



Universidad Autónoma de San Luis Potosí  
Facultad de ingeniería  
Inteligencia Artificial Aplicada  
**Practica 6**  
**Implementación de una red neuronal**  
**en sistemas embebidos**  
**Ana Sofía Medina Martínez**



**Fecha 3/10/2024**

## Objetivo

El estudiante aprenderá a construir, compilar y entrenar una red neuronal utilizando la librería Keras para resolver problemas de clasificación y regresión y que aprenda a importar modelos de redes neuronales en sistemas embebidos.

## Procedimiento

6.1.- Sigue las instrucciones del archivo “practica\_6\_training” para desarrollar y entrenar un modelo de red neuronal de clasificación.

6.2.- Sigue las instrucciones del archivo “practica\_6\_inferencia” para implementar una red neuronal en un microcontrolador.

## Resultados

### 1.1 Importar los datos

Vamos a importar los datos que adquirimos en la práctica anterior utilizando la librería pandas.

```
[1] import pandas as pd

data = pd.read_csv("../data/datos_mpu6850.csv") #TODO: Cargar el dataset usando pandas

#Mezclamos los datos para que no haya sesgo
data = data.sample(frac=1)

data.head()
```

	accelX	accelY	accelZ	output
1506	9.03	0.40	5.71	DOWN
396	2.22	9.53	-0.24	RIGHT
596	1.32	9.64	0.46	RIGHT
624	1.08	9.70	0.11	RIGHT
655	1.04	9.30	2.92	RIGHT

Next steps: [Generate code with data](#) [View recommended plots](#) [New interactive sheet](#)

### 1.2 Preprocesar los datos

Vamos a realizar los siguientes pasos para preprocesar los datos:

- Separar los datos en entrada y salida.
- Codificar las etiquetas de salida aplicando one-hot encoding.
- Separar los datos en entrenamiento y prueba.

```
[2] #TODO: Crear un dataframe con las columnas accelX, accelY y accelZ, guardarlo en la variable X
X = data[["accelX", "accelY", "accelZ"]]
X.head()
```

[2]

	acelX	acelY	acelZ
1506	9.03	0.40	5.71
396	2.22	9.53	-0.24
596	1.32	9.64	0.46
624	1.06	9.70	0.11
655	1.04	9.30	2.92

Next steps: [Generate code with X](#) [View recommended plots](#) [New interactive sheet](#)

#TODO: Crear un dataframe con la columna output, guardalo en la variable y  
y = data["output"]  
y.head()

[3]

	output
1506	DOWN
396	RIGHT
596	RIGHT
624	RIGHT
655	RIGHT

dtype: object

+ Code + Text

Para codificar las etiquetas de salida vamos a utilizar la función `get_dummies` de pandas.

```
import pandas as pd

encoded_labels = pd.get_dummies(labels, dtype=dtype)
```

- `labels`: es un arreglo de numpy o una serie de pandas con las etiquetas de salida.
- `dtype`: es el tipo de dato de las columnas de la matriz de salida (int, float, etc).

#TODO: Convertir la columna y en un one-hot encoding usando `pd.get_dummies`, recuerda especificar el tipo de dato  
y = pd.get\_dummies(y, dtype=float)  
y.head()

[4]

	DOWN	IDLE	LEFT	RIGHT	TOP
1506	1.0	0.0	0.0	0.0	0.0
396	0.0	0.0	0.0	1.0	0.0
596	0.0	0.0	0.0	1.0	0.0
624	0.0	0.0	0.0	1.0	0.0
655	0.0	0.0	0.0	1.0	0.0

Next steps: [Generate code with y](#) [View recommended plots](#) [New interactive sheet](#)

+ Code + Text

Un paso muy importante es separar los datos en entrenamiento y prueba, esto nos permitirá evaluar el modelo en datos que no ha visto durante el entrenamiento.

Realizaremos la separación utilizando la función `sample` de pandas.

```
train_data = data.sample(frac=0.8)
test_data = data.drop(train_data.index)
```

- `frac`: es el porcentaje de datos que se utilizarán para entrenamiento. En este caso, el 80% de los datos se utilizarán para entrenamiento.
- `drop`: Elimina las filas con los índices especificados, en este caso, eliminamos las filas que se utilizaron para entrenamiento.

[5] #TODO: Dividir los datos en entrenamiento y prueba usando `X.sample` y `y.sample`  
# Guarda los datos de entrenamiento en las variables `X_train` y `y_train`  
# Guarda los datos de prueba en las variables `X_test` y `y_test`  
`X_train = X.sample(frac=0.8)`  
`y_train = y.sample(frac=0.8)`  
`X_test = X.drop(X_train.index)`  
`y_test = y.drop(y_train.index)`

[6] 

```
mean = X_train.mean()
std = X_train.std()

print(f"MEAN: {mean.values}")
print(f"STD: {std.values}")
```

MEAN: [ 1.47194548 -0.1952974 3.06564436]  
STD: [ 6.01119322 5.59077888 4.29326891]

Los valores obtenidos en la celda anterior se utilizaran para normalizar los datos en la ESP32.

# `X_train = (X_train - mean) / std`  
`X_test = (X_test - mean) / std`  
`X_train.head()`

[7]

	acelX	acelY	acelZ
1778	1.341725	0.204855	-1.550254
188	-0.010302	0.093551	1.629148
467	-0.100104	1.594843	0.322449
803	-1.808015	-0.185073	-0.842167
1716	1.389953	-0.160032	-1.440780

Next steps: [Generate code with X\\_train](#) [View recommended plots](#) [New interactive sheet](#)

+ Code + Text

Keras ya proporciona una capa de normalización que se puede utilizar en la red neuronal, no la utilizamos en este caso por que TF Lite Micro aun no soporta esta capa de Keras.

+ Code + Text

Hasta este punto, hemos realizado todos los pasos necesarios para preprocesar los datos, pero tenemos los datos en un formato de pandas, necesitamos convertirlos a un formato que pueda ser utilizado por Keras.

Vamos a convertir los datos a un arreglo de numpy utilizando el atributo `values`.

```
train_data = train_data.values
test_data = test_data.values
```

[8] #TODO: Convertir los datos de entrenamiento y prueba a un arreglo de numpy usando `.values`  
`X_train = X_train.values`  
`X_test = X_test.values`  
`y_train = y_train.values`  
`y_test = y_test.values`

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

#TOD00: Crear un modelo secuencial
model = Sequential()

#TOD00: Agregar una capa de entrada con 3 neuronas, una para cada columna de X
model.add(Input(shape=(3,)))

#TOD00: Agregar una capa densa con 8 neuronas y activación relu
model.add(Dense(8, activation='relu'))
model.add(Dense(8, activation='relu'))

#TOD00: Agregar una capa densa con 5 neuronas y activación softmax, 5 neuronas porque tenemos 5 clases en la salida
model.add(Dense(5, activation='softmax'))

model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 8)	32
dense_3 (Dense)	(None, 8)	72
dense_4 (Dense)	(None, 5)	45

Total params: 149 (596.00 B)  
Trainable params: 149 (596.00 B)  
Non-trainable params: 0 (0.00 B)

Después de definir la arquitectura de la red, vamos a compilar el modelo.

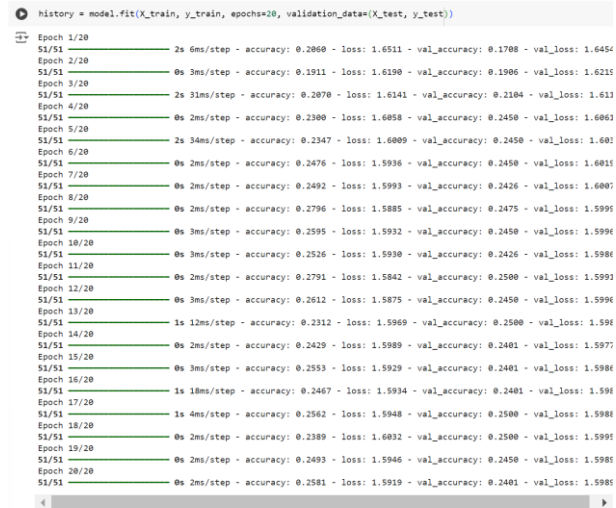
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

- optimizer: Optimizador.
- loss: Función de pérdida, en este caso, utilizamos la entropía cruzada categórica porque estamos realizando una clasificación multiclase.
- metrics: Métricas que se utilizarán para evaluar el modelo.

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

Por último, vamos a entrenar el modelo.

```
model.fit(train_data, train_labels, epochs=20, validation_data=(test_data, test_labels))
```



Si el modelo se entrena correctamente debes de obtener un valor de pérdida bajo y un valor de precisión alto, además de que la precisión en los datos de validación debe ser similar a la precisión en los datos de entrenamiento.

### 3. Convertir el modelo a TensorFlow Lite

Una vez que hemos entrenado el modelo, vamos a convertirlo a TensorFlow Lite para poder ejecutarlo en la ESP32.

Para convertir el modelo a TensorFlow Lite, vamos a utilizar la función `TFLiteConverter` de TensorFlow.

```

from tensorflow import lite as tflite

model_name = "mpu6050_model" #Nombre del archivo donde se guardará el modelo

converter = tflite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert() #Convertimos el modelo a un modelo tflite

with open(f"{model_name}.tflite", 'wb') as f: #Abrimos un archivo en modo escritura binaria
    f.write(tflite_model) #Guardamos el modelo en un archivo llamado model.tflite

```

Saved artifact at '/tmp/tmpf95mcgdg'. The following endpoints are available:

```

+ Endpoint 'serve'
  args_0 (POSITIONAL_ONLY): TensorSpec(shape=(None, 3), dtype=tf.float32, name='keras_tensor_4')
  Output Type:
    TensorSpec(shape=(None, 5), dtype=tf.float32, name=None)
  Captures:
    135366646408176: TensorSpec(shape=(), dtype=tf.resource, name=None)
    135366646415216: TensorSpec(shape=(), dtype=tf.resource, name=None)
    135366636488096: TensorSpec(shape=(), dtype=tf.resource, name=None)
    135366636476144: TensorSpec(shape=(), dtype=tf.resource, name=None)
    135366636484416: TensorSpec(shape=(), dtype=tf.resource, name=None)
    135366638526968: TensorSpec(shape=(), dtype=tf.resource, name=None)

```

Muchas plataformas de microcontroladores no tienen soporte para TensorFlow Lite. La forma más sencilla de ejecutar un modelo de TensorFlow Lite en un microcontrolador es convertirlo a una matriz de bytes y ejecutarlo en el microcontrolador.

```

17] def tflite_to_array(model_data, model_name):
    c_str = ""

    #Creamos las cabeceras del archivo
    c_str += f"#ifndef {model_name.upper()}_H\n"
    c_str += f"#define {model_name.upper()}_H\n\n"

    #Agregamos una variable con el tamaño del modelo
    c_str += f"const unsigned int {model_name}_len = {len(model_data)};\n\n"

    #Agregamos el modelo como un arreglo de bytes
    c_str += f"const unsigned char {model_name}[] = {{\n"

    for i, byte in enumerate(model_data):
        c_str += f"0x{byte:02X},\n"
        if (i + 1) % 12 == 0:
            c_str += "\n"

    c_str += "};\n\n"

    #Cerramos las cabeceras del archivo
    c_str += f"#endif // {model_name.upper()}_H\n"

    return c_str

return c_str

[18] model_array = tflite_to_array(tflite_model, model_name)

with open(f"{model_name}.h", "w") as f:
    f.write(model_array)

```

## Comprensión

1. ¿Qué pasos se deben seguir para entrenar una red neuronal con Keras?
  - Definir el modelo.
  - Compilar el modelo especificando optimizador, función de pérdida y métricas.
  - Preparar los datos.
  - Entrenar el modelo con el conjunto de entrenamiento
  - Evaluar el rendimiento con el conjunto de prueba
2. ¿Cuál es la función del conjunto de entrenamiento y del conjunto de prueba en el proceso de entrenamiento?
  - El conjunto de entrenamiento se usa para ajustar los pesos del modelo.
  - El conjunto de prueba evalúa el rendimiento generalizado del modelo, comprobando que no haya sobreajuste.
3. ¿Qué se entiende por función de pérdida, optimizador y métricas en el contexto del entrenamiento de una red neuronal?
  - Función de pérdida: mide qué tan bien o mal está funcionando el modelo.
  - Optimizador: ajusta los pesos para minimizar la pérdida.
  - Métricas: evalúan el rendimiento del modelo durante el entrenamiento.
4. ¿Qué tipo de problemas se pueden resolver utilizando una red neuronal entrenada con Keras?

Clasificación, regresión, reconocimiento de imágenes, procesamiento de lenguaje natural, predicción de series temporales, entre otros.

**5. 5.- ¿Qué es IA on the Edge?**

Es el uso de inteligencia artificial directamente en dispositivos locales, sin necesidad de depender de servidores o la nube para realizar cálculos.

**6. 6.- ¿Qué ventajas tiene IA on the Edge en los sistemas embebidos?**

Menor latencia, mayor seguridad de datos, menor consumo de ancho de banda y operación en tiempo real, incluso sin conexión a internet.

**7. 7.- ¿Qué es TensorFlow Lite?**

Es una versión optimizada de TensorFlow diseñada para ejecutar modelos de machine learning en dispositivos móviles y sistemas embebidos.

## Conclusiones

En conclusión, entrenar una red neuronal con Keras implica seguir un proceso estructurado que incluye la preparación de datos, la definición del modelo y su evaluación. La importancia de usar conjuntos de entrenamiento y prueba asegura que el modelo generalice correctamente, evitando el sobreajuste. Además, herramientas como TensorFlow Lite permiten llevar la inteligencia artificial a dispositivos embebidos, impulsando el desarrollo de soluciones de IA on the Edge, que ofrecen ventajas como mayor eficiencia, menor latencia y mejor privacidad de los datos.

El reto en esta práctica fue la falta de espacio de almacenamiento en mi computadora y la falta de conocimiento de microcontroladores y de Arduino, mismos que me impidió desarrollar por completo la práctica.