```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
import ipywidgets as widgets
from IPython.display import display, HTML

# Cargar el archivo de datos
data = pd.read_excel('datos.xlsx')

# Preparar los datos: Selección de columnas relevantes
data.columns = data.columns.str.strip()
input_columns = [
    "pH", "Sólidos Disueltos Totales", "Sólidos Suspendidos Totales", "Sólidos Sedimentables",
    "Cloruros", "Sulfatos", "Color", "Demanda quimica de oxigeno", "Demanda Bioquímica de oxígeno
    "Conductividad", "Turbidez"
]
output_columns = ["ACIDO", "COAGULANTE", "DECOLORANTE", "ANIONICO", "CATIONICO"]
category_columns = ["UNIDAD", "FECHA TRATAMIENTO"]  # Columnas categóricas

# Filtrar filas donde las columnas relevantes sean NaN
data = data.dropna(subset=input_columns + output_columns)

# Generar datos sintéticos para aumentar el tamaño del dataset
def generate_synthetic_data(df, n_samples=1000, noise_factor=0.05, category_cols=None):
    numeric_cols = [col for col in df.columns if col not in category_cols]
    synthetic_data = df.copy()
    for _ in range(n_samples // len(df)):
        noisy_data = df[numeric_cols].apply(lambda x: x + np.random.normal(0, noise_factor * np.s
        for cat_col in category_cols:
            noisy_data[cat_col] = df[cat_col].values  # Conservar las categorías originales
        synthetic_data = pd.concat([synthetic_data, noisy_data], ignore_index=True)
    return synthetic_data

# Aplicar la generación de datos
data = generate_synthetic_data(data, n_samples=5000, category_cols=category_columns)

# Separar datos de entrada y salida (sin incluir columnas categóricas en el modelo)
X = data[input_columns]
y = data[output_columns]

# Normalizar los datos de entrada
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Definir el modelo de redes neuronales
model = Sequential([
    Dense(512, input_dim=X_train.shape[1], activation='relu'),  # Capa oculta 1
    Dropout(0.3),                                               # Regularización
    Dense(256, activation='relu'),                             # Capa oculta 2
    Dropout(0.3),
    Dense(128, activation='relu'),                             # Capa oculta 3
    Dropout(0.3),
    Dense(64, activation='relu'),                              # Capa oculta 4
    Dense(32, activation='relu'),                              # Capa oculta 5
    Dense(y_train.shape[1], activation='linear')               # Capa de salida
])

# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.0001), loss='mean_squared_error', metrics=['mae'])

# Configurar Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Entrenar el modelo
history = model.fit(
    X_train, y_train,
    epochs=150,
```

```
    batch_size=16,
    validation_split=0.2,
    verbose=1,
    callbacks=[early_stopping]
)

# Evaluar el modelo
evaluation = model.evaluate(X_test, y_test, verbose=1)
mae = evaluation[1]
print(f"\nEvaluación del modelo:")
print(f"Loss (MSE): {evaluation[0]:.4f}, MAE: {mae:.4f}")

# Calcular porcentaje de certeza basado en MAE
output_range = y.max().max() - y.min().min()  # Rango aproximado de las salidas
certainty = 100 - (mae / output_range * 100)
print(f"Porcentaje de certeza del modelo basado en MAE: {certainty:.2f}%")

# Graficar el entrenamiento del modelo
def plot_training_history(history):
    plt.figure(figsize=(12, 5))

    # Gráfica de pérdida
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Pérdida de Entrenamiento')
    plt.plot(history.history['val_loss'], label='Pérdida de Validación')
    plt.title('Pérdida durante el Entrenamiento')
    plt.xlabel('Épocas')
    plt.ylabel('Pérdida')
    plt.legend()

    # Gráfica de MAE
    plt.subplot(1, 2, 2)
    plt.plot(history.history['mae'], label='MAE de Entrenamiento')
    plt.plot(history.history['val_mae'], label='MAE de Validación')
    plt.title('MAE durante el Entrenamiento')
    plt.xlabel('Épocas')
    plt.ylabel('MAE')
    plt.legend()

    plt.show()

# Mostrar las gráficas
plot_training_history(history)

# Crear la interfaz gráfica con ipywidgets
widgets_list = {}
for column in input_columns:
    widgets_list[column] = widgets.FloatText(description=f"{column}:",
                                             placeholder="Dejar vacío si no se midió",
                                             style={'description_width': 'initial'})

button = widgets.Button(description="Calcular Dosificación", button_style='success', tooltip="Hag
output = widgets.Output()

# Función para generar predicciones
def calcular_dosificacion(button):
    # Recopilar los valores ingresados
    input_data = [widget.value if widget.value else X[column].mean() for column, widget in widget
    input_data_scaled = scaler.transform([input_data])  # Normalización
    predicted_dosage = model.predict(input_data_scaled)

    # Mostrar predicciones
    with output:
        output.clear_output()
        display(HTML("<h4 style='color: blue;'>Recomendación de dosificación (en PPM):</h4>"))
        for i, chem in enumerate(output_columns):
            display(HTML(f"<p>{chem}: <b>{predicted_dosage[0][i]:.2f} PPM</b></p>"))

button.on_click(calcular_dosificacion)

# Encabezado de la interfaz
header = widgets.HTML(value=f"<h2 style='color: green;'>Sistema de Dosificación Inteligente</h2>"
precision_widget = widgets.HTML(value=f"<h4 style='color: yellow;'>Precisión del modelo: {certain

# Mostrar la interfaz
display(widgets.VBox([header, precision_widget, widgets.HTML("<h4>Ingrese los valores medidos:</h
                      widgets.VBox(list(widgets_list.values())), button, output]))
```

154