

▼ Práctica 2:

Consideraciones en el entrenamiento de las redes neuronales

En esta segunda práctica vamos a enfocarnos en el proceso de entrenamiento de una red neuronal. El entrenamiento es el proceso por el cual nuestra red neuronal "aprende" de los datos de entrenamiento, es decir, intenta aproximar la función que ha generado los datos originales. Esta etapa es en la que más tiempo invierte un desarrollador de IA, ya que el éxito de la futura solución dependerá (en gran parte) de un entrenamiento óptimo.

▼ Aprendizaje técnico:

- Utilizar la librería de Machine Learning [Scikit-Learn](#)
- Utilizar la librería de visualización de datos [Matplotlib](#)
- Utilizar en un nivel más avanzado la API [Keras](#) de [Tensorflow](#)

Consideraciones generales en el uso de Notebooks

- Asegúrate de que todo el código necesario para ejecutarlo completamente está disponible.
- Asegúrate de usar un orden lógico de ejecución de celdas para permitir una ejecución completa y automática. El Notebook debe poder funcionar perfectamente simplemente lanzando la opción "Restart & Run All" de la sección "Kernel". Cuidado con instanciar variables o métodos en celdas posteriores a las cuales están utilizadas, una ejecución independiente y manual puede hacer que funcione, pero la ejecución automática fallará.
- Asegúrate de utilizar las celdas de tipo "Markdown" para texto y las de "Code" para código. Combinar distintos tipos de celda ayuda a la explicabilidad de la resolución y a la limpieza en general.
- Puedes crear celdas adicionales si lo necesitas.
- A la hora de la entrega del Notebook definitivo, asegúrate que lo entregas ejecutado y con las salidas de las celdas guardadas. Esto será necesario para la generación del PDF de entrega.
- El código cuanto más comentado mejor. Además, cualquier reflexión adicional siempre será bienvenida.

```
import keras
from keras.models import Sequential
import tensorflow as tf
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

▼ Métodos auxiliares

Se proporcionan ciertos métodos para la visualización de gráficos. Utilízalos cuando sea necesario y acompáñalos de la máxima información posible.

```
def plot_acc(history, title="Model Accuracy"):
    """Imprime una gráfica mostrando la accuracy por epoch obtenida en un entrenamiento
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title(title)
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'], loc='best')
    plt.show()
```

```
def plot_loss(history, title="Model Loss"):
    """Imprime una gráfica mostrando la pérdida por epoch obtenida en un entrenamiento
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title(title)
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'], loc='best')
    plt.show()
```

```
def plot_compare_losses(history1, history2, name1="Red 1",
                        name2="Red 2", title="Graph title"):
    """Compara losses de dos entrenamientos con nombres name1 y name2"""
    plt.plot(history1.history['loss'], color="green")
    plt.plot(history1.history['val_loss'], 'r--', color="green")
    plt.plot(history2.history['loss'], color="blue")
    plt.plot(history2.history['val_loss'], 'r--', color="blue")
    plt.title(title)
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train ' + name1, 'Val ' + name1,
                'Train ' + name2, 'Val ' + name2],
                loc='upper right')
    plt.show()
```

```
def plot_compare_accs(history1, history2, name1="Red 1",
                      name2="Red 2", title="Graph title"):
    """Compara accuracies de dos entrenamientos con nombres name1 y name2"""
    plt.plot(history1.history['accuracy'], color="green")
    plt.plot(history1.history['val_accuracy'], 'r--', color="green")
    plt.plot(history2.history['accuracy'], color="blue")
    plt.plot(history2.history['val_accuracy'], 'r--', color="blue")
    plt.title(title)
    plt.ylabel('Accuracy')
```

```
plt.xlabel('Epoch')
plt.legend(['Train ' + name1, 'Val ' + name1,
           'Train ' + name2, 'Val ' + name2],
           loc='lower right')
plt.show()
```

▼ 1. Diagnóstico y solución de un modelo I (4 p.)

En este primer ejercicio, se proporciona un modelo de base **modelo_1** y un conjunto de datos creado sintéticamente (gracias al método [make_classification](#) de Keras) identificado por **X_train_1**, **X_val_1**, **X_test_1**, **y_train_1**, **y_val_1**, **y_test_1**. El objetivo es diagnosticar qué le ocurre al modelo y crear una segunda versión del modelo, llamado **modelo_1_v2**, que se entrene sobre el mismo conjunto de datos y que no sufra el mismo problema que el original (o si lo sufre que al menos sea de forma claramente reducida).

El diagnóstico del modelo original debe ir acompañado de las máximas evidencias y justificaciones posibles, así como cada una de las técnicas o consideraciones de diseño elegidas para solventar el problema.

IMPORTANTE: NO editar aquellas celdas protegidas con la etiqueta #NOTOCAR

Creamos el dataset para un problema de clasificación multiclase con las siguientes propiedades:

- 1000 ejemplos (*n_samples*)
- 50 atributos (*n_features*)
- 25 de ellos "informativos", es decir, valiosos (*n_informative*)
- 4 clases de salida (*n_classes*)
- Cada clase agrupada en un clúster (*n_clusters_per_class*)
- Un 30% de las clases de salida se asignarán aleatoriamente, introduciendo ruido en el conjunto de datos (*flip_y*)

El atributo *random_state* se utiliza como semilla para la generación de aleatoriedad. Si fijamos la semilla nos aseguramos que los datos siempre serán los mismos para distintas ejecuciones.

El conjunto de datos lo dividiremos en los 3 subconjuntos correspondientes: train, validation y test. Lo repartiremos de la siguiente manera:

- 64% de los datos para train
- 16% de los datos para validación
- 20% de los datos para test

#NOTOCAR

```
X, y = make_classification(n_samples=1000, n_features=50, n_informative=25, n_redundant=25,
                           #25 feats son contributivas a las clases.
                           random_state=1)
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(X, y, test_size=0.2)
X_train_1, X_val_1, y_train_1, y_val_1 = train_test_split(X_train_1, y_train_1, test_size=0.2)

X_train_1.shape

(640, 50)
```

Double-click (or enter) to edit

Creamos el modelo **modelo_1** que vamos a usar como base y sobre el cual habrá que dar un diagnóstico de su desempeño sobre el conjunto de datos creamos anteriormente. Originalmente, el diseño del modelo será el siguiente:

- Una capa de entrada de 50 neuronas
- 2 hidden layers de 2056 y 1024 neuronas, respectivamente, que usarán ReLU como función de activación
- 1 capa de salida adaptada al problema de clasificación multiclase

Es importante usar el método *clear_session* para limpiar el grafo de computación del modelo y empezar de cero

```
#NOTOCAR
tf.keras.backend.clear_session()
modelo_1 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(2056, input_shape = (50,), activation='relu'),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(4, activation='softmax')
])
```

Con la llamada al método *summary* podemos ver cierta información sobre la arquitectura del modelo:

- La secuencia de capas con sus tamaños
- El número de parámetros entrenables y no que tiene la red, desglosados por capa y totales

```
#NOTOCAR
modelo_1.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
dense (Dense)	(None, 2056)	104856

```
dense_1 (Dense)          (None, 1024)          2106368

dense_2 (Dense)          (None, 4)              4100
```

```
=====
Total params: 2,215,324
Trainable params: 2,215,324
Non-trainable params: 0
=====
```

Compilación del modelo, utilizando Adam como optimizador y la función de coste apropiada para el problema de clasificación multiclase:


```
#NOTOCAR
modelo_1.compile(optimizer="adam",
                  loss="sparse_categorical_crossentropy",
                  metrics=['accuracy'])
```

Entrenamiento del modelo, durante 20 épocas, utilizando un tamaño de batch de 256 y utilizando el subconjunto creado anteriormente para validar:

```
#NOTOCAR
N_EPOCHS = 20
history_1 = modelo_1.fit(X_train_1, y_train_1, epochs=N_EPOCHS, batch_size = 256, vali
```

```
Epoch 1/20
3/3 [=====] - 1s 116ms/step - loss: 1.9041 - accuracy: 0
Epoch 2/20
3/3 [=====] - 0s 68ms/step - loss: 1.3608 - accuracy: 0
Epoch 3/20
3/3 [=====] - 0s 58ms/step - loss: 0.9623 - accuracy: 0
Epoch 4/20
3/3 [=====] - 0s 59ms/step - loss: 0.8780 - accuracy: 0
Epoch 5/20
3/3 [=====] - 0s 67ms/step - loss: 0.7836 - accuracy: 0
Epoch 6/20
3/3 [=====] - 0s 67ms/step - loss: 0.6569 - accuracy: 0
Epoch 7/20
3/3 [=====] - 0s 67ms/step - loss: 0.5930 - accuracy: 0
Epoch 8/20
3/3 [=====] - 0s 69ms/step - loss: 0.4883 - accuracy: 0
Epoch 9/20
3/3 [=====] - 0s 59ms/step - loss: 0.4139 - accuracy: 0
Epoch 10/20
3/3 [=====] - 0s 60ms/step - loss: 0.3497 - accuracy: 0
Epoch 11/20
```

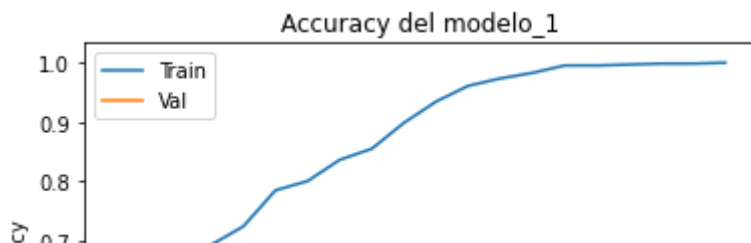
```
3/3 [=====] - 0s 61ms/step - loss: 0.2903 - accuracy: 0
Epoch 12/20
3/3 [=====] - 0s 63ms/step - loss: 0.2372 - accuracy: 0
Epoch 13/20
3/3 [=====] - 0s 60ms/step - loss: 0.1849 - accuracy: 0
Epoch 14/20
3/3 [=====] - 0s 59ms/step - loss: 0.1496 - accuracy: 0
Epoch 15/20
3/3 [=====] - 0s 65ms/step - loss: 0.1162 - accuracy: 0
Epoch 16/20
3/3 [=====] - 0s 60ms/step - loss: 0.0896 - accuracy: 0
Epoch 17/20
3/3 [=====] - 0s 63ms/step - loss: 0.0707 - accuracy: 0
Epoch 18/20
3/3 [=====] - 0s 67ms/step - loss: 0.0535 - accuracy: 0
Epoch 19/20
3/3 [=====] - 0s 66ms/step - loss: 0.0422 - accuracy: 0
Epoch 20/20
3/3 [=====] - 0s 62ms/step - loss: 0.0334 - accuracy: 1
```



Mostramos resultados de *accuracy* y *loss* utilizando los métodos auxiliares proporcionados al inicio de la práctica:

```
#NOTOCAR
```

```
plot_acc(history_1, title = "Accuracy del modelo_1")
plot_loss(history_1, title = "Loss del modelo_1")
```



Evaluamos el rendimiento del **modelo_1** ya entrenado sobre el conjunto de test:

```
#NOTOCAR
```

```
metricas = modelo_1.evaluate(X_test_1, y_test_1)
print("Accuracy del modelo_1 en test:", metricas[1])
print("Loss del modelo_1 en test:", metricas[0])
```

```
7/7 [=====] - 0s 5ms/step - loss: 1.4961 - accuracy: 0.6000000238418579
Accuracy del modelo_1 en test: 0.6000000238418579
Loss del modelo_1 en test: 1.4961427450180054
```

▼ 1.1. Diagnóstico (2 p.)

Una vez creado, entrenado y evaluado el modelo, es momento de sacar conclusiones. Hay que recordar que el problema que estamos enfrentando tiene 4 clases, por lo que si la decisión del modelo fuese completamente al azar, obtendría en torno a un 25% de accuracy. Observa el valor de *accuracy* en test, es evidente que algo le ocurre al **modelo_1**.

Explica lo que consideras que le está ocurriendo, acompañado del mayor número de evidencias y explicaciones posibles. Puedes usar tanto celdas de texto como de código por si consideras necesario ejecutar alguna prueba adicional.

Respuesta aquí

Mi Diagnóstico al Modelo 1

- El modulo al parece tiene overfitting, se puede ver en la representacion del accuracy y el loss de los subconjuntos de validacion, ya que nunca mejora con el train conjunto de accuracy, ni empeora con el loss, como podemos ver en las graficas del plot.
- La razón mas viable que le veo al problema es que tiene muchas neuronas en sus capas hidden, que ocasionan muchos parametros a procesar, como se ve en el summary
- No hay regularizacion ni ninguna tecnica para prevenir el overwitting.
- No hay procesamiento de datos previos al modelaje, como outliers o normalizacion de datos que afectan el ML, aunque debido a nuestro DS esto no es tan necesario

▼ 1.2. Modelo optimizado (2 p.)

Una vez hayas diagnosticado el problema, es momento de ponerle solución. Para ello, debes crear una versión nueva del modelo, almacenandola en la variable **modelo_1_v2**, que cuando se entrene sobre el mismo conjunto de datos que el **modelo_1** consiga:

- Por un lado, mejor accuracy en el conjunto de test
- Por otro lado, que se minimice al máximo posible el problema que sufría el modelo original

Se entiende que para dar con la versión definitiva del modelo tendrás que hacer muchas pruebas. No es necesario plasmar todo ese proceso, lo puedes hacer aquí y luego borrar las celdas o hacerlo en un proyecto separado, dejando únicamente el modelo final. Lo fundamental es que TODAS las decisiones que hayas tomado estén justificadas y que expliques como ayuda cada una de ellas a solventar el problema original que sufría el **modelo_1**.

¿Qué NO se puede modificar?

Para intentar que los escenarios de entrenamiento del **modelo_1** y del **modelo_1_v2** sean lo más similares posibles, hay ciertas cosas que no se pueden modificar:

- El número de épocas de entrenamiento: está almacenado en la variable **N_EPOCAS**
- El conjunto de datos original y los subconjuntos creados a partir de él

¿Que SI que se puede modificar?

Evidentemente habrá que hacer modificaciones en uno o varios puntos del diseño y entrenamiento del **modelo_1_v2** para que su desempeño sea diferente. Las modificaciones permitidas son las siguientes (si tienes dudas de si una modificación está permitida no dudes en escribir al profesor a través del correo o del tablón):

- Número de capas y número de neuronas por capa
- Funciones de activación
- Optimizador
- Capas de regularización
- Adición de callbacks en el entrenamiento

Es importante que la identificación del problema sea correcta, ya que las modificaciones tienen que ir acorde a solucionarlo. Cada modificación de las propuestas arriba va destinada a solucionar uno o varios problemas, por lo que no será necesario aplicarlas todas: simplemente las justas y necesarias.

Debes mostrar y ejecutar un procedimiento similar al llevado a cabo con el **modelo_1**:

- Diseñar y mostrar el modelo
- Compilación
- Entrenamiento

- Muestra de gráficas
- Evaluación

Es importante demostrar que se ha resuelto el problema original. Utiliza todas las celdas de texto o código que consideres necesario

```
train_mean = X_train_1.mean(axis=0)
train_std = X_train_1.std(axis=0)

X_train_norm = (X_train_1 - X_train_1.mean(axis=0))/ X_train_1.std(axis=0)

X_val_norm = (X_val_1 - X_train_1.mean(axis=0))/ X_train_1.std(axis=0)

# jugué con normalizacion de el conjunto pero no logre hacer cambios significativos en

X_train_norm.mean(axis=0)

array([ 9.88792381e-17,  3.29597460e-17,  4.85722573e-17,  3.71881345e-18,
        2.49800181e-17, -3.81639165e-18,  1.21430643e-17,  5.13478149e-17,
       -3.78169718e-17, -2.20309881e-17, -2.57172755e-17, -1.90819582e-17,
       -1.21430643e-18,  2.39391840e-17,  9.97465999e-18,  4.02455846e-17,
        1.10675358e-16,  5.04804532e-17, -6.49328681e-17,  2.79941001e-16,
        4.93095148e-17,  4.33680869e-18,  2.18575158e-17,  6.05608228e-17,
        2.09034179e-17,  5.16947596e-17,  4.89192020e-17,  4.87457297e-17,
        1.60982339e-16, -2.77555756e-18,  2.04697370e-17,  4.63388009e-17,
        6.93889390e-19, -1.44675938e-16, -6.71337985e-17, -5.03937170e-17,
       -4.38017678e-17, -1.12973866e-17, -1.08680426e-16,  3.99333344e-16,
        2.55004351e-17, -1.15185639e-16,  1.68268177e-17,  2.53269627e-17,
        2.91433544e-17,  3.81639165e-18,  3.05137859e-16,  5.27355937e-17,
        1.02695630e-16, -2.77555756e-18])

tf.keras.backend.clear_session()
modelo_1_v2 = tf.keras.models.Sequential([
    # Tu modelo 32
    tf.keras.layers.Dense(512, input_shape = (50,), activation='relu', kernel_regularizer=
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(256, activation='relu', kernel_regularizer= tf.keras.regulariz
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(4, activation='softmax')
])

# En este caso reduje el numero de neuronas, probe con varias combinaciones, para mejor
# que es uno de los problemas mas grandes de este modelo.
# Utilizo un layer de Dropout como método para prevenir overfitting
# Al final eliminé el overfitting y mejoré considerablemente el comportamiento tanto c

modelo_1_v2.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	26112
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131328
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 4)	1028
Total params: 158,468		
Trainable params: 158,468		
Non-trainable params: 0		

```

modelo_1_v2.compile(optimizer='adamax',
                    loss="sparse_categorical_crossentropy",
                    metrics=['accuracy'])
# Utilizo ADAMAX porque lei en la documentacion que en ocasiones es mejor que el adam,
# si se usan ciertos tipos de datos (embeded strings), y aunque no es el caso en este

es_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)

N_EPOCAS = 20
history_1 = modelo_1_v2.fit(X_train_1, y_train_1, epochs=N_EPOCAS, batch_size = 256, \
# utilizo un callback es_callback para detener el entrenamiento si no hay progreso en


```

```

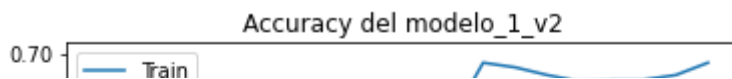
Epoch 1/20
3/3 [=====] - 1s 76ms/step - loss: 43.7290 - accuracy: 0.0000
Epoch 2/20
3/3 [=====] - 0s 15ms/step - loss: 40.2783 - accuracy: 0.0000
Epoch 3/20
3/3 [=====] - 0s 17ms/step - loss: 37.4310 - accuracy: 0.0000
Epoch 4/20
3/3 [=====] - 0s 24ms/step - loss: 34.7981 - accuracy: 0.0000
Epoch 5/20
3/3 [=====] - 0s 26ms/step - loss: 32.4128 - accuracy: 0.0000
Epoch 6/20
3/3 [=====] - 0s 18ms/step - loss: 30.1882 - accuracy: 0.0000
Epoch 7/20
3/3 [=====] - 0s 19ms/step - loss: 28.1457 - accuracy: 0.0000
Epoch 8/20
3/3 [=====] - 0s 28ms/step - loss: 26.2171 - accuracy: 0.0000
Epoch 9/20
3/3 [=====] - 0s 21ms/step - loss: 24.4329 - accuracy: 0.0000
Epoch 10/20
3/3 [=====] - 0s 19ms/step - loss: 22.7759 - accuracy: 0.0000

```

```
Epoch 11/20
3/3 [=====] - 0s 16ms/step - loss: 21.2556 - accuracy: 0.0000
Epoch 12/20
3/3 [=====] - 0s 17ms/step - loss: 19.8312 - accuracy: 0.0000
Epoch 13/20
3/3 [=====] - 0s 21ms/step - loss: 18.4921 - accuracy: 0.0000
Epoch 14/20
3/3 [=====] - 0s 16ms/step - loss: 17.2666 - accuracy: 0.0000
Epoch 15/20
3/3 [=====] - 0s 17ms/step - loss: 16.1447 - accuracy: 0.0000
Epoch 16/20
3/3 [=====] - 0s 19ms/step - loss: 15.0873 - accuracy: 0.0000
Epoch 17/20
3/3 [=====] - 0s 18ms/step - loss: 14.1216 - accuracy: 0.0000
Epoch 18/20
3/3 [=====] - 0s 25ms/step - loss: 13.2269 - accuracy: 0.0000
Epoch 19/20
3/3 [=====] - 0s 25ms/step - loss: 12.3883 - accuracy: 0.0000
Epoch 20/20
3/3 [=====] - 0s 16ms/step - loss: 11.6068 - accuracy: 0.0000
```



```
plot_acc(history_1, title = "Accuracy del modelo_1_v2")
plot_loss(history_1, title = "Loss del modelo_1_v2")
```



```
#NOTOCAR
```

```
metricas = modelo_1_v2.evaluate(X_test_1, y_test_1)
print("Accuracy del modelo_1_v2 en test:", metricas[1])
print("Loss del modelo_1_v2 en test:", metricas[0])
```

```
7/7 [=====] - 0s 2ms/step - loss: 11.1490 - accuracy: 0
Accuracy del modelo_1_v2 en test: 0.625
Loss del modelo_1_v2 en test: 11.148995399475098
```



```
# Usa todas las celdas que necesites, acompañadas de justificación en celdas de texto
```

▼ 2. Diagnóstico y solución de un modelo II (4 p.)



Este segundo ejercicio es análogo al primero. Los objetivos serán diagnosticar el problema que sufre, en este caso el **modelo_2**, cuando se entrena con el conjunto de datos creado sintéticamente (usaremos el mismo conjunto de datos que en el ejercicio 1) y crear una nueva versión del modelo **modelo_2_v2** que sea capaz de solventarlo.

Todas las exigencias en cuanto a demostraciones y explicaciones requeridas en el ejercicio 1 son aplicables a este segundo.



Creamos el modelo de referencia para diagnosticar y solucionar el problema, que lo almacenaremos en la variable **modelo_2**. Este tendrá las siguientes características:

- Una capa de entrada de 50 neuronas
- 2 hidden layers de 12 y 8 neuronas, respectivamente, que usarán ReLU como función de activación
- 1 capa de salida adaptada al problema de clasificación multiclase

```
# NOTOCAR
tf.keras.backend.clear_session()
modelo_2 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(12, input_shape = (50,), activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(4, activation='softmax')
])
```

Mostramos la descripción del modelo:

```
# NOTOCAR
```

```
modelo_2.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12)	612
dense_1 (Dense)	(None, 8)	104
dense_2 (Dense)	(None, 4)	36
Total params: 752		
Trainable params: 752		
Non-trainable params: 0		

Su compilación es idéntica a la del **modelo_1**:

```
#NOTOCAR
```

```
modelo_2.compile(optimizer="adam",
                  loss="sparse_categorical_crossentropy",
                  metrics=['accuracy'])
```

El entrenamiento, como bien se menciona en el enunciado, se va a llevar a cabo usando los mismos subconjuntos que en el ejercicio 1:

```
#NOTOCAR
```


```
history_2 = modelo_2.fit(X_train_1, y_train_1, epochs=N_EPOCHAS, batch_size = 256, vali
```

```
Epoch 1/20
3/3 [=====] - 0s 56ms/step - loss: 2.4462 - accuracy: 0
Epoch 2/20
3/3 [=====] - 0s 10ms/step - loss: 2.3428 - accuracy: 0
Epoch 3/20
3/3 [=====] - 0s 9ms/step - loss: 2.2496 - accuracy: 0.1
Epoch 4/20
3/3 [=====] - 0s 9ms/step - loss: 2.1631 - accuracy: 0.1
Epoch 5/20
3/3 [=====] - 0s 10ms/step - loss: 2.0861 - accuracy: 0
Epoch 6/20
3/3 [=====] - 0s 10ms/step - loss: 2.0154 - accuracy: 0
Epoch 7/20
3/3 [=====] - 0s 9ms/step - loss: 1.9511 - accuracy: 0.1
Epoch 8/20
3/3 [=====] - 0s 11ms/step - loss: 1.8925 - accuracy: 0
Epoch 9/20
3/3 [=====] - 0s 12ms/step - loss: 1.8396 - accuracy: 0
```

```

Epoch 10/20
3/3 [=====] - 0s 21ms/step - loss: 1.7917 - accuracy: 0
Epoch 11/20
3/3 [=====] - 0s 10ms/step - loss: 1.7478 - accuracy: 0
Epoch 12/20
3/3 [=====] - 0s 11ms/step - loss: 1.7067 - accuracy: 0
Epoch 13/20
3/3 [=====] - 0s 9ms/step - loss: 1.6710 - accuracy: 0.
Epoch 14/20
3/3 [=====] - 0s 10ms/step - loss: 1.6389 - accuracy: 0
Epoch 15/20
3/3 [=====] - 0s 11ms/step - loss: 1.6085 - accuracy: 0
Epoch 16/20
3/3 [=====] - 0s 12ms/step - loss: 1.5805 - accuracy: 0
Epoch 17/20
3/3 [=====] - 0s 11ms/step - loss: 1.5548 - accuracy: 0
Epoch 18/20
3/3 [=====] - 0s 12ms/step - loss: 1.5321 - accuracy: 0
Epoch 19/20
3/3 [=====] - 0s 14ms/step - loss: 1.5108 - accuracy: 0
Epoch 20/20
3/3 [=====] - 0s 11ms/step - loss: 1.4906 - accuracy: 0

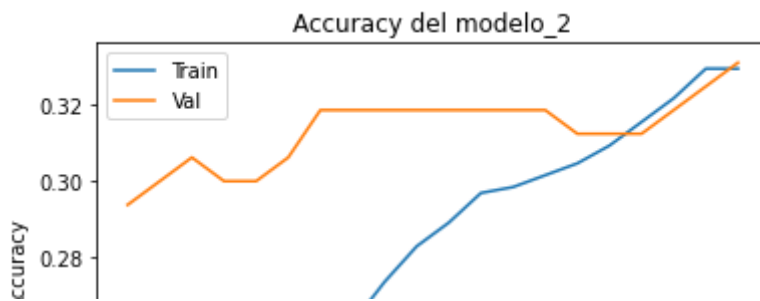
```



Mostramos las gráficas de *accuracy* y *loss*:

```
#NOTOCAR
```

```
plot_acc(history_2, title = "Accuracy del modelo_2")
plot_loss(history_2, title = "Loss del modelo_2")
```



```
metricas = modelo_2.evaluate(X_test_1, y_test_1)
print("Accuracy del modelo_2 en test:", metricas[1])
print("Loss del modelo_2 en test:", metricas[0])
```

```
7/7 [=====] - 0s 2ms/step - loss: 1.4204 - accuracy: 0.3499999940395355
Accuracy del modelo_2 en test: 0.3499999940395355
Loss del modelo_2 en test: 1.4203959703445435
```



Evaluamos el rendimiento del **modelo_2** ya entrenado sobre el conjunto de test:



▼ 2.1. Diagnóstico (2 p.)

Una vez creado, entrenado y evaluado el modelo, es momento de sacar conclusiones. Hay que recordar que el problema que estamos enfrentando tiene 4 clases, por lo que si la decisión del modelo fuese completamente al azar, obtendría en torno a un 25% de accuracy. Observa el valor de *accuracy* en test, es evidente que algo le ocurre al **modelo_2**.

Explica lo que consideras que le está ocurriendo, acompañado del mayor número de evidencias y explicaciones posibles. Puedes usar tanto celdas de texto como de código por si consideras necesario ejecutar alguna prueba adicional.

Respuesta aquí

▼ 2.2. Modelo optimizado (2 p.)

Una vez hayas diagnosticado el problema, es momento de ponerle solución. Para ello, debes crear una versión nueva del modelo, almacenandola en la variable **modelo_2_v2**, que cuando se entrene sobre el mismo conjunto de datos que el **modelo_2** consiga:

- Por un lado, mejor accuracy en el conjunto de test
- Por otro lado, que se minimice al máximo posible el problema que sufría el modelo original

Se entiende que para dar con la versión definitiva del modelo tendrás que hacer muchas pruebas. No es necesario plasmar todo ese proceso, lo puedes hacer aquí y luego borrar las celdas o hacerlo en un proyecto separado, dejando únicamente el modelo final. Lo fundamental es que TODAS las

decisiones que hayas tomado estén justificadas y que expliques como ayuda cada una de ellas a solventar el problema original que sufría el **modelo_2**.

¿Qué NO se puede modificar?

Para intentar que los escenarios de entrenamiento del **modelo_2** y del **modelo_2_v2** sean lo más similares posibles, hay ciertas cosas que no se pueden modificar:

- El número de épocas de entrenamiento: está almacenado en la variable **N_EPOCAS**
- El conjunto de datos original y los subconjuntos creados a partir de él

¿Que SI que se puede modificar?

Evidentemente habrá que hacer modificaciones en uno o varios puntos del diseño y entrenamiento del **modelo_2_v2** para que su desempeño sea diferente. Las modificaciones permitidas son las siguientes (si tienes dudas de si una modificación está permitida no dudes en escribir al profesor a través del correo o del tablón):

- Número de capas y número de neuronas por capa
- Funciones de activación
- Optimizador
- Capas de regularización
- Adición de callbacks en el entrenamiento

Es importante que la identificación del problema sea correcta, ya que las modificaciones tienen que ir acorde a solucionarlo. Cada modificación de las propuestas arriba va destinada a solucionar uno o varios problemas, por lo que no será necesario aplicarlas todas: simplemente las justas y necesarias.

Debes mostrar y ejecutar un procedimiento similar al llevado a cabo con el **modelo_2**:

- Diseñar y mostrar el modelo
- Compilación
- Entrenamiento
- Muestra de gráficas
- Evaluación

Es importante demostrar que se ha resuelto el problema original. Utiliza todas las celdas de texto o código que consideres necesario

```
tf.keras.backend.clear_session()
```

```
modelo_2_v2 = tf.keras.models.Sequential([
    # Tu modelo aquí
    tf.keras.layers.Dense(512, input_shape = (50,), activation='relu', kernel_regularizer=
```



```

tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(256, activation='relu', kernel_regularizer= tf.keras.regularizers.l2(0.01)),
tf.keras.layers.Dropout(0.1),

tf.keras.layers.Dense(4, activation='softmax'))
])
# Aumenté el numero de neuronas, de hecho probe con varias combinaciones, para poder c
# tambien apliqué un regularizador en el segundo layer para el L1 y L2 no solo para p
# mejorar el accuracy que es una de las principales fallas de este modelo.
# Utilizo un layer de Dropout como método para prevenir overfitting

# Usa todas las celdas que necesites, acompañadas de justificación en celdas de texto

```

```

modelo_2_v2.summary()

```

```

Model: "sequential"

```

Layer (type)	Output Shape	Param #
=====		
dense (Dense)	(None, 512)	26112
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131328
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 4)	1028
=====		
Total params: 158,468		
Trainable params: 158,468		
Non-trainable params: 0		
=====		

```

#Mi compilador
modelo_2_v2.compile(optimizer="adamax",
                    loss="sparse_categorical_crossentropy",
                    metrics=['accuracy'])

#Utilizo ADAMAX porque lei en la documentacion que en ocasiones es mejor que el adam
# principalmente si se usan ciertos tipos de datos, que aunque no es el caso en este c
# al final mejoré el accuracy a casi el doble del modelo original.

#Mi history 2
history_2_v2 = modelo_2_v2.fit(X_train_1, y_train_1, epochs=N_EPOCAS, batch_size = 256)

# utilizo un callback es_callback para detener el entrenamiento si no hay progreso en

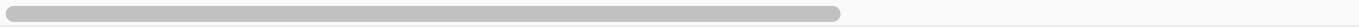
```

Epoch 1/20

```

3/3 [=====] - 1s 76ms/step - loss: 44.0325 - accuracy: 0.0000
Epoch 2/20
3/3 [=====] - 0s 17ms/step - loss: 40.6480 - accuracy: 0.0000
Epoch 3/20
3/3 [=====] - 0s 17ms/step - loss: 37.7986 - accuracy: 0.0000
Epoch 4/20
3/3 [=====] - 0s 17ms/step - loss: 35.2168 - accuracy: 0.0000
Epoch 5/20
3/3 [=====] - 0s 16ms/step - loss: 32.8152 - accuracy: 0.0000
Epoch 6/20
3/3 [=====] - 0s 16ms/step - loss: 30.5799 - accuracy: 0.0000
Epoch 7/20
3/3 [=====] - 0s 17ms/step - loss: 28.5221 - accuracy: 0.0000
Epoch 8/20
3/3 [=====] - 0s 16ms/step - loss: 26.5887 - accuracy: 0.0000
Epoch 9/20
3/3 [=====] - 0s 17ms/step - loss: 24.7999 - accuracy: 0.0000
Epoch 10/20
3/3 [=====] - 0s 16ms/step - loss: 23.1329 - accuracy: 0.0000
Epoch 11/20
3/3 [=====] - 0s 28ms/step - loss: 21.5958 - accuracy: 0.0000
Epoch 12/20
3/3 [=====] - 0s 23ms/step - loss: 20.1672 - accuracy: 0.0000
Epoch 13/20
3/3 [=====] - 0s 17ms/step - loss: 18.8482 - accuracy: 0.0000
Epoch 14/20
3/3 [=====] - 0s 19ms/step - loss: 17.5940 - accuracy: 0.0000
Epoch 15/20
3/3 [=====] - 0s 29ms/step - loss: 16.4729 - accuracy: 0.0000
Epoch 16/20
3/3 [=====] - 0s 21ms/step - loss: 15.3980 - accuracy: 0.0000
Epoch 17/20
3/3 [=====] - 0s 19ms/step - loss: 14.4307 - accuracy: 0.0000
Epoch 18/20
3/3 [=====] - 0s 17ms/step - loss: 13.5186 - accuracy: 0.0000
Epoch 19/20
3/3 [=====] - 0s 48ms/step - loss: 12.6747 - accuracy: 0.0000
Epoch 20/20
3/3 [=====] - 0s 35ms/step - loss: 11.8822 - accuracy: 0.0000

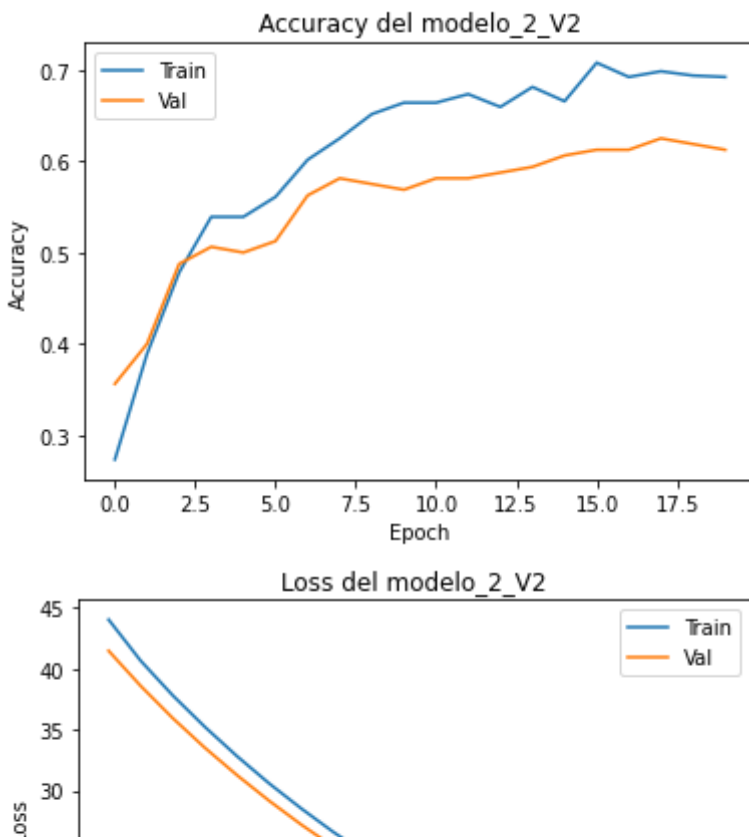
```



```

plot_acc(history_2_v2, title = "Accuracy del modelo_2_v2")
plot_loss(history_2_v2, title = "Loss del modelo_2_v2")

```



#NOTOCAR

```
metricas = modelo_2_v2.evaluate(X_test_1, y_test_1)
print("Accuracy del modelo_2_v2 en test:", metricas[1])
print("Loss del modelo_2_v2 en test:", metricas[0])
```

```
7/7 [=====] - 0s 2ms/step - loss: 11.4515 - accuracy: 0
Accuracy del modelo_2_v2 en test: 0.6399999856948853
Loss del modelo_2_v2 en test: 11.451476097106934
```

▼ 3. Libertad para diseñar un modelo (2 p.)

En este tercer y último ejercicio, serás (prácticamente) libre para crear tu propia red neuronal que resuelva un problema de clasificación multiclase. El objetivo será que diseñes, entrenes y evalúes tu propio modelo, obteniendo con ello las mejores métricas posibles en el conjunto de test. Tan fácil (o no) como eso. La evaluación dependerá de las explicaciones en cuanto al diseño y entrenamiento, así como al resultado de accuracy obtenido.

Tendrás que utilizar el conjunto de datos que se va a crear a continuación definido por **X₂** e **y₂**, que tiene las siguientes propiedades:

- 10000 registros
- 40 atributos, 5 de ellos "informativos"
- 4 clases de salida
- Un 10% de aleatoriedad en la salida

Se crearán los subconjuntos **X_train_2, X_val_2, X_test_2, y_train_2, y_val_2, y_test_2** según la siguiente división:

- 64% de los datos para train
- 16% de los datos para validación
- 20% de los datos para test

#NOTOCAR

```
X_2, y_2 = make_classification(n_samples=10000, n_informative = 5, n_features=40, n_cl
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(X_2, y_2, test_size=0.2)
X_train_2, X_val_2, y_train_2, y_val_2 = train_test_split(X_train_2, y_train_2, test_s
```

A partir de aquí habrá que diseñar, compilar, entrenar y evaluar el modelo. La única restricción será que el modelo debe de tener, al menos, 3 capas ocultas. Es la única restricción de aquí en adelante. Usa todas las celdas que consideres necesario para argumentar tu diseño y tu modelo final.

Acuérdete de considerar todas las opciones que se han visto en clase referentes a:

- Distintos optimizadores
- Distintas funciones de activación
- Distintos tipos de capas
- Distintas funciones de coste
- Distintas técnicas de regularización
- Distinto número de batch_size y épocas
- Distintos inicializadores de pesos

Realiza experimentos y juega con Keras. Acuérdete de intentar justificar los motivos por los que la red funciona mejor con una características u otra. Debido a que los datos se están generando de manera sintética y no hemos llevado a cabo el proceso de EDA, no podemos sacar todas las conclusiones ni diseñar una arquitectura acorde a los datos. Por eso el proceso será prueba y error, siempre de la manera más justificada posible.

```
tf.keras.backend.clear_session()
modelo_3 = tf.keras.models.Sequential([
    # Diseña tu modelo aquí
    tf.keras.layers.Dense(2056, input_shape = (40, ), activation='relu', kernel_regula
    tf.keras.layers.BatchNormalization(
        axis=1,
        momentum=0.99,
        epsilon=0.001,
        center=True,
```

```

        scale=True,
        beta_initializer="zeros",
        gamma_initializer="ones",
        moving_mean_initializer="zeros",
        moving_variance_initializer="ones",
        beta_regularizer=None,
        gamma_regularizer=None,
        beta_constraint=None,
        gamma_constraint=None,

    ),
    #tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1024, activation='relu', kernel_regularizer= tf.keras.regularizers.l2(0.01)),
    #tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(4, activation='softmax')
])

#
# en mi modelo quise utilizar el Batch normalization que se vio en clase, si me mejoró
# sigo usando regularizadores debido al buen resultado en el anterior.
# En este caso no usé un Drop layer ya que en mis pruebas no generó mayor mejor.

```

```

#from sklearn import preprocessing
#import pandas as pd
#import numpy as np
#df = pd.DataFrame(X_train_2)

#df.columns
#d = preprocessing.normalize(X_train_2)
#scaled_X_train_2 = pd.DataFrame(d, columns=df.columns)

#df = pd.DataFrame(X_val_2)

#d = preprocessing.normalize(X_val_2)
#scaled_X_val_2= pd.DataFrame(d, columns=df.columns)

#trate de nromaliazar los datos pero no logre mejorar los resultados.

```

```

modelo_3.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2056)	84296

```

batch_normalization (Batch Normalization) (None, 2056) 8224
dense_1 (Dense) (None, 1024) 2106368
dense_2 (Dense) (None, 4) 4100

```

```

=====
Total params: 2,202,988
Trainable params: 2,198,876
Non-trainable params: 4,112
=====

```

#Mi compilador

```

modelo_3.compile(optimizer=tf.keras.optimizers.Adamax(
    learning_rate=0.001),
    loss="sparse_categorical_crossentropy",
    metrics=['accuracy']
)

```

#en mi modelo quise jugar con el Learning rate, y logre un muy buen comportamiento, tal como si lo utilizo ya que me dio buenos resultados en los dos anteriores.

#Mi history 3

```

history_3 = modelo_3.fit(X_train_2, y_train_2, epochs=N_EPOCHAS, batch_size = 286, validation_data=(X_val_2, y_val_2))
#utilice un batch size pequeño y un learning rate pequeño para segun mi parecer, mejor
# tengo una pequeña desviación durante el inicio en unas 4 épocas, fuera de eso, mi accuracy
# el loss esta bastante similar al de validación

```

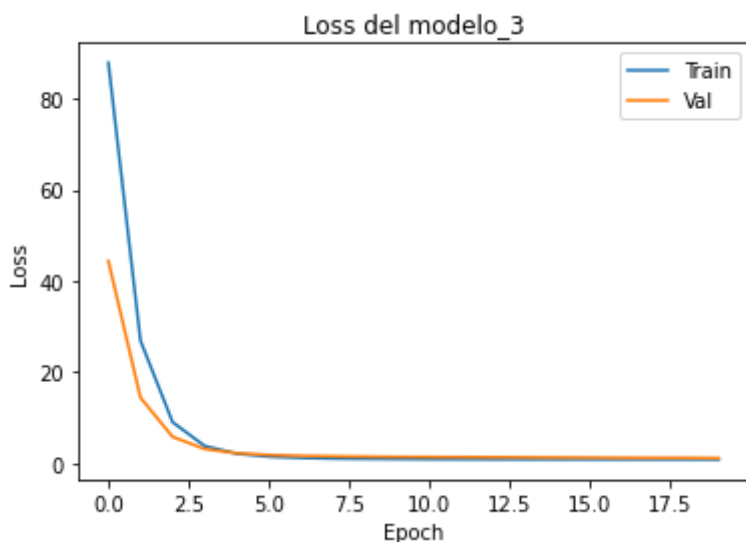
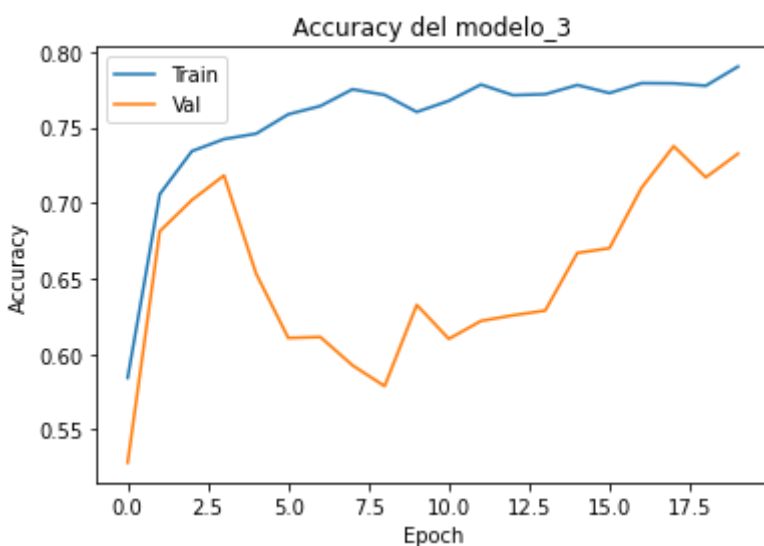
```

Epoch 1/20
23/23 [=====] - 3s 89ms/step - loss: 87.9386 - accuracy: 0.0000
Epoch 2/20
23/23 [=====] - 2s 75ms/step - loss: 26.8851 - accuracy: 0.0000
Epoch 3/20
23/23 [=====] - 2s 73ms/step - loss: 8.9911 - accuracy: 0.0000
Epoch 4/20
23/23 [=====] - 2s 73ms/step - loss: 3.7572 - accuracy: 0.0000
Epoch 5/20
23/23 [=====] - 2s 73ms/step - loss: 2.0672 - accuracy: 0.0000
Epoch 6/20
23/23 [=====] - 2s 73ms/step - loss: 1.4162 - accuracy: 0.0000
Epoch 7/20
23/23 [=====] - 2s 73ms/step - loss: 1.1444 - accuracy: 0.0000
Epoch 8/20
23/23 [=====] - 2s 73ms/step - loss: 0.9976 - accuracy: 0.0000
Epoch 9/20
23/23 [=====] - 2s 73ms/step - loss: 0.9295 - accuracy: 0.0000
Epoch 10/20
23/23 [=====] - 2s 74ms/step - loss: 0.8988 - accuracy: 0.0000
Epoch 11/20
23/23 [=====] - 2s 97ms/step - loss: 0.8610 - accuracy: 0.0000
Epoch 12/20
23/23 [=====] - 2s 82ms/step - loss: 0.8389 - accuracy: 0.0000

```

```
Epoch 13/20
23/23 [=====] - 2s 72ms/step - loss: 0.8254 - accuracy:
Epoch 14/20
23/23 [=====] - 2s 72ms/step - loss: 0.8181 - accuracy:
Epoch 15/20
23/23 [=====] - 2s 73ms/step - loss: 0.8119 - accuracy:
Epoch 16/20
23/23 [=====] - 2s 73ms/step - loss: 0.8139 - accuracy:
Epoch 17/20
23/23 [=====] - 2s 73ms/step - loss: 0.8050 - accuracy:
Epoch 18/20
23/23 [=====] - 2s 72ms/step - loss: 0.7986 - accuracy:
Epoch 19/20
23/23 [=====] - 2s 72ms/step - loss: 0.8057 - accuracy:
Epoch 20/20
23/23 [=====] - 2s 73ms/step - loss: 0.7962 - accuracy:
```


```
plot_acc(history_3, title = "Accuracy del modelo_3")
plot_loss(history_3, title = "Loss del modelo_3")
```



```
#NOTOCAR
```

```
metricas = modelo_3.evaluate(X_test_2, y_test_2)
print("Accuracy del modelo_3 en test:", metricas[1])
print("Loss del modelo_3 en test:", metricas[0])
```

```
63/63 [=====] - 0s 6ms/step - loss: 1.0635 - accuracy: 0.7390000224113464
Accuracy del modelo_3 en test: 0.7390000224113464
Loss del modelo_3 en test: 1.0635144710540771
```



[Colab paid products](#) - [Cancel contracts here](#)

✓ 0s completed at 9:09 AM

