UA

Sistemas Inteligentes

Grupo 5 prácticas

VadymFormanyuk: **x5561410x**

20 de diciembre de 2022

Índice

1. Parte 1. Aprende las bases de un MLP

1.1. Resolviendo una función booleana mediante 'Multi Layer Perceptron'MLP

La función booleana que me ha tocado amí es la siguiente:

$$(\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge d) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge \bar{d}) \vee (a \wedge \bar{b} \wedge \bar{d}) \vee (\bar{a} \wedge \bar{b} \wedge c \wedge d)$$

Figura 1: Función booleana mia

La idea de este ejercicio es realizar el 'entrenamiento' de una red neuronal a mano teniendo en cuenta la siguiente figura:

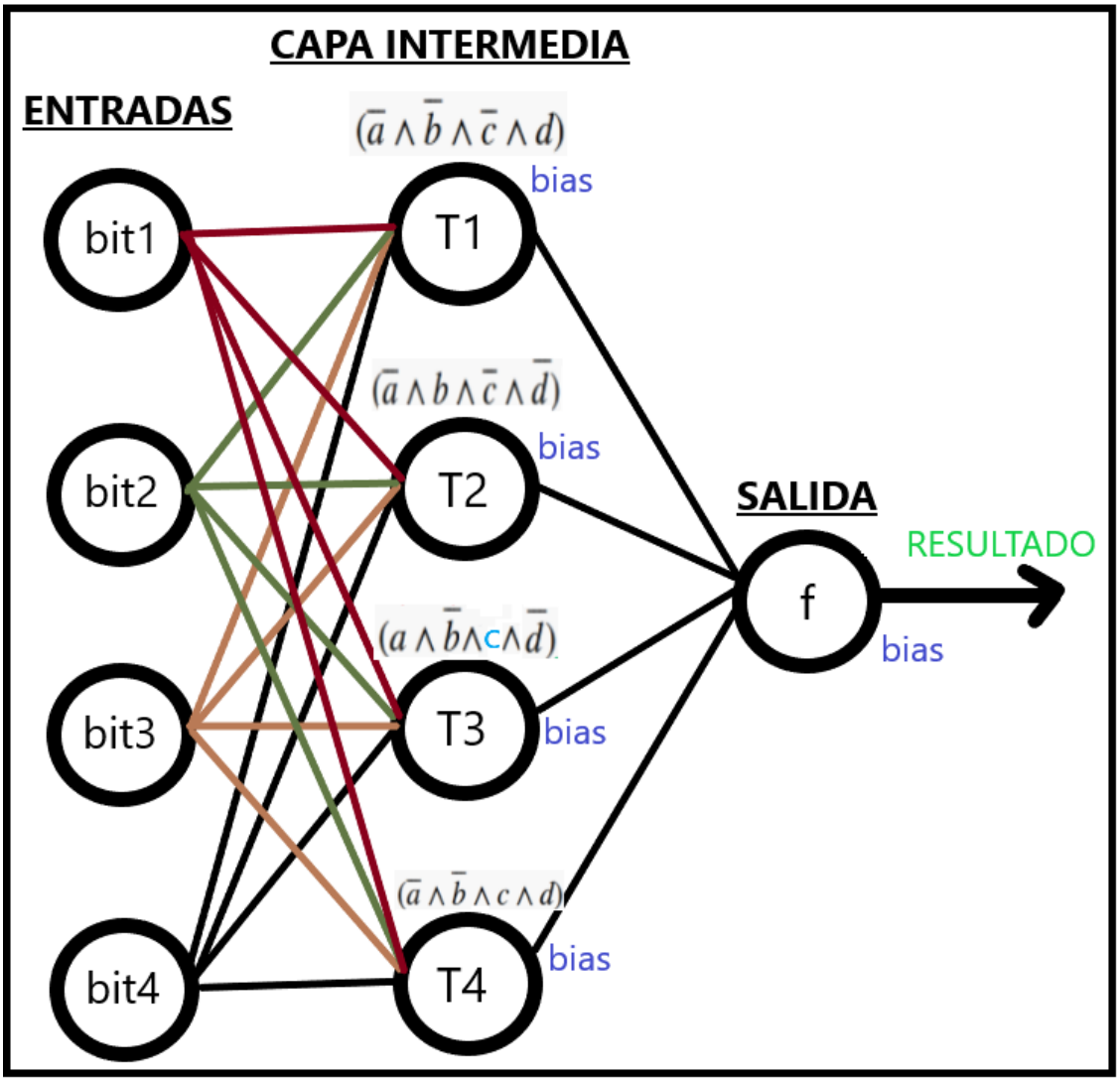


Figura 2: Red neuronal ejercicio

Resuelvo la función booleana empezando por cada tupla (la tupla es cada paréntesis y lo que hay dentro). Al tener 4 tuplas tengo que hacer 4 tablas de bits de 4 bits y aplicarle el proceso 'AND' teniendo en cuenta

que las las letras negadas cambian de valor al contrario (si es 1 pasa a 0 y viceversa), y donde falte un valor ponerle un 0 (tal como se observa en la tercera tupla donde falta el 'c').

Y se me quedaría así:

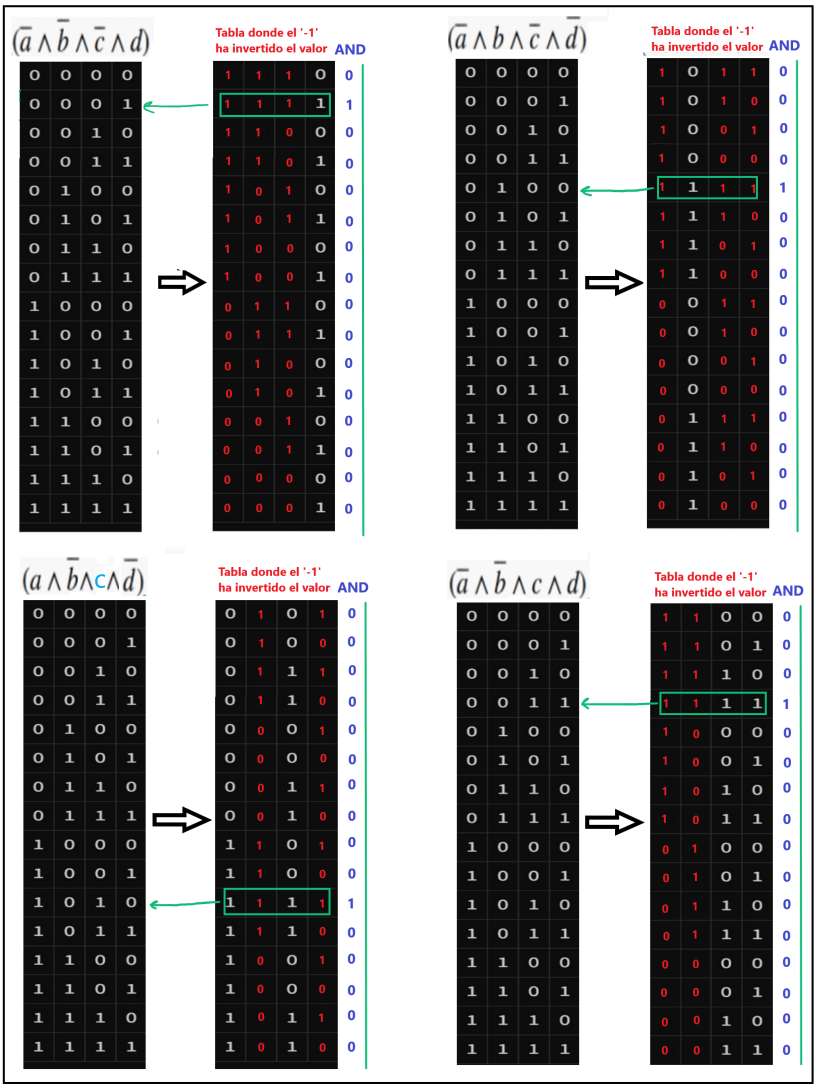


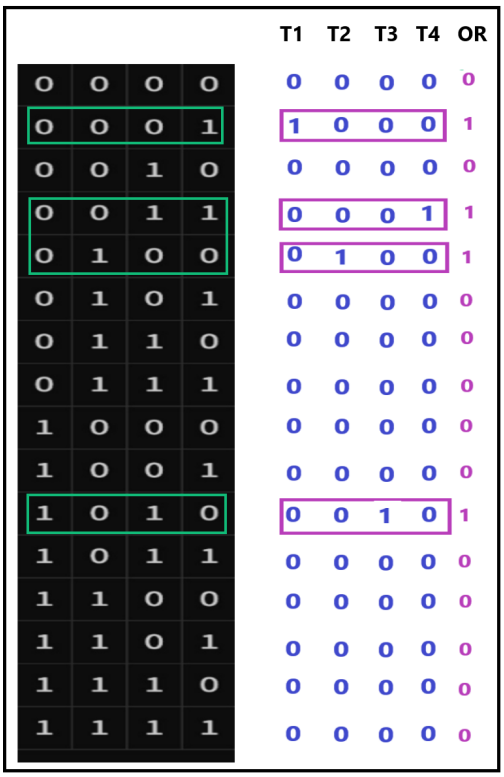
Figura 3: Tablas de bits

La idea de resolver las tablas de bits es luego en la función forward saber cuales son las entradas que nos tienen que dar mayores que 0,5 (y el resto menores que 0.5) de cada tupla. Esto es así porque una red neuronal escoge como ganador a uno de los valores que tengan como resultado mayor que 0.5. Con esto sabido, se ha obtenido (después de realizar el 'AND' en las 4 tuplas) la entrada que tiene que dar mayor que 0.5 de cada tupla. Con esto, las entradas que tienen que dar mayor que 0.5 de cada tupla son las siguientes:

- Tupla 1: '0001'.
- Tupla 2: '0100'.
- Tupla 3: '1010'.
- Tupla 4: '0011'.

Con lo anteriores se han obtenido las entradas deseadas que den mayores que 0.5 de la capa intermedia, pero faltaría la capa de salida (llamada f) que solo tiene 1 neurona. Las entradas de la neurona 'f' son las

salidas de las 4 tuplas, es decir, el resultado de aplicar el 'AND' a cada combinación de bits. Al ser 4 las entradas que le entran a 'f' entonces se formaría una nueva tabla de bits que donde cada posición es el resultado del 'AND' de las 4 tuplas.



				T1	T2	T3	T4	OR
0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	1
0	0	1	0	0	0	0	0	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	1	0	0	1
0	1	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	1	0	0	0	1	0	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0

Figura 4: Neurona f

Con lo cual, la neurona 'f' tiene que estar 'entrenada' para que estas entradas den como salida mayor que 0.5 (y el resto menor que 0.5):

- 0001
- 0011
- 0100
- 1010

A parte de lo anterior se usará la función sigmoidea para resolver el ejercicio planteado. La función sigmoidea básicamente 'curva' los resultados que nos dé el entrenamiento de la neurona entre 0 y 1 (es decir, cuanto más se acerque el resultado a '+infinito' más se curvará hacía 1 , y lo mismo con '-infinito').

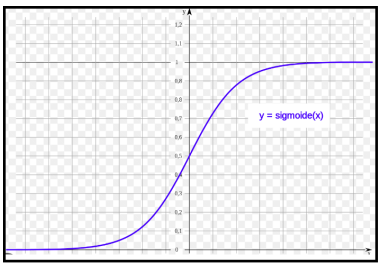


Figura 5: Sigmoidea

Explicado todo lo anteior procedo a crear la función forward (que contendrá la lógica de las neuronas) y la función sigmoidea (la sigmoidea es programar básicamente una función matemática).

Todo junto quedaría así:

```python
import math
import numpy as np

def sig(x):
    sig = 1/(1 + math.exp(-x))
    return sig

def forward(entrada):
    entrada_array = np.asarray(entrada)

    ############################[Capa AND]##############################
    T1 = sig(-0.5 + np.sum((entrada_array*np.asarray((-10, -10, -10, 9)))))
    T2 = sig(-3 + np.sum((entrada_array*np.asarray((-10, 9, -10, -10)))))
    T3 = sig(-13 + np.sum((entrada_array*np.asarray((9, -18, 9, -18)))))
    T4 = sig(-13 + np.sum((entrada_array*np.asarray((-18, -18, 9, 9)))))
    ############################[Capa OR y salida]##########################
    f = sig(-6 + np.sum(np.asarray((T1, T2, T3, T4))*np.asarray((10,10, 10, 10 ))))
    return f #valor entero
```

Figura 6: Ejercicio 1

El proceso que sigo para ajustar los pesos es el siguiente. Para ayudarme he creado una función que me crea las entradas de los bits en un bucle (va desde 0000 hasta 1111): La idea aquí es que tengo que ir ajustando de forma manual el bias y los pesos para obtener que la entrada deseada nos dé 0.5 y el resto menos que 0.5. Este proceso de ajustar manualmente los valores se llama 'entrenar'.

TUPLA: sig(bias + np.sum((entrada*np.assarray((pesos))))

```python
def evalua_ej1():
    for i in range (16):
        m = format(i, '04b')
        bits_entrada = (int(m[0]),int(m[1]),int(m[2]),int(m[3]))
        salida = forward(bits_entrada)
        print(salida)
```

Figura 7: Función auxiliar eval

Los valores 'correctos' del 'entrenamiento' son los que se pueden observar en la figura 6. Y los resultados de entrenar las 4 tuplas son los siguientes:

Figura 8: Resultados entrenamiento Tuplas

Una vez entrenadas las 4 tuplas procedí a entrenar la neurona 'f' que al final me dió los valores correctos con el siguiente bias y pesos:

Figura 9: Resultados entrenamiento f

1.2. Modelar, entrenar y probar la red en Keras

Tal como nos lo pide este apartado he creado una función evalua y la función de entrenamiento en keras.

La función evalua básicamente recorre en un bucle los bits desde 0000 hasta 1111 y se van pasando como entradas a la función forward. Cada vez que recorro una interación del bucle guardo esos bits en 1 array, y la salida con esos bits en otro array , de modo que la posición del array en la cual estén esos bits se corresponda a la posición del otro array en el cual están las salidas del forward.

La función keras en la parte del 'Sequential' le he puesto 3 capas, la primera que es la de los inputs que significa que tiene 4 entradas, la segunda como la capa intermedia que tendrá 4 neuronas, y la última la neurona 'f' (es decir, estoy recreando la imagen 1 que puse en el ejercicio 1).

```python
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow as tf

def evalua():
    X = [] #entradas
    Y = [] #salidas
    for i in range (16):
        m = format(i, '04b')
        bits_entrada = (int(m[0]),int(m[1]),int(m[2]),int(m[3]))
        X.append(bits_entrada)

        #Para el apartado 2, entranamiento de keras
        salida = forward(bits_entrada)
        if salida > 0.5: salida = 1
        else: salida = 0
        Y.append(salida)

    return X,Y

def mlp_keras():

    X,Y = evalua()

    model = keras.Sequential([ #Creación de la estructura

        #Primera capa, 4 entradas (a,b,c,d)
        keras.Input(shape=(4)),

        #Segunda capa, 4 neuronas (Cada 'Tx' en la función forward) AND
        layers.Dense(4,activation="sigmoid"),

        #Tercera capa, 1 neurona (la 'f' en la función forward) OR
        layers.Dense(1, activation="sigmoid"),

    ])


    model.compile(loss="mean_squared_error", optimizer="adam", metrics=["accuracy"])

    h = model.fit(X, Y, epochs=8000, batch_size=32)
    Y_pred = model.predict(X)

    print(Y_pred)

    print(f"{model.summary()}")

    for i , e in enumerate(X):
        print(X[i] , Y[i] , Y_pred[i] )
```

Figura 10: Función evalua y keras

1.3. Analizar el entrenamiento y comparar con la red ajustada a mano

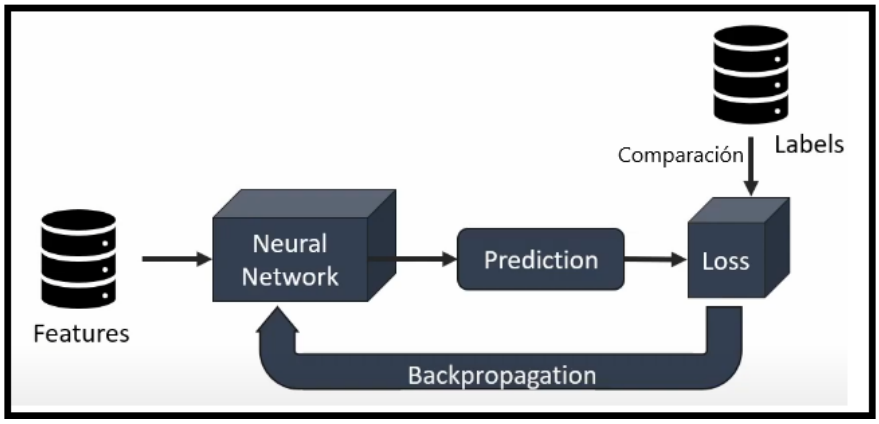


Figura 11: Representación visual keras

Definiciones que piden las preguntas de este apartado:

- Un 'epoch' es finalizado cuando la red 'ha visto' todo el dataset entero. El entrenamiento se realiza para varios epoch (8000 en mi caso) y este número nunca es fijo ya que depende de de muchos factores como la red a entrenar, la complejidad del dataset, etc. Los epochs varian mucho en tiempo a realizar, puede variar desde segundos hasta horas (depende de los factores). También se define como: 1 paso forward y un paso backward de todas las muestras.
- El 'batch size' define el número de muestras que serán propagadas en la red. Por ejemplo, si tengo 1050 muestras de entreno y pongo un batch size de 100 , el algoritmo coge 100 muestras de las 1050 que puse y entrena la red, luego coge otras 100 muestras (diferentes a las 100 primeras) y entrena la segunda ronda, así con todas las muestras de esas 1050. Cuanto menor sea el batch size menos preciso será el sistema, por lo que se recomienda alrededor de 32 (o umpoco más). Cuanto mayor sea el batch size mayor será la memoria ram que se necesite consumir del ordenador.
- El 'optimizer' se refiere al algoritmo que se usa para la búsqueda del modelo óptimo de parámetros que minimizan el loss. Hay diferentes optimizadores diferentes en keras como: 'SGD', 'Adam' o 'RMSprop'. Cada optimizador tiene diferentes características que se ajustan a los diferentes problemas.
- El parámetro 'loss' en nuestra función utilizamos 'mean squared error' o 'MSE' que es una función que se usa para mesusrar las diferencias entre el resultado predicho en el modelo y el verdadero resultado. El 'MSE' es usualmente usando para la regresión de problemas donde la finalizad es predecir un resultado continuo y se define como: MSE: 1/n * sum((y true - y pred) elevado 2) , donde y 'y true' es el resultado y 'y pred' es el resultado predicho, y 'n' es el número de muestras del batch size.
- El 'loss rate' se puede ajustar en keras cambiando el valore de 'learning rate' cuando definimos el optimizer de mi modelo. El 'learning rate' determina el tamaño del paso sobre el cual el optimizador hará las actualizaciones a los parametros del modelo con la función de minimizar la función loss. Por ejemplo si usamos el optimizador 'Adam' puedo ajustar el learning rate pasándo como parametro a 'Adam' el learning rate optimizer = Adam(learning rate=0.001), en este ejemplo el learning rate es 0.001 lo que significa que el optimizador hará pequeñas actualizaciones

En cuanto a los pasos que necesita el keras para aprender mi función, he estado probando diferentes epoch y batch size, llegando a la conclusión de que con 8000 epoch y un batch size de 32 consigo los resultados que concuerdan con lo que estoy buscando, es decir, que dan mayor de 0.5 donde lo requiere, y menor de 0.5 donde lo require. Cabe destacar que aun con 8000 epochs hay veces (y otras veces que sí da lo correcto) donde una de las entradas me dá de forma incorrecta, por lo que aún hay posibilidad de aumentar el tamaño de entrenamiento.

```
Epoch 7995/8000
1/1 [==============================] - 0s 2ms/step - loss: 0.0074 - accuracy: 1.0000
Epoch 7996/8000
1/1 [==============================] - 0s 5ms/step - loss: 0.0074 - accuracy: 1.0000
Epoch 7997/8000
1/1 [==============================] - 0s 3ms/step - loss: 0.0074 - accuracy: 1.0000
Epoch 7998/8000
1/1 [==============================] - 0s 3ms/step - loss: 0.0074 - accuracy: 1.0000
Epoch 7999/8000
1/1 [==============================] - 0s 3ms/step - loss: 0.0074 - accuracy: 1.0000
Epoch 8000/8000
1/1 [==============================] - 0s 2ms/step - loss: 0.0074 - accuracy: 1.0000
1/1 [==============================] - 0s 69ms/step
[[0.07649834]
 [0.9413187 ]
 [0.06208495]
 [0.9520224 ]
 [0.8219603 ]
 [0.10680413]
 [0.07639239]
 [0.01638545]
 [0.0790318 ]
 [0.04144926]
 [0.8437867 ]
 [0.08883741]
 [0.07213335]
 [0.03459288]
 [0.06876785]
 [0.04110664]]
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 20

 dense_1 (Dense)             (None, 1)                 5

=================================================================
Total params: 25
Trainable params: 25
Non-trainable params: 0
```

Figura 12: Resultados keras

He juntado los arrays de entradas y salidas, y la salida del keras para poder visualizarlo mejor:

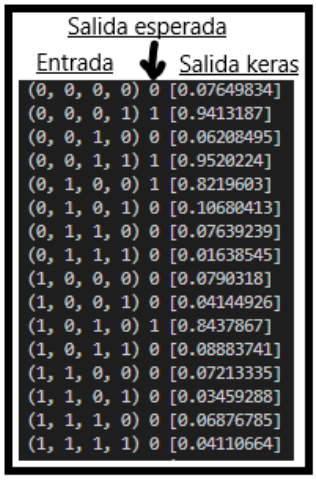


Figura 13: Resultados keras comparativa

En cuanto a la comparación de bias y pesos he utilizado la siguiente instrucción:

Figura 14: Comparación biases y pesos

Tal como se observa en la imagen, he utilizado valores más altos (o más bajos) para los pesos y biases que keras. Coincidimos en algunos que usamos negativo ambos. Esto es así porque keras hace un procesado más metódico y rápido que yo procesandolo de poco a poco y lento.

2. Parte 2. Entrena un MLP mediante Deep Learning usando Keras

2.1. Procesamiento de los datos

MNIST es un dataset de manuscritos escritos a mano y se usa para el entrenamiento de reconocimiento de imagenes. Éste se ha convertido en un benchmark bastante popular para evaluar el comportamiento de los modelos de reconocimiento de imagenes. El dataset contiene 60.000 imagenes de entrenamiento y 10000 de test, donde cada imagen es de 28 pixel por 28 pixel.

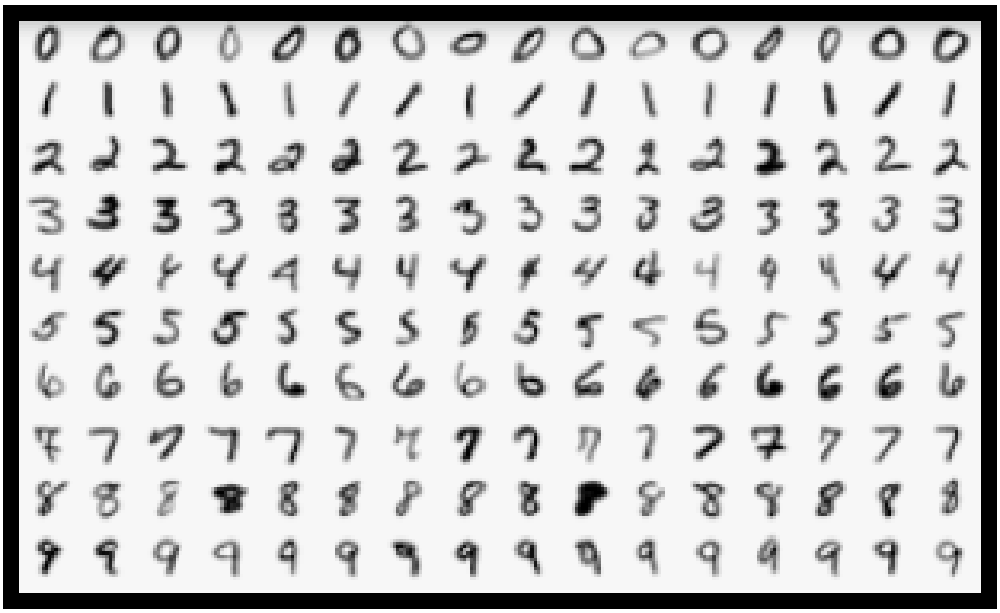


Figura 15: Mnist

2.1.1. ¿Cuántas neuronas necesitamos en la capa de entrada?¿Y en la capa de salida?

Dado que son imagenes de 28 pixel por 28 pixel, la capa de entrada contendría 784 neuronas. Como lo que queremos obtener es un valor entre el 0 y el 9 la capa de salida contendría 10 neuronas.

2.1.2. ¿Cómo dividirás el conjunto de entrenamiento/test/validación?

En esta parte del ejercicio no se ha dividido el conjunto de datos en conjuntos de entrenamiento, test y validación. El conjunto de datos MNIST ya viene dividido en dos conjuntos: el conjunto de entrenamiento ('Xtrain' y 'Ytrain') y el conjunto de test ('Xtest' y 'Ytest')

El conjunto de entrenamiento se utiliza para entrenar la red neuronal, mientras que el conjunto de test se utiliza para evaluar cómo el modelo se desempeña en datos que no ha visto durante el entrenamiento. Es importante tener un conjunto de datos de test para poder evaluar el rendimiento del modelo de manera objetiva y verificar si está sobreajustado (es decir, si ha aprendido demasiado bien los patrones del conjunto de entrenamiento y no generaliza bien a datos nuevos).

```
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Figura 16: División de datos

Para dividir el conjunto de entrenamiento en dos conjuntos más pequeños (uno para entrenar y otro para validar), puedes utilizar la función 'train test split' de scikit-learn. Por ejemplo:

```
from sklearn.model_selection import train_test_split

# Dividir el conjunto de entrenamiento en dos conjuntos: entrenamiento y validación
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2)
```

En este caso, se ha dividido el conjunto de entrenamiento original en dos conjuntos, uno con el 80 por ciento de los datos (que se utilizará para entrenar la red) y otro con el 20 por ciento restante (que se utilizará para validar el modelo durante el entrenamiento).

Es importante tener en cuenta que el conjunto de validación no se debe utilizar para evaluar el rendimiento del modelo final, ya que el modelo ha visto estos datos durante el entrenamiento. En su lugar, se debe utilizar el conjunto de test para evaluar el rendimiento del modelo final. La validación se utiliza principalmente para ajustar los hiperparámetros del modelo durante el entrenamiento, como el tamaño del batch o el número de épocas.

2.1.3. ¿Cómo preprocesarás los datos de entrada?

Los datos de entrada (`x_train` y `y_train`) se procesan de la siguiente manera:

1. Se cargan los datos de entrenamiento y test del conjunto de datos MNIST utilizando la función `load_data` de Keras. Esta función devuelve dos tuplas, una con los datos de entrenamiento y otra con los datos de test.
2. Se normalizan los datos de entrada dividiéndolos por 255. Esto se hace para que todos los valores de los pixeles estén entre 0 y 1. Los datos de entrada del conjunto de datos MNIST consisten en imágenes de 28x28 pixeles en escala de grises, con valores de pixel entre 0 y 255. Al dividir los valores de pixel por 255, se asegura que todos los valores estén entre 0 y 1.
3. No se realiza ningún otro procesamiento adicional de los datos de entrada. En algunos casos, puede ser necesario aplicar otras técnicas de preprocesamiento, como redimensionar las imágenes a un tamaño específico o aplicar filtros para mejorar la calidad de las imágenes. Sin embargo, en este caso no es necesario realizar ningún otro procesamiento adicional.

Una vez procesados, los datos de entrada se utilizan para entrenar la red neuronal mediante la función fit de Keras. Luego, se evalúa el rendimiento del modelo en el conjunto de datos de test utilizando la función evaluate.

2.1.4. ¿Cómo transformarás la imagen para poder entrenarla con una red MLP?

(Respondido en la pregunta anterior.)

2.2. Implementa la red en keras

```python
def mnist():

    # Load the MNIST dataset
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    # Normalizar datos para que estén en un rango entre 0 y 1
    x_train = x_train.reshape(60000, 784)
    x_test = x_test.reshape(10000, 784)

    x_train = x_train.astype('float32') / 255
    x_test = x_test.astype('float32') / 255
    y_train = keras.utils.to_categorical(y_train, 10)
    y_test = keras.utils.to_categorical(y_test, 10)

    model = keras.Sequential([

        #ENTRADAS
        keras.Input(shape=( 784,)),

        #CAPAS INTERMEDIAS
        layers.Dense(128,activation="relu"),
        layers.Dense(128,activation="relu"),

        #SALIDAS
        layers.Dense(10, activation="softmax"),

    ])

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

    # Train the model
    model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1)

    # Evaluate the model on the test data
    test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test loss: {test_loss:.3f}, Test accuracy: {test_accuracy:.3f}')

    #Saco el test error
    test_error = 1 - test_accuracy
    print(f"Test_error: {test_error}")


    #1000 elementos al azar del conjunto de entrenamiento
    X_train_subset = x_train[:1000]
    y_train_subset = y_train[:1000]
    test_loss_train, test_acc_train = model.evaluate(X_train_subset, y_train_subset, verbose=0)
    print('(1000 de entrenamiento) Test loss:', test_loss_train)
    print('(1000 de entrenamiento) Test accuracy:', test_acc_train)

    #1000 elementos al azar del conjunto de test
    X_test_subset = x_test[:1000]
    y_test_subset = y_test[:1000]
    test_loss_test, test_acc_test = model.evaluate(X_test_subset, y_test_subset, verbose=0)
    print('(1000 de test) Test loss:', test_loss_test)
    print('(1000 de test) Test accuracy:', test_acc_test)

    print(f"{model.summary()}")
```

Figura 17: Red en Keras

2.2.1. ¿Qué es la función de activación relu? ¿Cómo varia de la sigmoidea vista en teoría?

Relu (Rectified Linear Unit) es una función de activación no lineal que se utiliza comúnmente en redes neuronales y para cada entrada x, relu devuelve x si x es mayor que 0, y 0 en caso contrario. Su función matemática es la siguiente, `f(x) = max(0,x)`

Relu es una función de activación muy utilizada en redes neuronales debido a su simplicidad y eficiencia. Tiene varias ventajas en comparación con otras funciones de activación:

- Es muy rápida de evaluar, ya que solo requiere comparar un valor con 0 y devolver el mayor.
- No tiene problemas de vanishing gradient, lo que permite que la red neuronal aprenda más rápidamente.
- Introduce una no linealidad en la red neuronal, lo que permite que pueda aprender patrones más complejos.

Sin embargo, relu también tiene algunas desventajas, como el hecho de que puede "morir"(es decir, dejar de ser útil) si se inicializa mal o si se utiliza una tasa de aprendizaje demasiado alta. Esto se debe a que la función relu devuelve siempre 0 para valores negativos, lo que significa que las neuronas que utilizan esta función de activación no pueden aprender si sus pesos se actualizan de forma negativa.

La función sigmoidea y la función relu son dos funciones de activación no lineales que seutilizan comúnmente en redes neuronales. Cada una de ellas tiene sus propias características y se utilizan en diferentes contextos según las necesidades del modelo.

Una de las principales diferencias entre la función sigmoidea y la función relu es la forma en que varían sus salidas en función de la entrada. La función sigmoidea es una función sigmoidal, lo que significa que tiene una forma en S. Su salida varía suavemente desde 0 hasta 1 a medida que la entrada aumenta desde -infinito hasta +infinito. Esto hace que la función sigmoidea sea una buena opción para problemas de clasificación binaria, ya que permite que la salida se ajuste gradualmente entre 0 y 1.

Por otro lado, la función relu es una función lineal para entradas positivas y una función constante para entradas negativas. Su salida es siempre 0 para entradas negativas y aumenta de forma lineal con la entrada para entradas positivas. Esto hace que la función relu sea más eficiente y rápida de evaluar que la función sigmoidea.

2.2.2. ¿Qué consigue la función de activación softmax de la última capa? ¿Se podría usar una función relu en la última capa?

La función de activación softmax es comúnmente utilizada en la última capa de una red neuronal cuando se desea predecir la probabilidad de una clase o etiqueta particular en un problema de clasificación de múltiples clases.

La función softmax toma como entrada un vector de valores reales y devuelve un vector de valores reales que suman 1 y que pueden interpretarse como probabilidades. Cada elemento del vector de salida es una probabilidad de la clase correspondiente. Por ejemplo, si la salida de la red es [0.1, 0.3, 0.6], entonces se puede interpretar que hay una probabilidad del 10 porciento de que la entrada pertenezca a la clase 0, una probabilidad del 30 porciento de que pertenezca a la clase 1 y una probabilidad del 60 porciento de que pertenezca a la clase 2.

La función softmax es útil en problemas de clasificación de múltiples clases ya que proporciona una forma de convertir las salidas de la red en probabilidades y tomar decisiones basadas en estas probabilidades. También es comúnmente utilizada en conjunción con una función de pérdida llamada çategorical cross-entropy", que mide la distancia entre las etiquetas verdaderas y las predicciones de la red.

Es posible utilizar la función de activación relu en la última capa de una red neuronal en Keras, pero no es la elección más común en esta posición.

La función relu es una función de activación muy utilizada en redes neuronales ya que tiene una derivada constante y es fácil de computar. Sin embargo, suele utilizarse en capas ocultas y no en la última capa de la red. Esto se debe a que la función relu produce valores de salida solo para entradas positivas y, por lo tanto, no es adecuada para problemas de clasificación en los que se esperan valores de salida continuos y no solo valores binarios (0 o 1).

En su lugar, en la última capa de una red neuronal se suele utilizar una función de activación que permita obtener valores de salida continuos y que sea fácil de interpretar como probabilidades. Una opción común es la función softmax, que se mencionó anteriormente.

2.2.3. ¿Qué función de activación crees que es mejor? ¿Por qué crees que se suele utilizar más relu que su alternativa sigmoidea?

En general, no existe una función de activación "mejor"que se aplique a todos los problemas de aprendizaje automático. La elección de la función de activación adecuada depende del tipo de problema y de la arquitectura de la red neuronal.

Sin embargo, algunas funciones de activación son más comunes que otras debido a su simplicidad y buen rendimiento en la mayoría de los casos. A continuación se mencionan algunas de las funciones de activación más comunes utilizadas en Keras

- Relu (Rectified Linear Unit): es una función de activación lineal que devuelve 0 para entradas negativas y el valor de la entrada para entradas positivas. Es muy utilizada en las capas ocultas de redes neuronales debido a su sencillez y buen rendimiento.
- Sigmoid: es una función de activación sigmoide que devuelve valores entre 0 y 1. Es adecuada para problemas de clasificación binaria, ya que devuelve valores cercanos a 0 o 1 según la clase a la que pertenezca la entrada. Sin embargo, tiene el inconveniente de que puede llevar a valores muy pequeños o muy grandes cuando la entrada es muy negativa o muy positiva, lo que puede dificultar el entrenamiento de la red.
- Tanh: es una función de activación tangencial que devuelve valores entre -1 y 1. Al igual que la función sigmoid, es adecuada para problemas de clasificación binaria y tiene el inconveniente de que puede llevar a valores muy pequeños o muy grandes cuando la entrada es muy negativa o muy positiva.
- Softmax: es una función de activación que se utiliza comúnmente en la última capa de una red neuronal cuando se desea predecir la probabilidad de una clase o etiqueta particular en un problema de clasificación de múltiples clases. Devuelve un vector de valores reales que suman 1 y que pueden interpretarse como probabilidades.

En general, la función relu se utiliza más que la función sigmoide debido a su sencillez y buen rendimiento en la mayoría de los casos. Además, la función relu es más rápida de computar que la función sigmoide y tiene una derivada constante, lo que facilita el entrenamiento de la red.

2.2.4. Explica qué son y para qué sirven los parámetros batch size y validation split.

Los parámetros batch size y validation split son dos parámetros muy comunes en el entrenamiento de redes neuronales y se utilizan para controlar cómo se dividen los datos y cómo se procesan durante el entrenamiento.

- Batch size: El parámetro batch size indica el número de muestras que se utilizan en una iteración del proceso de entrenamiento. Por ejemplo, si el batch size es 10 y se tienen 1000 muestras de entrenamiento, entonces el proceso de entrenamiento se dividirá en 100 iteraciones (1000/10=100), cada una de las cuales utiliza 10 muestras para actualizar los pesos de la red. El parámetro batch size puede afectar la precisión y el tiempo de entrenamiento de la red. Un batch size muy pequeño puede resultar en un entrenamiento más preciso, pero también puede tomar más tiempo debido a que se necesitan más iteraciones para procesar todas las muestras. Un batch size muy grande puede resultar en un entrenamiento más rápido, pero también puede ser menos preciso debido a que se están utilizando menos muestras para actualizar los pesos de la red.
- Validation split: El parámetro validation split indica el porcentaje de muestras de entrenamiento que se deben utilizar para validación. Durante el proceso de entrenamiento, se utilizan las muestras de validación para evaluar el rendimiento de la red en datos que no se han utilizado para entrenarla. Esto es útil para detectar el sobreajuste (overfitting), es decir, cuando la red se ajusta demasiado a

los datos de entrenamiento y no generaliza bien a datos nuevos. Por ejemplo, si el validation split es 0.2 y se tienen 1000 muestras de entrenamiento, entonces se reservarán 200 muestras para validación y se utilizarán 800 muestras para entrenar la red. Durante el proceso de entrenamiento, se evaluará el rendimiento de la red en las muestras de validación y se utilizarán para hacer ajustes en la red si es necesario.

Es importante elegir adecuadamente el valor del batch size y del validation split para asegurar que la red se esté entrenando de manera óptima. Un batch size demasiado grande puede resultar en un entrenamiento menos preciso y un validation split demasiado pequeño puede resultar en una evaluación del rendimiento menos precisa. Por lo tanto, es importante hacer algunos experimentos para encontrar los valores óptimos para tu problema.

2.2.5. ¿Conoces algún otro algoritmo de optimización distinto adam? Comenta brevemente el funcionamiento de otro distinto.

Sí, hay varios algoritmos de optimización disponibles en el campo del aprendizaje automático, además del algoritmo Adam. Algunos ejemplos de algoritmos de optimización populares son:

- Gradiente Descendente Estocástico (SGD, Stochastic Gradient Descent): es un algoritmo de optimización simple que actualiza los pesos de la red a medida que avanza en el proceso de entrenamiento. El SGD utiliza una tasa de aprendizaje fija y no tiene en cuenta el momento o la historia de los pesos de la red.
- Gradiente Descendente con Momento (Momentum): es una variante del SGD que utiliza una tasa de aprendizaje y una cantidad de momento. El momento se utiliza para dar mayor importancia a las actualizaciones de peso más recientes y puede ayudar a evitar que el algoritmo se quede atrapado en un mínimo local.
- RMSProp: es un algoritmo de optimización que utiliza una tasa de aprendizaje adaptativa y que se centra en actualizar los pesos de la red de forma más efectiva cuando la curvatura de la función de pérdida es más plana.
- Adagrad: es un algoritmo de optimización que utiliza una tasa de aprendizaje adaptativa y que reduce la tasa de aprendizaje a medida que avanza el proceso de entrenamiento. Esto ayuda a evitar que el algoritmo se quede atrapado en un mínimo local.

Cada uno de estos algoritmos tiene sus propias ventajas y desventajas y puede ser adecuado para diferentes tipos de problemas. Es importante elegir el algoritmo de optimización adecuado para tu problema y hacer algunos experimentos para encontrar los valores óptimos de los parámetros de entrada.

2.3. Prueba el modelo

```
422/422 [==============================] - 1s 2ms/step - loss: 0.0174 - accuracy: 0.9946 - val_loss: 0.0921 - val_accuracy: 0.9783
Epoch 12/15
422/422 [==============================] - 1s 2ms/step - loss: 0.0128 - accuracy: 0.9959 - val_loss: 0.0796 - val_accuracy: 0.9813
Epoch 13/15
422/422 [==============================] - 1s 2ms/step - loss: 0.0116 - accuracy: 0.9964 - val_loss: 0.0863 - val_accuracy: 0.9798
Epoch 14/15
422/422 [==============================] - 1s 2ms/step - loss: 0.0103 - accuracy: 0.9968 - val_loss: 0.1072 - val_accuracy: 0.9772
Epoch 15/15
422/422 [==============================] - 1s 2ms/step - loss: 0.0126 - accuracy: 0.9959 - val_loss: 0.1068 - val_accuracy: 0.9778
Test loss: 0.093, Test accuracy: 0.976
Test_error: 0.023899972438812256
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 128)               16512

 dense_2 (Dense)             (None, 10)                1290

=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
```

Figura 18: Resultados

2.3.1. ¿Qué error de clasificación obtienes para los datos de test? ¿Y qué valor de pérdida de la función de coste? Para este apartado gasta la función de keras.

El error de clasificación que obtengo está cercano al 0.023 y el valor de pérdida obtengo 0.093

2.3.2. Escoge 1000 elementos al azar del conjunto de entrenamiento y comprueba qué tasa de error de clasificación obtienes. ¿Coincide con la métrica anterior?

Sí que casi coincide con la métrica anterior.

```python
#1000 elementos al azar del conjunto de entrenamiento
X_train_subset = x_train[:1000]
y_train_subset = y_train[:1000]
test_loss_train, test_acc_train = model.evaluate(X_train_subset, y_train_subset, verbose=0)
print('(1000 de entrenamiento) Test loss:', test_loss_train)
print('(1000 de entrenamiento) Test accuracy:', test_acc_train)
```

```
(1000 de entrenamiento) Test loss: 0.011255544610321522
(1000 de entrenamiento) Test accuracy: 0.9959999918937683
```

Figura 19: Resultados 1k entrenamiento

2.3.3. Escoge 1000 elementos al azar del conjunto de test y comprueba qué tasa de error de clasificación obtienes. ¿Coincide con la métrica anterior?

Sí que casi coincide con la métrica anterior.

```
#1000 elementos al azar del conjunto de test
X_test_subset = x_test[:1000]
y_test_subset = y_test[:1000]
test_loss_test, test_acc_test = model.evaluate(X_test_subset, y_test_subset, verbose=0)
print('(1000 de test) Test loss:', test_loss_test)
print('(1000 de test) Test accuracy:', test_acc_test)
```

```
(1000 de test) Test loss: 0.08021081984043121
(1000 de test) Test accuracy: 0.9769999980926514
```

Figura 20: Resultados 1k test

2.4. Mejora la red (opcional)

Hay algunas maneras en las que puedes intentar mejorar el rendimiento de la red neuronal en el conjunto de datos MNIST:

- Usar más capas ocultas: Agregar más capas ocultas a la red puede ayudarla a aprender patrones más complejos en los datos, pero también aumenta el riesgo de sobreajuste. Se pueden probar agregando más capas ocultas y ver si mejora el rendimiento de la red.
- Usar más neuronas en las capas ocultas: Aumentar el número de neuronas en las capas ocultas también puede ayudar a la red a aprender patrones más complejos, pero nuevamente aumenta el riesgo de sobreajuste. Se pueden probar aumentando el número de neuronas en las capas ocultas y ver si mejora el rendimiento de la red.
- Usar técnicas de regularización: Las técnicas de regularización pueden ayudar a reducir el riesgo de sobreajuste penalizando al modelo por tener demasiados parámetros. Se pueden probar agregando una capa de dropout después de cada capa oculta con una tasa de dropout de 0.5 a 0.8 para ver si ayuda a mejorar el rendimiento de la red.
- Usar un algoritmo de optimización diferente: El algoritmo de optimización Adam es una buena opción por defecto, pero se pueden probar usar un algoritmo de optimización diferente como RMSprop o SGD y ver si mejora el rendimiento de la red.
- Usar una función de activación diferente: La función de activación ReLU es una buena opción por defecto, pero se puede probar usar una función de activación diferente como sigmoid o tanh y ver si mejora el rendimiento de la red.

3. Bibliografía

https://www.youtube.com/watch?v=tUoUdOdTkRw&t=667s (Perceptrón)
https://chat.openai.com/chat (ChatGPT)
https://www.youtube.com/watch?v=ZyauOVzjg9Q (Keras)
https://www.youtube.com/watch?v=CU24iC3grq8 (Keras)