes de entrenamiento y 10000 imágenes de prueba.

```
(x_train, y_train), (x_test, y_test) = tf.keras.da
tasets.mnist.load_data()
print(x_train.shape, x_test.shape)
print(y_train.shape, y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
(60000, 28, 28) (10000, 28, 28)
(60000,) (10000,)
```

Tras ejecutar el código de carga, los datos se descargan desde el repositorio de tensorflow y se almacenan en tuplas de arreglos de Numpy. Las dos primeras variables corresponden a los 60000 datos de entrenamiento (`x_train`: imágenes, `y_train`: etiquetas) y las dos últimas a los 10000 datos de prueba (`x_test`: imágenes, `y_test`: etiquetas). Es posible corroborar las dimensiones del dataset, teniendo en cuenta que las dimensiones del arreglo corresponden al número de datos (60000 en entrenamiento y 10000 en prueba), número de filas y número de columnas (28×28). En el caso de las etiquetas, estas corresponden a un vector por lo cual solo se muestra una dimensión (número de ejemplos).

Para involucrar el número de canales o bandas de la imagen, a continuación, se realiza un redimensionado de los datos, agregando una cuarta dimensión que permite obtener tensores de 4 dimensiones: número de ejemplos, alto, ancho, canales.

```
x_train = x_train.reshape( (x_train.shape[0], 28,
28, 1))
print(x_train.shape)
```

```
(60000, 28, 28, 1)
```

Dado que los datos o intensidades de píxel de las imágenes están representados en uint8 (0-255), es recomendable normalizarlos, acotando su valor, por ejemplo, entre 0 y 1. Esto se debe a que trabajar con valores altos en una red neuronal puede hacer que los pesos de la red sean más difíciles de ajustar.


```
x_train = x_train.astype('float32') / 255.0
```

En el caso de las etiquetas, estas se representan de acuerdo con el dígito de la imagen. Es decir, la etiqueta de cada imagen corresponde a un valor entero entre 0 y 9. Esto se puede evidenciar visualizando los valores del arreglo:

```
y_train
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Los valores de etiquetas con los cuales se trabaje en el algoritmo de clasificación pueden representarse fundamentalmente de dos formas: con valores enteros (como el de la variable y_train), o con una representación conocida como "*one hot encoding*", que consiste en convertir cada valor de etiqueta hacia un vector binario de n elementos, siendo n el número de clases del dataset; todos los elementos de este vector serán cero, excepto por el elemento que ocupa la posición del valor entero original. Por ejemplo, para diez clases, el "0" se representará como [1 0 0 0 0 0 0 0 0 0], o el "8" se representará como [0 0 0 0 0 0 0 0 1 0]. Al convertir las etiquetas de los datos de entrenamiento, es posible comprobar cómo el segundo elemento y el último elemento de `y_train` y corresponden a los ejemplos dados.

```
y_train = to_categorical(y_train)
y_train
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]], dtype=float32)
```

Es importante tener en cuenta aquí que el tipo de representación de las etiquetas deberá estar acorde con el tipo de función de pérdida utilizada. En Keras[15], por ejemplo, para un problema de clasificación binario (dos clases), es posible utilizar la función de pérdida *BinaryCrossentropy*, con etiquetas representadas como 0 o 1, y valores de predicción ya sea en *logits* o en valores de probabilidad. Por su parte, para un problema de clasificación multiclase, la herramienta permite utilizar etiquetas representadas en entero, caso en el cual se utilizará una función

[15] https://keras.io/api/losses/

de pérdida tipo *Sparse* (*sparse_categorical_crossentropy*, por ejemplo), o etiquetas con representación *one-hot*, caso en el cual se utilizará una función de pérdida categórica (*categorical_crossentropy*, por ejemplo). En cualquier caso, es posible especificar si la predicción está como *logits* o como probabilidades.

Hiperparámetros

Como hiperparámetros, en este ejemplo se utilizará un tamaño de lote de 32, una tasa de aprendizaje de 0.1 para un optimizador SGD y se entrenará durante veinte épocas.

```
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 20
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)
```

Modelo

En este ejemplo se crea un modelo secuencial que corresponde a un perceptrón de dos capas: una capa oculta de 256 unidades y una capa de salida cuyo número de unidades es igual al número de clases del problema (10). Dado que los datos de entrada tienen más de una dimensión (ancho, alto, canales), la primera capa aplana la entrada, convirtiéndola en un vector de w×h×c elementos, donde w es el número de columnas, h es el número de filas y c es el número de canales. Con este vector, es posible utilizar capas totalmente interconectadas.

```
modelo = Sequential([
                    Flatten(),  # 28×28×1=784
                    Dense(256, activation='sigmoid
'),
                    Dense(10, activation='softmax'
)])
```

En un modelo creado en Keras, es posible especificar la función de activación para cada capa. En este ejemplo se ha utilizado la función `'sigmoid'` para la capa oculta. En la capa de salida (clasificación), cuando el número de unidades corresponde al número de clases, es común utilizar la función `'softmax'`. Cuando se trata de un problema binario, es posible utilizar una capa `Dense` de 1 unidad y la función de activación `'sigmoid'`, o también utilizar una capa `Dense` de n unidades (clases) y la función de activación `'softmax'`.

Luego de crear el modelo, es necesario configurarlo para entrenamiento. Aquí es posible especificar hiperparámetros como el optimizador, la función de pérdida o métricas de rendimiento:

```
modelo.compile(
  optimizer=optimizerf,
  loss = 'categorical_crossentropy',
  metrics=['accuracy'])
```

A continuación, es posible entrenar el modelo durante un número establecido de épocas o iteraciones en el conjunto completo de datos[16]. Además, es necesario especificar datos y etiquetas de entrenamiento o el tamaño del lote si los datos no han sido ya separados. Argumentos adicionales y opcionales que pueden especificarse, involucran la segmentación del dataset en entrenamiento y validación, aleatorización de los datos, ponderación de clases, entre otros.

```
history = model.fit(x_train,y_train,epochs=num_epo
chs,batch_size=batch_s)
```

```
Epoch 1/20
1875/1875 [==============================] - 7s 3ms/step - loss: 0.5250 - accuracy: 0.8560
Epoch 2/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3090 - accuracy: 0.9100
Epoch 3/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2738 - accuracy: 0.9209
Epoch 4/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2454 - accuracy: 0.9290
Epoch 5/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2199 - accuracy: 0.9373
```

La ejecución del entrenamiento devuelve un diccionario con el registro de valores de pérdida de entrenamiento y valores de las métricas definidas en la compilación, y los correspondientes valores cuando se especifican datos de validación.

Adicionalmente, al obtener el modelo entrenado, es posible visualizar un resumen de la red, que muestra las capas del modelo, las dimensiones de los datos y el número de parámetros.

```
modelo.summary()
```

[16] https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

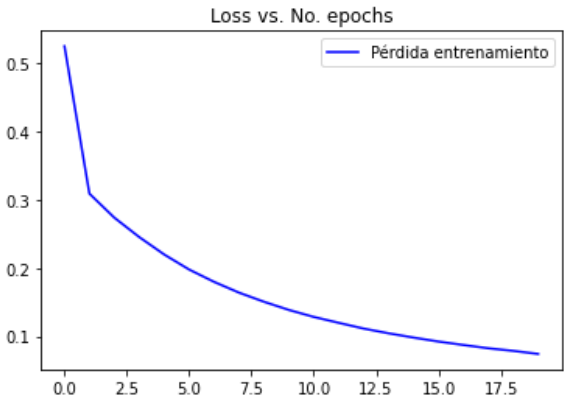

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (32, 784)                 0

 dense (Dense)               (32, 256)                 200960

 dense_1 (Dense)             (32, 10)                  2570

=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
```

Para cada una de las capas Dense, el número de parámetros equivale al número de unidades multiplicado por el número de atributos de entrada de la capa (w) más el número de unidades de la capa (b).

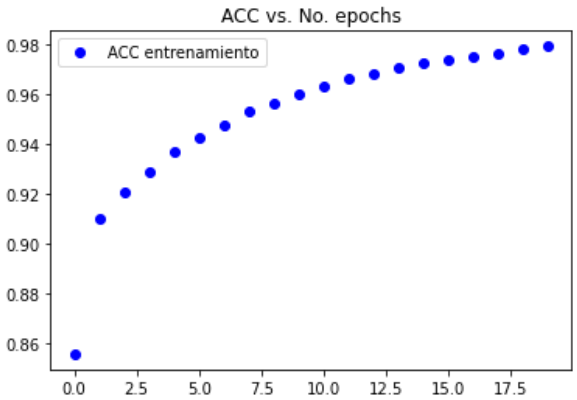
Visualizar evolución del entrenamiento

Utilizando el diccionario `history`, es posible graficar la evolución del valor de pérdida o de una métrica dada en función de la época de entrenamiento. La visualización de este comportamiento dependerá de la librería utilizada. A continuación, se muestran dos ejemplos básicos utilizando `matplotlib`. Sin embargo, se invita al lector a profundizar en estas herramientas.

```
# Gráfica de pérdida
loss = history.history['loss']
epochs = range(len(loss))
plt.plot(epochs, loss, 'b', label='Pérdida entrena
miento')
plt.title('Loss vs. No. epochs')
plt.legend()
plt.show()
```



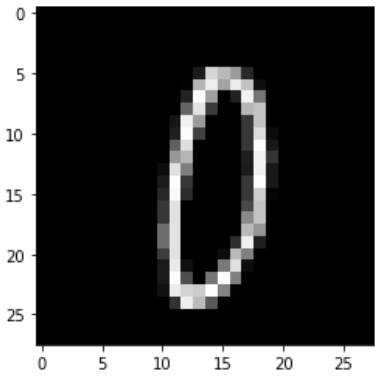

```
# Gráfica de métrica Accuracy
acc = history.history['accuracy']
epochs = range(len(acc))
plt.plot(epochs, acc, 'bo', label='ACC entrenamien
to')
plt.title('ACC vs. No. epochs')
plt.legend()
plt.show()
```



Prueba del modelo

Para evaluar el modelo se realizará una predicción sobre una nueva imagen utilizando el modelo entrenado. El resultado será la categoría que predice el modelo, es decir, el dígito. Como imagen de prueba, se seleccionará al azar una imagen del conjunto de datos de prueba, es decir de datos que no conoció el modelo cuando fue entrenado.

```
image = random.choice(x_test)
plt.imshow(image, cmap=plt.get_cmap('gray'))
plt.show()
```



A continuación, se realiza un pre-procesamiento similar al de los datos de entrenamiento (redimensionamiento y normalización):

```
image = (image.reshape((1, 28, 28, 1))).astype('float32') / 255.0
```

Finalmente, con el modelo entrenado se genera la predicción de salida para la imagen de entrada, utilizando el método `predict`. Aunque aquí se utiliza este método para una sola imagen, este método está diseñado para el procesamiento por lotes de un gran número de entradas.

```
modelo.predict(image)[0]
```

```
array([9.5489484e-01, 4.4818721e-06, 2.6543860e-03, 8.5353706e-04,
       3.6025063e-05, 3.5713132e-02, 1.8909280e-04, 3.4984839e-04,
       2.5580593e-03, 2.7466319e-03], dtype=float32)
```

La salida corresponde a un vector de predicciones tipo NumPy, que en este caso representan las probabilidades de pertenencia a cada una de las diez clases. Es decir, la suma de los diez valores es igual a 1, donde el mayor valor corresponde a la clase más probable. En el ejemplo mostrado, la clase más probable es la cero. Para solo mostrar el índice (clase) del valor máximo de un arreglo NumPy (a lo largo de un eje) se utiliza `argmax` de NumPy.

```
digit = np.argmax(modelo.predict(image)[0], axis=-1)
print("Prediction: ", digit)
```

```
Prediction:  0
```

MÉTRICAS DE EVALUACIÓN DE MODELOS DE CLASIFICACIÓN

El diseño de un algoritmo de clasificación por lo general está asociado a una métrica objetivo, la cual se puede evaluar durante el entrenamiento (con datos de entrenamiento y/o validación), y con datos de prueba. Durante el entrenamiento y validación, esta métrica permite evaluar no solamente el desempeño del algoritmo, sino también ajustar hiperparámetros del modelo y comparar diferentes versiones para facilitar la selección del modelo final. Durante

la fase de prueba, es posible evaluar la capacidad de generalización del modelo, con datos que no fueron utilizados ni en entrenamiento ni en validación.

Existe una gran diversidad de métricas de evaluación para modelos de clasificación, y la selección de la métrica o métricas objetivo dependerá del problema a resolver (teniendo en cuenta qué se espera del modelo) y su contexto de aplicación (por ejemplo, para efectos de comparación con el estado del arte). De cualquier forma, la gran mayoría de las métricas de evaluación de algoritmos de clasificación, como el *accuracy* o el F1-Score están basadas en la construcción de una matriz de confusión.

La matriz de confusión o matriz de error[17] corresponde a una tabla que permite visualizar el rendimiento de un algoritmo de clasificación, donde cada columna corresponde a la condición real de las clases y cada fila corresponde a la condición vaticinada en cada clase, o viceversa. La estructura de la matriz de confusión se muestra a continuación:

		Condición verdadera	
		Condición positiva	Condición negativa
Predicción	Predicción clase positiva		
	Predicción clase negativa		

El cruce de la condición real contra la condición predicha por el modelo facilita ver si el modelo etiqueta erróneamente una clase como otra, es decir si las confunde. Para ello, los resultados de clasificación se estructuran en cuatro grupos: Verdaderos positivos (TP), verdaderos negativos (TN), falsos negativos (FN) y falsos positivos (FP). Para realizar esta clasificación, es necesario establecer previamente cuál de las dos clases será la positiva (clase 1) y cuál la negativa (clase 0), donde la clase positiva por lo general se relaciona con el objetivo del modelo, es decir, corresponde a la clase que tiene mayor relevancia en la clasificación.

[17] https://en.wikipedia.org/wiki/Confusion_matrix

En una matriz de confusión binaria, TP y TN corresponden a las muestras clasificadas correctamente (para las clases 0 y 1, respectivamente), FN corresponde a muestras de la clase 1 que fueron clasificadas incorrectamente y FP corresponde a muestras de la clase 0 clasificadas como clase 1. La relación de TP, TN, FP y FN en una matriz de confusión se muestra a continuación:

		Condición verdadera	
		1	0
Predicción	1	**TP**	**FP**
	0	**FN**	**TN**

Para ilustrar el cálculo de algunas métricas basadas en una matriz de confusión, se utilizará el resultado de los dos modelos de clasificación que se muestran a continuación:

Tabla 2. Ejemplo de datos para matriz de confusión

	Modelo 1	**Modelo 2**
No. de ejemplos evaluados	200	200
TP	75	100
FN	25	50
FP	25	0
TN	75	50

Accuracy (ACC, exactitud)

Esta métrica evalúa el porcentaje de casos correctos (es decir solo TP & TN) en un clasificador. Es una de las métricas más utilizadas y se recomienda su uso cuando se tienen problemas balanceados (similar número de muestras en cada clase), o cuando la importancia de las clases es igual. Sin embargo, no se recomienda su utilización cuando el número de casos por clase difiere significativamente entre sí. El *accuracy* se define por la siguiente Ecuación:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{62}$$

El cálculo de *accuracy* para los dos modelos de ejemplo se muestra a continuación:

Modelo 1	Modelo 2
$ACC_1 = \frac{75 + 75}{200} = 0.75$	$ACC_2 = \frac{100 + 50}{200} = 0.75$

Desde el punto de vista de *accuracy*, ¿cuál de los dos modelos evaluados es el mejor? En este caso, aunque los dos modelos tienen diferentes valores en su matriz de confusión, su valor de *accuracy* es igual. Esto impide determinar cuál de ellos presenta un mejor rendimiento, y será necesario considerar otras métricas.

Precisión (P)

Esta métrica evalúa la precisión en las predicciones positivas, es decir sólo tiene obtiene la relación entre el número de veces donde acertó el modelo en la clase positiva respecto a la cantidad de casos vaticinados como positivos. Se calcula por medio de la siguiente Ecuación:

$$\mathrm{P} = \frac{TP}{TP + FP} \tag{63}$$

El cálculo de *la precisión* para los dos modelos de ejemplo se muestra a continuación:

Modelo 1	Modelo 2
$P_1 = \frac{75}{75 + 25} = 0.75$	$P_2 = \frac{100}{100 + 0} = 1.0$

En este caso, sí es posible discriminar una diferencia entre los dos modelos desde el punto de vista de precisión. En particular, el segundo modelo no presenta casos clasificados erróneamente como positivos, por lo cual la precisión es del 100%.

Recall (R)

El *recall* o *sensibilidad* determina el grado en el cual un modelo logra clasificar correctamente todas las muestras positivas del conjunto de datos evaluado. El recall está dado por:

$$\mathrm{R} = \frac{TP}{TP + FN} \tag{64}$$

Al calcular el *recall* para los dos modelos de ejemplo se obtiene lo siguiente:

Modelo 1	Modelo 2
$R_1 = \frac{75}{75+25} = 0.75$	$R_2 = \frac{100}{100+50} = 0.67$

En este caso, también es posible discriminar una diferencia entre los dos modelos. En particular, aunque el primer modelo clasifica correctamente un menor número de muestras como positivas, también presenta un menor número de casos clasificados erróneamente como positivos, por lo cual la precisión es mayor en relación con la del segundo modelo.

F1-Score

Otro tipo de métricas realizan una combinación de métricas para determinar su balance. En particular, el *F1-Score* calcula la media armónica entre P y R. Es decir, el cálculo del compromiso entre la precisión y el *recall* puede facilitar la selección de un modelo, dado que, en un caso puede ser mejor el modelo 1, mientras que desde el otro punto de vista puede ser mejor el otro. El *F1-Score* se define como:

$$\mathrm{F1} = 2\frac{P * R}{P + R} \tag{65}$$

El *F1_Score* para cada modelo del ejemplo es el siguiente:

Modelo 1	Modelo 2
$F1_1 = 2\frac{0.75}{0.75+0.75} = 0.75$	$F1_2 = 2\frac{1.0 * 0.67}{1.0+0.67} = 0.80$

Ya con el cálculo del F1-Score es posible determinar que el modelo 2 presenta un mejor rendimiento.

Otro tipo de métricas incluyen la curva ROC (*Receiver operation characteristic*) con aplicación en clasificación binaria, la cual relaciona la tasa de falsos positivos versus el *recall* (tasa de verdaderos positivos), o la métrica AUC (área bajo la curva) donde el mejor clasificador presenta un AUC de 1.0.

Como complemento a este tema, se sugiere al lector consultar información adicional relacionada con la generalización de la matriz de confusión en problemas multiclase o métricas diseñadas para problemas no balanceados.

CONJUNTOS DE DATOS

Como se ha comentado hasta aquí, los modelos de aprendizaje supervisado requieren un conjunto de datos cuyo patrón de comportamiento es modelado por el algoritmo durante la fase de entrenamiento. De aquí, que, la elección o construcción del conjunto de datos debe estar orientada de tal manera que su comportamiento refleje el de casos reales (datos similares a los que se esperan cuando el modelo esté implementado) (Ng, 2019).

Durante la fase de entrenamiento es común utilizar datos adicionales, conocidos como datos de validación, cuya función es orientar los cambios más importantes que se deben realizar al modelo para mejorar su rendimiento, es decir que facilitan el ajuste de los hiperparámetros. Por su parte, los datos de prueba permitirán evaluar el rendimiento del modelo con datos que nunca hayan sido utilizados ni en entrenamiento ni en validación, con el fin de evitar sesgos en el diseño del modelo. Lo recomendable es separar estos datos desde la fase inicial y sólo utilizarlos previo a la implementación del modelo o a la difusión de este.

También es recomendable que los datos de validación y prueba se extraigan de la misma distribución. Esto importante dado que durante el entrenamiento se puede correr el riesgo de que el modelo se sobreajuste a los datos de validación, o que lo datos de prueba sean mucho más complejos que los datos de validación o que los datos de prueba sean simplemente diferentes a los datos de validación. En cualquier caso, se tendría que, aunque el rendimiento del modelo presente un alto rendimiento en entrenamiento/validación, su rendimiento en con datos de prueba sea bajo. Es decir, que el modelo no logre generalizar el comportamiento de los datos.

De acuerdo con lo anterior, un conjunto de datos se distribuye en datos de entrenamiento, validación y prueba. El enfoque tradicional en aprendizaje

automático ha sido asignar alrededor de un 70% de los datos para entrenamiento, y 30% para validación y prueba. Sin embargo, estos porcentajes pueden variar en función del tamaño del dataset. Por ejemplo, se puede tener un 60% de datos para entrenamiento y 40% para validación/prueba cuando el número de muestras no es muy grande (ej. ~1000 muestras), o también el porcentaje para validación y prueba podría reducirse a menos del 10% cuando el número de ejemplos sea muy grande (ej. > 100.000 muestras). En cualquier caso, el número de muestras de validación y prueba se defina en función del grado de confianza deseado a la hora de evaluar el rendimiento del algoritmo (Ng, 2019).

En relación con la decisión de si construir un conjunto de datos desde cero o usar un dataset existente para un problema dado, cada alternativa tiene sus particularidades. En el caso de construir un dataset hay que tener en cuenta que la distribución de los datos sea similar a lo que se esperaría en la operación del sistema, evaluar que el número de muestras sea el adecuado para su rendimiento y que el tamaño del conjunto de datos no tenga incidencia directa sobre un posible sub-ajuste o sobreajuste del modelo (dependiendo de la complejidad de este). En el caso de utilizar un dataset existente, se tienen como ventajas que el número de muestras ya ha sido probado, una reducción en el tiempo asociada a la fase de preparación de datos y posibles resultados sobre los datos que ya han sido obtenidos por otros investigadores. Como reto se tendría que es necesario analizar el origen y la distribución de los datos con respecto al problema a resolver.

Fuentes de datos

En los últimos años la disponibilidad de datos para abordar problemas de aprendizaje automático ha crecido sustancialmente, con la posibilidad de contar con datos etiquetados y no etiquetados a nivel de imagen (ej. reconocimiento facial, reconocimiento de acciones, detección y reconocimiento de objetos, escenas naturales, datos geoespaciales), datos de texto (opiniones, noticias, mensajes de texto, redes sociales, chats, respuestas a preguntas), sonido (voz, música), video, sentimientos o sistemas de recomendación.

Este aumento en la disponibilidad de datos está asociado con el aumento de repositorios y buscadores de datasets. Por ejemplo, Google dispone tanto de su propio metabuscador (https://datasetsearch.research.google.com), como de un repositorio donde publican conjuntos de datos que han sido utilizados en sus investigaciones (https://research.google/tools/datasets/). Otra fuente reconocida de conjuntos de datos asociados a desafíos abiertos a la comunidad

es la plataforma Kaggle (https://www.kaggle.com/datasets); aquí es posible encontrar no solo los datos en sí, sino también soluciones y códigos que han sido utilizados para resolver un problema específico y que facilitan la comparación del rendimiento de un modelo respecto a resultados obtenidos por otros investigadores.

Por su parte, la plataforma *Papers with code* (https://paperswithcode.com/datasets) dispone también no solo de artículos de investigación y su implementación a nivel de código, sino también del respectivo conjunto de datos utilizado para obtener los resultados relacionados en los documentos científicos. De aquí que esta plataforma constituye un referente de gran importancia a la hora de evaluar los resultados de un sistema en relación con el estado del arte. Otros repositorios incluyen IEEE Dataport (https://ieee-dataport.org) o Mendeley (https://data.mendeley.com).

SELECCIÓN DE UN MODELO DE APRENDIZAJE AUTOMÁTICO

Configuración inicial del modelo

Como regla general a la hora de enfrentar un nuevo proyecto de aprendizaje automático por primera vez, es necesario definir una configuración inicial que involucra fundamentalmente tres aspectos: configuración del modelo, hiperparámetros del optimizador y número de pasos o épocas de entrenamiento (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023).

Respecto a la configuración del modelo, se sugiere iniciar por una arquitectura simple y relativamente rápida que permita alcanzar un resultado razonable para el problema en cuestión. Otra alternativa, sugiere empezar con un modelo que funcione sobre ese tipo de problema, o un problema similar. Es decir, el modelo base se puede construir a partir de una revisión del estado del arte, con el fin de encontrar un trabajo previo que haya abordado un problema lo más similar posible (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023).

De igual forma, la selección del optimizador sugiere iniciar con el optimizador más popular para el tipo de problema en cuestión y su tasa de aprendizaje predeterminada, de acuerdo con la revisión de trabajos previos. De igual forma, es importante considerar el ajuste de otros hiperparámetros del optimizador como el *momentum*.

En relación con el número de pasos de entrenamiento, su valor presenta un compromiso entre tiempo de ejecución (de cada época de entrenamiento) y rendimiento alcanzado. En este sentido, entrenar durante un mayor número de épocas puede permitir que se obtenga un modelo de mejores prestaciones, pero con un tiempo de ejecución más prolongado, lo que limita el número de experimentos (modelos o variaciones de hiperparámetros) que se pueden ejecutar en un tiempo dado. Aquí es importante también tener en cuenta el tamaño del lote, dado que este influye directamente en la velocidad de entrenamiento. Para el caso de imágenes, tradicionalmente se había sugerido un tamaño de lote igual o superior a 32 (Masters & Luschi, s.f.), sin embargo, estudios recientes sugieren que el tamaño ideal del lote usualmente es el mayor tamaño que soporte el hardware con el cual se entrenará el modelo (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023). Es necesario tener en cuenta que para existen hiperparámetros que interactúan en gran medida con el tamaño del lote, como los específicos de cada optimizador o los de regularización.

Selección del modelo final

A la hora de seleccionar un modelo de aprendizaje automático, es importante tener en cuenta dos conceptos: error de entrenamiento y error de generalización. Como su nombre lo indica, el error de entrenamiento es el que se obtiene al usar los respectivos datos de entrenamiento, mientras que el error de generalización está relacionado con la evaluación del modelo sobre datos procedentes de la misma distribución de datos pero que no han sido conocidos por el modelo ni en entrenamiento ni en validación. Por lo general, la selección de un modelo se realiza luego de evaluar diferentes arquitecturas o modelos candidatos con base en la utilización de datos de validación que permiten escoger el modelo entre los candidatos.

Durante esta evaluación se realiza una evaluación de ajuste (*fitting*) del modelo, considerando si existe un sub-ajuste (*underfitting*) que se presenta cuando el modelo no es capaz de reducir el error de entrenamiento (aquí se dice también que el modelo presenta un alto *bias*), es decir, que el modelo se ajusta mal al conjunto de entrenamiento y sucede también que el error en el conjunto de validación es muy similar al error en entrenamiento. Por otro lado, puede existir sobreajuste (*overfitting*) que ocurre cuando el error de entrenamiento es mucho menor que el error de validación; aquí se dice que el modelo presenta una alta varianza, dado que el clasificador tiene un error de entrenamiento muy bajo, pero no logra generalizar en el conjunto de validación (Ng, 2019).

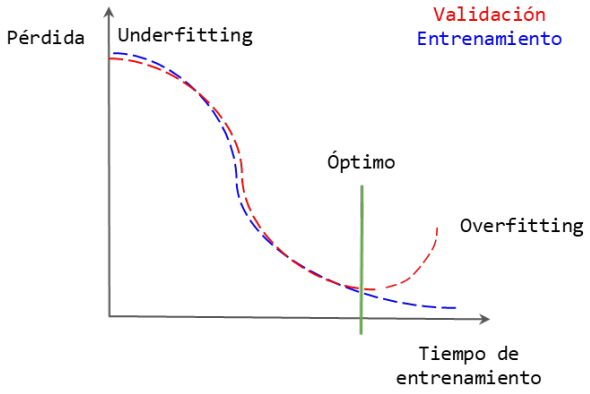


Figura 21. Ilustración de sub-ajuste (*underfitting*) y sobreajuste (*overfitting*) en modelos de aprendizaje automático.

Por lo general la complejidad de un modelo y el tamaño del conjunto de datos tienen una relación directa en el ajuste de este, donde a mayor complejidad del modelo se requiere un conjunto de datos más amplio. En este sentido, si el modelo es susceptible de mejora (es decir que existe posibilidad de reducir el *bias*) lo recomendable es aumentar el tamaño o complejidad del modelo (ej. no. de capas o neuronas); por su parte, si la varianza es alta, lo recomendable es agregar datos al conjunto de entrenamiento o aumentar su diversidad (Ng, 2019).

Sin embargo, existe un compromiso entre el *bias* y la varianza. Por ejemplo, incrementar la complejidad de un modelo puede mejorar el rendimiento (disminuir el *bias*), pero podría agregar *overfitting* (incrementar la varianza). A su vez, la aplicación de una técnica de regularización está orientada a reducir el *overfitting* (disminuir la varianza) pero puede afectar el rendimiento del modelo incrementando la varianza.

Las técnicas principales para mejorar el rendimiento de un modelo (reducir el *bias*), involucran aumentar el tamaño o complejidad de este sin descuidar lo relacionado con el *overfitting* (en caso de presentarse podría utilizare una técnica de regularización), modificar las características de entrada (a partir de un análisis de rendimiento), también en caso de que el modelo cuente con alguna técnica de regularización se puede reducir o eliminar (siempre teniendo en cuenta el compromiso entre *bias* y varianza), y también se puede considerar modificar la arquitectura del modelo (lo que también afecta *bias* y varianza).

Como técnicas de reducción de overfitting, se tiene en primer lugar el aumento de los datos de entrenamiento (en número y diversidad), agregar técnicas de regularización, agregar un criterio de detención temprana (*early stopping*) al

entrenamiento, reducir el número de atributos de entrada o reducir la complejidad del modelo. En cualquier caso, estas técnicas pueden ayudar a reducir la varianza, pero a su vez pueden aumentar el *bias*, por lo que se deben aplicar con precaución.

Regularización

Como se ha comentado hasta aquí, la reducción de overfitting se puede abordar en primera instancia aumentando el número de muestras de entrenamiento o reduciendo la complejidad del sistema ya sea limitando el número de atributos de los datos de entrada, o reduciendo el tamaño del modelo. También se ha comentado la posibilidad de incluir una técnica de regularización. En este aparatado se presentan tres técnicas comunes de regularización: *weight decay*, *dropout* y Batch normalization.

La técnica *Weight decay* consiste en agregar un término de penalización a la función de pérdida durante el entrenamiento con el fin de reducir la complejidad del modelo (Zhang, 2021). Esta técnica se conoce también como regularización L2.

Al aplicar *weight decay*, la Ecuación de pérdida presentada en el capítulo 4 se complementa con un término de penalización que depende de una constante de regularización (λ) que pondera la norma L2 del vector de pesos (por lo general no se regulariza el bias) (Zhang, 2021):

$$L(\boldsymbol{w}, b) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{66}$$

Una vez calculada la función de pérdida junto con el término de penalización, la actualización de los parámetros ($\mathbf{w}$) se realiza tal como se explicó en el capítulo 2.

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i \in BS} \boldsymbol{x^i}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{67}$$

$$b = b - \frac{\alpha}{BS}\sum_{i \in BS} \left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{68}$$

En cuanto al *Dropout*, corresponde a una técnica de regularización que fija en cero y de forma aleatoria *X* porcentaje de unidades de entrada en cada iteración durante el tiempo de entrenamiento[18] (es decir, una vez entrenado el modelo no se aplica el *dropout*).

Dado que se anulan ciertas unidades de entrada, las restantes se escalan a:

$$\frac{1}{1-X} \tag{69}$$

Con el fin de garantizar que el valor de todas las unidades no se altere. Un ejemplo de aplicación de dropout al 40% se muestra en la siguiente Figura (Saravia, 2021).

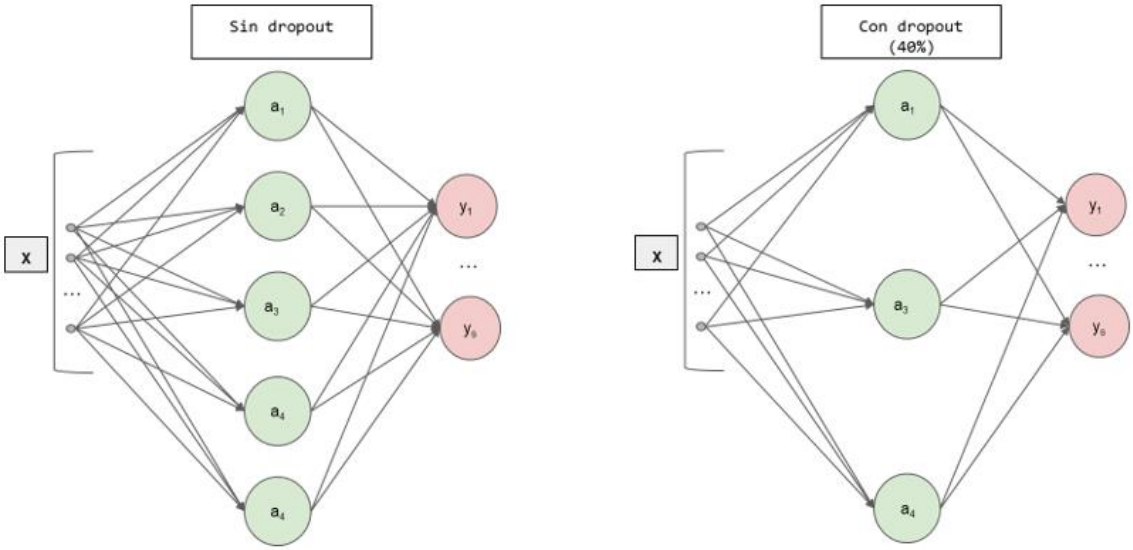


Figura 22. Ejemplo de redes neuronales con y sin regularización por *Dropout*.

Por su parte, *Batch normalization* (BN) es una técnica diseñada para acelerar la convergencia en redes profundas que pueden implicar un mayor desafío a nivel de complejidad y sobreajuste. BN se fundamenta en la importancia del pre-procesamiento de datos de entrada en las redes neuronales, especialmente los relacionados con normalización y estandarización. Dado que las funciones de activación en las capas intermedias de las redes neuronales pueden ocasionar que el rango dinámico de dichos datos sea muy amplio y en consecuencia puedan

[18] https://keras.io/api/layers/regularization_layers/dropout/

tomar valores muy altos, BN propone estandarizar estos datos, de tal forma que su media sea cero y su varianza sea unitaria.

La aplicación de BN en una capa involucra estandarizar los datos de entrada, substrayendo la media de estos y dividiendo por su desviación estándar. Además de aplicar un coeficiente de escalamiento (γ) y un valor offset (β) (Ecuación (71)). Estos dos valores corresponden a hiperparámetros que debe aprender la red durante su entrenamiento (Zhang, 2021).

$$BN(x)\gamma\frac{\boldsymbol{X}-\mu}{\sigma}+\beta \tag{70}$$

Es importante tener en cuenta que la aplicación de BN tiene relación directa con el tamaño del lote, dado que la media y la desviación estándar dependen de dicho valor. Además, su comportamiento presenta diferencias entre el entrenamiento (en función de las estadísticas de cada lote) y la predicción (en función de las estadísticas del dataset de prueba) (Zhang, 2021).

Ejemplo en python: *underfitting*, *overfitting* & regularización

Con el fin de contextualizar los conceptos de overfitting, underfitting y regularización se retomará el ejemplo de la sección anterior (perceptrón multicapa utilizando el dataset MNIST). En este caso, se evaluarán diferentes modelos que ilustran estos conceptos.

La definición de librerías y lectura de datos no tiene variaciones:

```
# Carga de librerías
import numpy as np
#import random
import tensorflow as tf
import matplotlib.pyplot as plt

# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.utils import to_categorical

# Lectura de datos
(x_train , y_train), (x_test , y_test) = tf.keras.dat
asets.mnist.load_data()
```



```
x_train = x_train.reshape( (x_train.shape[0], 28, 28,
 1))

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Modelo 1 (*underfitting*)

- 60 mil ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: SGD
- Capa oculta: 1 FC de 1 unidad

En primer lugar, se muestra un modelo muy simple (1 capa densa con una neurona), que presenta *underfitting*:

```
# Modelo 1 (Underfitting)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(1, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```


Al entrenar este modelo se obtiene un *accuracy* de alrededor del 30% en entrenamiento y en validación. Es decir, el *bias* del modelo es alto (bajo rendimiento).

Modelo 2 (*overfitting*)

- **500** ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de **8 unidades**

Para entrenar el segundo modelo se utilizarán solo 500 ejemplos, pero se aumenta ligeramente la complejidad del modelo a una capa densa de 8 unidades y se utiliza el optimizador Adam.

```
# Modelo 2 (Mejora el rendimiento, pero presenta o
verfitting)
# Datos
x_train_Mod = x_train[1:500,:,:,:]
y_train_Mod = y_train[1:500,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(8, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```

Luego de entrenar este modelo, se alcanza un *accuracy* de 82% en entrenamiento, pero sólo 67% en validación. Es decir, el modelo no logra generalizar de forma óptima, por lo cual existe overfitting.

Modelo 3 (Reducción de overfitting)

- **60000** ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 8 unidades

Para el tercer modelo se utilizan de nuevo los 60000 ejemplos (es decir se aumenta el número de muestras respecto al modelo 2, con el fin de reducir el overfitting).

```
# Modelo 3 (Combatir overfitting reduciendo el tam
año del dataset)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(8, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```

En este caso, se alcanzan valores similares de *accuracy* para entrenamiento y validación (alrededor del 86%), lo que equivale a una reducción de overfitting en el sistema.

Modelo 4 (Reducción de overfitting)

- 60000 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: **20**
- Optimizador: Adam
- Capa oculta: 2 FC de **256 unidades, ReLU**

En cuarto lugar, se trata de mejorar el rendimiento del modelo, aumentando su complejidad y con un mayor tiempo de entrenamiento:

```
# Modelo 4 (Aumentar complejidad del modelo)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 20
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(256, activation='sigmoid
'),
                    Dense(10, activation='softmax'
)])
```

Con este último modelo, el rendimiento mejora y se aproxima al 90% tanto entrenamiento como en validación.

Modelo 5 (*overfitting*)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10

- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU

Para evaluar el impacto de aplicar técnicas de regularización, como punto de partida se tomará un bajo número de muestras de entrenamiento (500) y una complejidad media del modelo:

```
# Modelo 5
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dense(10, activation='softmax'
)])
```

Al entrenar el modelo 5, se obtiene un *accuracy* en entrenamiento del 99% y 84% en validación, por lo cual existe overfitting.

Modelo 6 (regularización L2)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y regularización L2 (*weight decay* de 0.01)

```
# Modelo 6 (Weight Decay)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
```

```
weight_decay1 = 0.02
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(weight
_decay1)),
                    Dense(10, activation='softmax'
)])
```

Los resultados del entrenamiento del modelo entregan un *accuracy* de entrenamiento de 97% y en validación de 83%. Es decir, aunque los resultados son sutiles, se presenta un acercamiento entre el resultado de entrenamiento y el de validación.

Modelo 7 (dropout de 0.2)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y dropout de 0.2

```
# Modelo 7 (Dropout: 0.2)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dropout(0.2),
```



```
                    Dense(10, activation='softmax'
)])
```

Para el modelo 7 se alcanza una pequeña reducción de overfitting respecto al modelo 5. En este caso, se reduce el *accuracy* en entrenamiento al 98% en entrenamiento, manteniendo el 84% de *accuracy* en validación.

Modelo 8 (dropout de 0.5)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y dropout de 0.5

```
# Modelo 8 (Dropout: 0.5)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dropout(0.5),
                    Dense(10, activation='softmax'
)])
```

En este último ejemplo, el grado de dropout es mayor, lo cual impacta principalmente en el *accuracy* de entrenamiento (92%), acercando su valor al rendimiento del algoritmo en datos desconocidos (datos de validación), que para este caso están alrededor del 83%.

