Práctica 2: Inteligencia y Razonamiento

Enunciado:
Vamos a resolver un problema muy conocido en aprendizaje automático que es la Clasificación de flores basándose en la longitud de sus pétalos, anchura etc.

Primeramente, vamos a inicializar todas las librerías con la misma semilla, para que en todo momento nos muestre el mismo resultado.

```
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)
[729]
```

Una vez hemos hecho este paso, el cual es necesario para que, al ejecutar lo mismo dos veces, no nos muestre distintos resultados, procedemos a realizar el entrenamiento de nuestra red neuronal. Todo esto estará dividido en una serie de pasos:

Paso 1: Tratamiento y análisis del dataset

Para ello, primeramente, tenemos que extraer la información del dataset en crudo, sin embargo, aún no se puede analizar, debido a que es un formato que no se puede interpretar.

```
# Paso 1: Extraemos el dataset y lo analizamos

# Extraccion de datos del dataset
dataset = datasets.load_iris()
x = dataset.data
y = dataset.target
```

Una vez hecho esto, debemos analizar un poco el dataset para ver los tipos de datos que hay, si hay nulos, si la columna "target" es de clasificación binaria o si es clasificación múltiple… para ello, usaremos la librería pandas, con la que transformaremos el dataset en un formato legible y con el que podemos hacer un análisis más en profundidad.

```
# Convertir a dataframe para su representacion
dataframe = pd.DataFrame(x, columns = dataset.feature_names)
dataframe['target'] = dataset.target

# Para evitar overfitting le hace un barajado del dataset
dataframe=dataframe.sample(frac=1, random_state=42).reset_index(drop=True)
```

En la captura se ve como se transforma el dataset original en crudo a un dataframe más entendible. A continuación, se muestra como está representado el dataset, formado por 4 variables independientes y 1 variable dependiente (objetivo):

```
# Mostramos el dataframe
dataframe
[731]
```

11 rows | 150 rows × 5 columns | Static Output

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	6.1	2.8	4.7	1.2	1
1	5.7	3.8	1.7	0.3	0
2	7.7	2.6	6.9	2.3	2
3	6.0	2.9	4.5	1.5	1
4	6.8	2.8	4.8	1.4	1
...	...	...	...	...	...
145	6.1	2.8	4.0	1.3	1
146	4.9	2.5	4.5	1.7	2
147	5.8	4.0	1.2	0.2	0
148	5.8	2.6	4.0	1.2	1
149	7.1	3.0	5.9	2.1	2

El dataset contiene 150 registros y 5 columnas.

```
15 dataframe.shape
   [730]
   (150, 5)
```

Podemos verlo de una manera más resumida, en donde, podemos apreciar, 5 columnas, con 150 registros no nulos (importante lo de no nulos) en cada una y el contenido de las variables independientes son valores de tipo float, mientras que el contenido de la variable dependiente son enteros.

```
# Informacion resumida --> 4 variables independientes, 1 variable resultado con 150 registros y no nulos
dataframe.info()
[732]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

A continuación, vemos un resumen sobre la media, los cuartiles, los mínimos, máximos… de cada una de las columnas del dataset.

```
# Información general
dataframe.describe()
[733]
```

8 rows × 5 columns

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Como última observación, nos vamos a centrar en la salida, es decir, los resultados, que se encuentran en la columna "target", donde se puede apreciar que se trata de un dataset de clasificación múltiple (aprendizaje supervisado), en el que se van a clasificar en 3 clases: 1, 2 o 3.

```
# Información sobre la salida
dataframe["target"].value_counts()
[734]
```

```
target
1    50
0    50
2    50
Name: count, dtype: int64
```

Paso 2: División del dataset en entrenamiento, test y validación

En la siguiente captura se ve como dividimos el dataset en 3 conjuntos:

- El conjunto entrenamiento contendrá el 75% de los registros del dataset
- El 25% restante, se lo llevan el conjunto de test (55%) y el conjunto de la validación (45%)

```
# Paso 2: Dividir el dataset: train, test, validacion
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=0, stratify=y, shuffle=True)
x_test, x_val, y_test, y_val = train_test_split(x_test, y_test, test_size=0.45, random_state=0, stratify=y_test, shuffle=True)

x_train.shape, x_test.shape, x_val.shape, y_train.shape, y_test.shape, y_val.shape
[735]
```

```
((112, 4), (20, 4), (18, 4), (112,), (20,), (18,))
```

Paso 3: Estandarizamos / Normalizamos el dataset

Este paso es muy importante, ya que va a transformar los datos en un formato en donde tienen la misma escala (media 0 y desviación 1), de manera que ninguna variable sea más importante a otra. Para ello, usaremos la función StandarScaler, la cual nos permite normalizar los datos numéricos de la matriz de características.

```
# Paso 3: Estandarizamos el dataset (ya estan en label encoding la salida)
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)

x_train
[736]
```

```
array([[ 0.27307375, -0.52956568,  0.12849102,  0.1185571 ],
       [-0.0771277 , -0.75087671,  0.07249493, -0.01291216],
       [ 0.15633993, -1.85743186,  0.12849102, -0.27585067],
       [-0.54406297,  1.90485565, -1.38340334, -1.0646662 ],
       [ 0.97347666, -0.08694362,  0.80044407,  1.43324964],
       [ 0.62327521,  0.13436741,  0.96843233,  0.77590337],
       [-1.47793352,  0.35567844, -1.32740725, -1.32760471],
       [ 0.38980757, -1.85743186,  0.40847146,  0.38149561],
       [-0.31059534, -0.08694362,  0.40847146,  0.38149561],
       [-1.3611997 ,  0.35567844, -1.21541508, -1.32760471],
       [ 1.79061339, -0.52956568,  1.30440886,  0.90737262],
       [ 1.2069443 ,  0.13436741,  0.74444798,  1.43324964],
       [ 0.62327521, -0.75087671,  0.85644016,  0.90737262]])
```

En este caso, no será necesario normalizar la columna de resultados (labelEncoding o variables dummy), ya que se encuentran en un formato que entiende el modelo (salidas: 1, 2, 3).

Paso 4: Creación del modelo

En este paso, vamos a crear el modelo, es decir, vamos a estructurar como va a ser nuestra red neuronal. De esta manera, nuestra red neuronal quedaría de la siguiente manera:

- Una primera capa de entrada con 4 neuronas, que corresponden con las cuatro variables de entrada del dataset.
- Una segunda capa oculta con 32 neuronas, donde se emplea la función de activación RELU, que es muy utilizada en capas internas.
- La tercera capa oculta, tiene un total de 16 neuronas, en donde, se vuelve a emplear la misma función de activación (Relu)
- La cuarta capa oculta, es igual, contiene 8 neuronas y tiene la función de activación Relu.
- La quinta y última capa de salida, contiene 3 neuronas, que equivale al número de clases en las que se clasifica el dataset. Para ello, la función de activación que se usa para clasificación múltiple es la softmax.

```
# Paso 4: Creamos el modelo (estructura neuronal)

modelo = models.Sequential([
    layers.Input(shape=(x_train.shape[1],)),
    layers.Dense(32, activation='relu'),
    #layers.BatchNormalization(),
    layers.Dense(16, activation='relu'),
    #layers.BatchNormalization(),
    layers.Dense(8, activation='relu'),
    layers.Dense(3, activation='softmax'),
])
#early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1, restore_best_weights=True)
```

```
modelo.summary()
[737]
```

Model: "sequential_56"

Layer (type)	Output Shape	Param #
dense_205 (Dense)	(None, 32)	160
dense_206 (Dense)	(None, 16)	528
dense_207 (Dense)	(None, 8)	136
dense_208 (Dense)	(None, 3)	27

Total params: 851 (3.32 KB)

Trainable params: 851 (3.32 KB)

Non-trainable params: 0 (0.00 B)

Paso 5: Compilación del modelo

En este paso compilamos el modelo, es decir, indicamos la función de coste (loss) que se va a usar para calcular el error, la función para la reducción de la función de coste, el backpropagation (optimizer) y las métricas con las que vamos a comprobar si nuestro modelo está siendo bien entrenado o no.

Para ello, utilizaremos como función para reducir la función de coste la función Adam y como función de pérdida, utilizaremos SparseCategoricalCrossentropy, que se encarga de tratar con datos de una dimensión (LabelEncoding), ya que si hubiéramos normalizado los datos de salida con un OneHotEncoding, hubiéramos tenido que utilizar otra función que nos permitiera tratar con 3 dimensiones (un dato por cada salida)

```
# Paso 5: Compilar el modelo --> necesitamos: funcion de coste (loss), funcion de backpropagation (optimizer + tasa de aprendizaje) y metricas
learning_rate= 0.001
optimizer= Adam(learning_rate=learning_rate)

modelo.compile(
    loss = SparseCategoricalCrossentropy(from_logits=False), # estamos con label encoding
    metrics = ["accuracy"],
    optimizer = optimizer
)
```

Paso 6: Entrenamos el modelo

Este entrenamiento será de 300 épocas e irá cogiendo datos del dataset de 32 en 32 y ajustará los pesos.

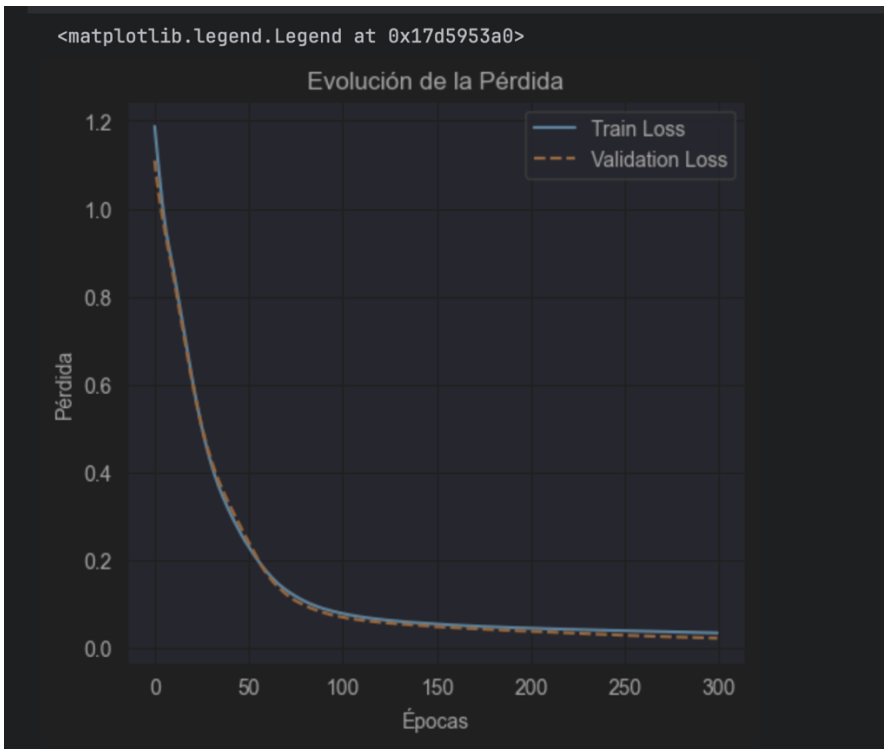
```
# Paso 6: Entrenamos el modelo
history = modelo.fit(x_train, y_train, epochs = 300, validation_data = (x_test, y_test),batch_size= 32,verbose = 1 )
[739]
```

Paso 7: Dibujo de la gráfica de pérdidas

```python
# Paso 7: Grafica de perdida
history_dict = history.history

# Pérdida (loss)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history_dict['loss'], label='Train Loss')
plt.plot(history_dict['val_loss'], label='Validation Loss', linestyle='--')
plt.title('Evolución de la Pérdida')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
[740]
```

Como se puede apreciar, no hay overfitting, eso significa que el modelo está aprendiendo correctamente



Paso 8: Evaluamos el modelo

Vamos a comparar mediante métricas los resultados obtenidos al usar el conjunto de test y el conjunto de validación. Para que sea un buen modelo, los resultados del de validación deben ser mejores. Tal y como se aprecia en la captura, la pérdida es muy pequeña, pero en el de validación es ligeramente mejor, mientras que en el accuracy ambos obtienen resultados perfectos

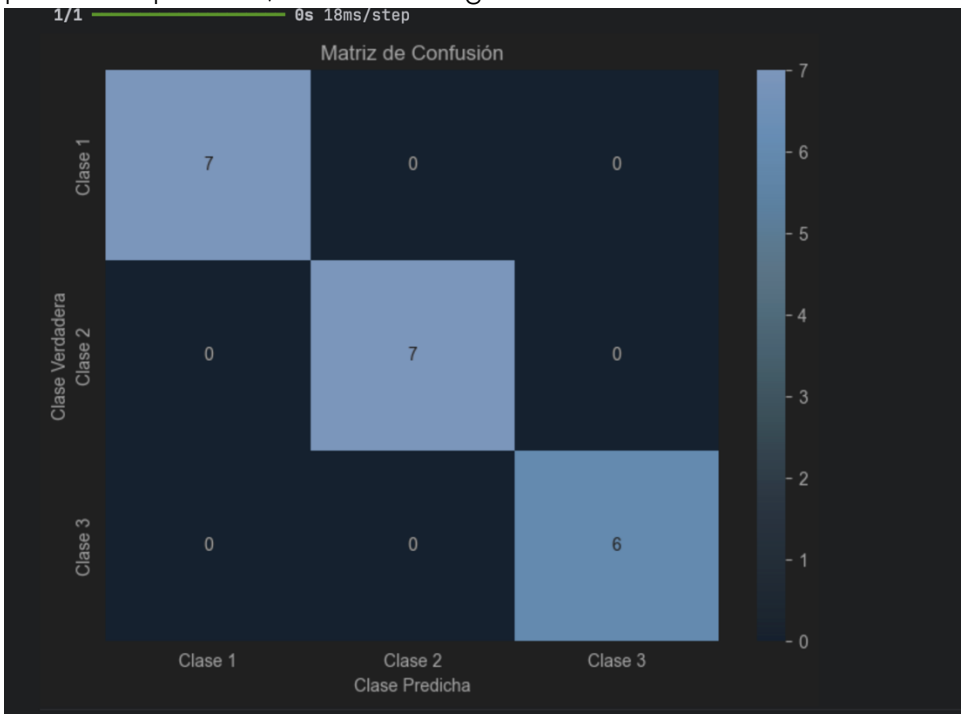
```python
# Paso 8: Evaluamos el modelo
test_loss, test_accuracy = modelo.evaluate(x_test, y_test, verbose=0)
print(f"Loss en prueba: {test_loss}")
print(f"Accuracy en prueba: {test_accuracy}")

print("******************************")
test_loss, test_accuracy = modelo.evaluate(x_val, y_val, verbose=0)
print(f"Loss en validacion: {test_loss}")
print(f"Accuracy en validacion: {test_accuracy}")
```

[741]

```
Loss en prueba: 0.023026520386338234
Accuracy en prueba: 1.0
******************************
Loss en validacion: 0.0257782693952322
Accuracy en validacion: 1.0
```

Por otro lado, realizamos la matriz de confusión para la que vamos a comparar los verdaderos positivos frente a los falsos positivos. Tal y como se puede ver, hace una predicción perfecta, no falla en ningún momento.



Aquí mostramos una comparativa de las distintas métricas para cada clase.

Classification Report:

	precision	recall	f1-score	support
0	1.0000	1.0000	1.0000	7
1	1.0000	1.0000	1.0000	7
2	1.0000	1.0000	1.0000	6
accuracy			1.0000	20
macro avg	1.0000	1.0000	1.0000	20
weighted avg	1.0000	1.0000	1.0000	20