Dado que el entrenamiento de redes neuronales es una tarea muy costosa, **se recomienda ejecutar el notebooks en Google Colab**, por supuesto también se puede ejecutar en local.

Al entrar en Google Colab bastará con hacer click en `upload` y subir este notebook. No olvide luego descargarlo en `File->Download .ipynb`

El examen deberá ser entregado con las celdas ejecutadas, si alguna celda no está ejecutadas no se contará.

El examen se divide en tres partes, con la puntuación que se indica a continuación. La puntuación máxima será 10.

- Actividad 1: Redes Densas: 4 pts
  - Correcta normalización: máximo de 0.25 pts
  - Cuestión 1: 1 pt
  - Cuestión 2: 1 pt
  - Cuestión 3: 0.5 pts
  - Cuestión 4: 0.25 pts
  - Cuestión 5: 0.25 pts
  - Cuestión 6: 0.25 pts
  - Cuestión 7: 0.25 pts
  - Cuestión 8: 0.25 pts
- Actividad 2: Redes Convolucionales: 4 pts
  - Cuestión 1: 1 pt
  - Cuestión 2: 1.5 pt
  - Cuestión 3: 0.5 pts
  - Cuestión 4: 0.25 pts
  - Cuestión 5: 0.25 pts
  - Cuestión 6: 0.25 pts
  - Cuestión 7: 0.25 pts
- Actividad 3: Redes Recurrentes: 2 pts
  - Cuestión 1: 0.5 pt
  - Cuestión 2: 0.5 pt
  - Cuestión 3: 0.5 pts
  - Cuestión 4: 0.25 pts
  - Cuestión 5: 0.25 pts

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Actividad 1: Redes Densas

Para esta primera actividad vamos a utilizar el boston housing dataset. Con el que trataremos de predecir el precio de una casa con 13 features.

Puntuación:

Normalizar las features correctamente (x_train, x_test): 0.1 pts , 0.25 si se normalizan con el Normalization layer de Keras. Ejemplo de uso: Introduction_to_RNN_Time_Series

```
tf.keras.layers.experimental.preprocessing.Normalization(
    axis=-1, dtype=None, mean=None, variance=None, **kwargs
)
```

- Correcta normalización: máximo de 0.25 pts
- Cuestión 1: 1 pt
- Cuestión 2: 1 pt
- Cuestión 3: 0.5 pts
- Cuestión 4: 0.25 pts
- Cuestión 5: 0.25 pts
- Cuestión 6: 0.25 pts
- Cuestión 7: 0.25 pts

- Cuestión 8: 0.25 pts

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.boston_housing.load_data(
    path='boston_housing.npz',
    test_split=0.2,
)
print('x_train, y_train shapes:', x_train.shape, y_train.shape)
print('x_test, y_test shapes:', x_train.shape, y_train.shape)
print('Some prices: ', y_train[:5])
```

```
x_train, y_train shapes: (404, 13) (404,)
x_test, y_test shapes: (404, 13) (404,)
Some prices:  [15.2 42.3 50.  21.1 17.7]
```

```
x_train[0].shape
```

```
(13,)
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
X_train_norm = scaler.fit_transform(x_train)
X_test_norm = scaler.transform(x_test)
```

Cuestión 1: Cree un modelo secuencial que contenga 4 capas ocultas(hidden layers), con más de 60 neuronas por capa, sin regularización y obtenga los resultados.

Puntuación:

- Obtener el modelo correcto: 0.8 pts
- Compilar el modelo: 0.1pts
- Acertar con la función de pérdida: 0.1 pts

```
model = tf.keras.models.Sequential()

...

model.add(layers.Dense(256, input_shape=(13,),activation='relu'))

model.add(layers.Dense(128, activation='relu'))

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(64, activation='relu'))

...

model.add(layers.Dense(1, activation='relu'))

model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_6 (Dense)             (None, 256)               3584      
                                                                 
 dense_7 (Dense)             (None, 128)               32896     
                                                                 
 dense_8 (Dense)             (None, 64)                8256      
                                                                 
 dense_9 (Dense)             (None, 64)                4160      
                                                                 
 dense_10 (Dense)            (None, 1)                 65        
                                                                 
=================================================================
Total params: 48,961
Trainable params: 48,961
Non-trainable params: 0
_________________________________________________________________
```

```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.MSE,
    metrics=['mae']
)
```

```
# No modifique el código
model.fit(X_train_norm,
          y_train,
          epochs=200,
          batch_size=32,
          validation_split=0.2,
          verbose=1)
```

```
Epoch 1/200
11/11 [==============================] - 1s 17ms/step - loss: 560.2420 - mae: 21.8059 - val_loss: 591.9921 - val_mae: 22.5034
Epoch 2/200
11/11 [==============================] - 0s 5ms/step - loss: 461.9830 - mae: 19.4196 - val_loss: 381.6768 - val_mae: 17.3743
Epoch 3/200
11/11 [==============================] - 0s 5ms/step - loss: 187.7039 - mae: 10.9258 - val_loss: 90.0556 - val_mae: 7.5805
Epoch 4/200
11/11 [==============================] - 0s 5ms/step - loss: 93.4587 - mae: 7.2038 - val_loss: 55.9299 - val_mae: 5.3365
Epoch 5/200
11/11 [==============================] - 0s 5ms/step - loss: 48.2980 - mae: 5.0406 - val_loss: 35.8225 - val_mae: 4.2103
Epoch 6/200
11/11 [==============================] - 0s 5ms/step - loss: 27.5530 - mae: 3.6790 - val_loss: 24.4114 - val_mae: 3.8954
Epoch 7/200
11/11 [==============================] - 0s 5ms/step - loss: 22.3604 - mae: 3.3362 - val_loss: 21.8584 - val_mae: 3.4554
Epoch 8/200
11/11 [==============================] - 0s 5ms/step - loss: 20.0612 - mae: 3.0917 - val_loss: 20.2172 - val_mae: 3.4111
Epoch 9/200
11/11 [==============================] - 0s 5ms/step - loss: 17.4692 - mae: 2.8599 - val_loss: 18.2939 - val_mae: 3.2171
Epoch 10/200
11/11 [==============================] - 0s 5ms/step - loss: 16.1106 - mae: 2.7347 - val_loss: 16.4439 - val_mae: 3.0531
Epoch 11/200
11/11 [==============================] - 0s 8ms/step - loss: 14.8797 - mae: 2.6680 - val_loss: 15.3459 - val_mae: 2.9365
Epoch 12/200
11/11 [==============================] - 0s 5ms/step - loss: 13.6001 - mae: 2.5250 - val_loss: 14.9128 - val_mae: 2.9339
Epoch 13/200
11/11 [==============================] - 0s 5ms/step - loss: 13.1032 - mae: 2.4378 - val_loss: 14.3535 - val_mae: 2.8608
Epoch 14/200
11/11 [==============================] - 0s 5ms/step - loss: 12.6057 - mae: 2.4360 - val_loss: 13.9661 - val_mae: 2.7263
Epoch 15/200
11/11 [==============================] - 0s 5ms/step - loss: 12.0855 - mae: 2.4617 - val_loss: 14.3241 - val_mae: 2.7790
Epoch 16/200
11/11 [==============================] - 0s 5ms/step - loss: 11.2584 - mae: 2.3584 - val_loss: 14.2921 - val_mae: 2.7972
Epoch 17/200
11/11 [==============================] - 0s 5ms/step - loss: 11.1040 - mae: 2.3028 - val_loss: 13.9707 - val_mae: 2.6854
Epoch 18/200
11/11 [==============================] - 0s 5ms/step - loss: 10.2793 - mae: 2.2459 - val_loss: 13.5402 - val_mae: 2.6349
Epoch 19/200
11/11 [==============================] - 0s 5ms/step - loss: 10.2882 - mae: 2.2497 - val_loss: 14.3294 - val_mae: 2.6884
Epoch 20/200
11/11 [==============================] - 0s 5ms/step - loss: 9.9799 - mae: 2.1636 - val_loss: 13.5782 - val_mae: 2.6455
Epoch 21/200
11/11 [==============================] - 0s 6ms/step - loss: 9.5869 - mae: 2.1661 - val_loss: 14.7767 - val_mae: 2.6940
Epoch 22/200
11/11 [==============================] - 0s 5ms/step - loss: 10.1699 - mae: 2.2025 - val_loss: 13.9083 - val_mae: 2.7039
Epoch 23/200
11/11 [==============================] - 0s 5ms/step - loss: 9.6103 - mae: 2.1965 - val_loss: 16.2349 - val_mae: 2.8549
Epoch 24/200
11/11 [==============================] - 0s 5ms/step - loss: 9.2530 - mae: 2.1626 - val_loss: 14.8138 - val_mae: 2.7305
Epoch 25/200
11/11 [==============================] - 0s 5ms/step - loss: 9.6875 - mae: 2.2428 - val_loss: 15.8740 - val_mae: 2.7638
Epoch 26/200
11/11 [==============================] - 0s 5ms/step - loss: 8.7917 - mae: 2.1177 - val_loss: 13.4661 - val_mae: 2.6019
Epoch 27/200
11/11 [==============================] - 0s 6ms/step - loss: 8.2779 - mae: 1.9983 - val_loss: 14.5372 - val_mae: 2.5924
Epoch 28/200
11/11 [==============================] - 0s 5ms/step - loss: 8.5252 - mae: 2.0919 - val_loss: 15.6971 - val_mae: 2.7555
Epoch 29/200
11/11 [==============================] - 0s 5ms/step - loss: 8.1310 - mae: 2.0009 - val_loss: 14.4563 - val_mae: 2.7159
```

```
# No modifique el código
results = model.evaluate(X_test_norm, y_test, verbose=1)
print('Test Loss: {}'.format(results))
```

```
4/4 [==============================] - 0s 3ms/step - loss: 16.1102 - mae: 2.6489
Test Loss: [16.11017417907715, 2.648906707763672]
```

Cuestión 2: Utilice el mismo modelo de la cuestión anterior pero añadiendo al menos dos técnicas distinas de regularización.

Ejemplos de regularización: Prevent_Overfitting.ipynb

Puntuación:

- Obtener el modelo con la regularización: 0.8 pts
- Obtener un `test loss` inferior al anterior: 0.2 pts

```
from tensorflow.keras import regularizers

kerner_regularizer_l2 = regularizers.l2(5e-4)
```

```
model = tf.keras.models.Sequential()

...

model.add(layers.Dense(256, input_shape=(13,), activation='relu',))

model.add(layers.Dense(128, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

model.add(layers.Dense(64, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

model.add(layers.Dense(64, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

...

model.add(layers.Dense(1, activation='relu'))

model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_11 (Dense)            (None, 256)               3584

 dense_12 (Dense)            (None, 128)               32896

 dropout_14 (Dropout)        (None, 128)               0

 dense_13 (Dense)            (None, 64)                8256

 dropout_15 (Dropout)        (None, 64)                0

 dense_14 (Dense)            (None, 64)                4160

 dropout_16 (Dropout)        (None, 64)                0

 dense_15 (Dense)            (None, 1)                 65

=================================================================
Total params: 48,961
Trainable params: 48,961
Non-trainable params: 0
_________________________________________________________________
```

```
# Compilación del modelo
# Código aquí
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.MSE,
    metrics=['mae']
)
```

```
batch_size=16
```

```
# No modifique el código
model.fit(X_train_norm,
          y_train,
          epochs=200,
          batch_size=batch_size,
          validation_split=0.2,
          verbose=1)
```

```
Epoch 1/200
21/21 [==============================] - 1s 12ms/step - loss: 431.3814 - mae: 18.5549 - val_loss: 187.3122 - val_mae: 11.3327
Epoch 2/200
21/21 [==============================] - 0s 5ms/step - loss: 89.7197 - mae: 7.1778 - val_loss: 64.0380 - val_mae: 5.8094
Epoch 3/200
21/21 [==============================] - 0s 5ms/step - loss: 44.3435 - mae: 4.7607 - val_loss: 25.4526 - val_mae: 3.7679
Epoch 4/200
21/21 [==============================] - 0s 5ms/step - loss: 32.6523 - mae: 4.0688 - val_loss: 21.9850 - val_mae: 3.5821
Epoch 5/200
21/21 [==============================] - 0s 5ms/step - loss: 23.8592 - mae: 3.5551 - val_loss: 21.2203 - val_mae: 3.3484
Epoch 6/200
21/21 [==============================] - 0s 5ms/step - loss: 28.5719 - mae: 3.9083 - val_loss: 22.4220 - val_mae: 3.4998
Epoch 7/200
21/21 [==============================] - 0s 5ms/step - loss: 20.1374 - mae: 3.2297 - val_loss: 15.5656 - val_mae: 2.9175
```

```
Epoch 8/200
21/21 [==============================] - 0s 5ms/step - loss: 18.2076 - mae: 3.1737 - val_loss: 14.8446 - val_mae: 2.8775
Epoch 9/200
21/21 [==============================] - 0s 5ms/step - loss: 17.1698 - mae: 3.0145 - val_loss: 14.2335 - val_mae: 2.8790
Epoch 10/200
21/21 [==============================] - 0s 5ms/step - loss: 18.2228 - mae: 3.1052 - val_loss: 14.9696 - val_mae: 2.9913
Epoch 11/200
21/21 [==============================] - 0s 5ms/step - loss: 16.6675 - mae: 2.9512 - val_loss: 15.4334 - val_mae: 2.9794
Epoch 12/200
21/21 [==============================] - 0s 5ms/step - loss: 16.7696 - mae: 3.0856 - val_loss: 19.1354 - val_mae: 3.2352
Epoch 13/200
21/21 [==============================] - 0s 4ms/step - loss: 16.1223 - mae: 3.0364 - val_loss: 16.2442 - val_mae: 2.8717
Epoch 14/200
21/21 [==============================] - 0s 5ms/step - loss: 15.7036 - mae: 3.0696 - val_loss: 17.0772 - val_mae: 3.2583
Epoch 15/200
21/21 [==============================] - 0s 5ms/step - loss: 14.7487 - mae: 2.9807 - val_loss: 14.7731 - val_mae: 2.7980
Epoch 16/200
21/21 [==============================] - 0s 5ms/step - loss: 14.4759 - mae: 2.8602 - val_loss: 16.2116 - val_mae: 2.8621
Epoch 17/200
21/21 [==============================] - 0s 5ms/step - loss: 15.8817 - mae: 2.8561 - val_loss: 16.7018 - val_mae: 2.9646
Epoch 18/200
21/21 [==============================] - 0s 5ms/step - loss: 16.9115 - mae: 3.1186 - val_loss: 16.1967 - val_mae: 2.9748
Epoch 19/200
21/21 [==============================] - 0s 4ms/step - loss: 14.4128 - mae: 2.7717 - val_loss: 13.6277 - val_mae: 2.8036
Epoch 20/200
21/21 [==============================] - 0s 4ms/step - loss: 13.9301 - mae: 2.7594 - val_loss: 15.4364 - val_mae: 2.7225
Epoch 21/200
21/21 [==============================] - 0s 5ms/step - loss: 14.8429 - mae: 2.9142 - val_loss: 13.4206 - val_mae: 2.7663
Epoch 22/200
21/21 [==============================] - 0s 5ms/step - loss: 12.9756 - mae: 2.7145 - val_loss: 18.3395 - val_mae: 2.9300
Epoch 23/200
21/21 [==============================] - 0s 5ms/step - loss: 13.8112 - mae: 2.7866 - val_loss: 15.5638 - val_mae: 2.8707
Epoch 24/200
21/21 [==============================] - 0s 4ms/step - loss: 13.3696 - mae: 2.8228 - val_loss: 15.7864 - val_mae: 2.8396
Epoch 25/200
21/21 [==============================] - 0s 5ms/step - loss: 13.2526 - mae: 2.6572 - val_loss: 12.8364 - val_mae: 2.5575
Epoch 26/200
21/21 [==============================] - 0s 5ms/step - loss: 12.6971 - mae: 2.6355 - val_loss: 12.0348 - val_mae: 2.5882
Epoch 27/200
21/21 [==============================] - 0s 5ms/step - loss: 9.7950 - mae: 2.3343 - val_loss: 14.3669 - val_mae: 2.6749
Epoch 28/200
21/21 [==============================] - 0s 5ms/step - loss: 11.7892 - mae: 2.5935 - val_loss: 17.2355 - val_mae: 2.7550
Epoch 29/200
21/21 [==============================] - 0s 5ms/step - loss: 11.9110 - mae: 2.5476 - val_loss: 14.5795 - val_mae: 2.7398
```

```python
# No modifique el código
results = model.evaluate(X_test_norm, y_test, verbose=1)
print('Test Loss: {}'.format(results))
```

```
4/4 [==============================] - 0s 3ms/step - loss: 15.9122 - mae: 2.4922
Test Loss: [15.9121732711792, 2.4921603202819824]
```

Cuestión 3: Utilice el mismo modelo de la cuestión anterior pero añadiendo un callback de early stopping. Obtenga un test loss inferior al del modelo anterior

```python
# Código aquí
model = tf.keras.models.Sequential()

...

model.add(layers.Dense(256, input_shape=(13,), activation='relu',))

model.add(layers.Dense(128, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

model.add(layers.Dense(64, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

model.add(layers.Dense(64, kernel_regularizer=kerner_regularizer_l2, activation='relu'))
model.add(layers.Dropout(0.1))

...

model.add(layers.Dense(1, activation='relu'))

model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 256)               3584
```

```
 dense_17 (Dense)            (None, 128)               32896

 dropout_17 (Dropout)        (None, 128)               0

 dense_18 (Dense)            (None, 64)                8256

 dropout_18 (Dropout)        (None, 64)                0

 dense_19 (Dense)            (None, 64)                4160

 dropout_19 (Dropout)        (None, 64)                0

 dense_20 (Dense)            (None, 1)                 65

=================================================================
Total params: 48,961
Trainable params: 48,961
Non-trainable params: 0
_________________________________________________________________
```

```
# Compilación del modelo
# Código aquí
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.MSE,
    metrics=['mae']
)
```

```
## definir el early stopping callback
# Código aquí
es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=6,
    verbose=1)

model.fit(X_train_norm,
          y_train,
          epochs=200,
          batch_size=16,
          validation_split=0.2,
          verbose=1,
          callbacks=[es_callback]) # Código aquí
```

```
Epoch 1/200
21/21 [==============================] - 0s 6ms/step - loss: 7.3778 - mae: 2.0648 - val_loss: 10.6531 - val_mae: 2.3131
Epoch 2/200
21/21 [==============================] - 0s 5ms/step - loss: 10.2502 - mae: 2.3245 - val_loss: 13.6890 - val_mae: 2.6774
Epoch 3/200
21/21 [==============================] - 0s 4ms/step - loss: 7.4145 - mae: 2.0011 - val_loss: 11.6405 - val_mae: 2.3657
Epoch 4/200
21/21 [==============================] - 0s 4ms/step - loss: 9.7595 - mae: 2.3323 - val_loss: 14.3282 - val_mae: 2.8847
Epoch 5/200
21/21 [==============================] - 0s 4ms/step - loss: 9.9505 - mae: 2.4508 - val_loss: 8.7137 - val_mae: 2.2241
Epoch 6/200
21/21 [==============================] - 0s 5ms/step - loss: 8.2045 - mae: 2.1067 - val_loss: 9.1649 - val_mae: 2.1637
Epoch 7/200
21/21 [==============================] - 0s 5ms/step - loss: 8.5336 - mae: 2.1866 - val_loss: 10.4541 - val_mae: 2.2584
Epoch 8/200
21/21 [==============================] - 0s 5ms/step - loss: 9.3823 - mae: 2.2547 - val_loss: 13.8624 - val_mae: 2.4194
Epoch 9/200
21/21 [==============================] - 0s 4ms/step - loss: 9.2190 - mae: 2.2191 - val_loss: 11.5352 - val_mae: 2.4644
Epoch 10/200
21/21 [==============================] - 0s 5ms/step - loss: 7.6496 - mae: 2.1016 - val_loss: 9.7246 - val_mae: 2.2289
Epoch 11/200
21/21 [==============================] - 0s 5ms/step - loss: 7.6775 - mae: 2.0905 - val_loss: 8.4723 - val_mae: 2.1695
Epoch 12/200
21/21 [==============================] - 0s 5ms/step - loss: 8.1967 - mae: 2.1964 - val_loss: 10.0071 - val_mae: 2.2307
Epoch 13/200
21/21 [==============================] - 0s 4ms/step - loss: 7.0131 - mae: 1.9986 - val_loss: 9.1251 - val_mae: 2.2318
Epoch 14/200
21/21 [==============================] - 0s 4ms/step - loss: 7.3203 - mae: 2.0335 - val_loss: 9.7163 - val_mae: 2.2148
Epoch 15/200
21/21 [==============================] - 0s 4ms/step - loss: 9.7415 - mae: 2.3597 - val_loss: 11.0387 - val_mae: 2.4097
Epoch 16/200
21/21 [==============================] - 0s 6ms/step - loss: 8.1783 - mae: 2.1895 - val_loss: 9.1371 - val_mae: 2.2304
Epoch 17/200
21/21 [==============================] - 0s 4ms/step - loss: 9.6337 - mae: 2.3421 - val_loss: 11.9233 - val_mae: 2.5812
Epoch 17: early stopping
<keras.callbacks.History at 0x7f184018abe0>
```

```
# No modifique el código
results = model.evaluate(X_test_norm, y_test, verbose=1)
print('Test Loss: {}'.format(results))
```

```
4/4 [==============================] - 0s 4ms/step - loss: 12.6468 - mae: 2.5208
Test Loss: [12.646807670593262, 2.520766019821167]
```

Cuestión 4: ¿Podría haberse usado otra función de activación de la neurona de salida? En caso afirmativo especifíquela.

Si que se podría usar otra función de activación. En este problema, donde realizamos una regresión, se puede usar la función Linear o la ReLu.

La ReLu se puede usar en la capa de salida si la regresión tiene un resultado positivo y la Linear solo se usa en regresiones y va de menos infinito a más infinito..

Cuestión 5: ¿Qué es lo que una neurona calcula?

a) Una función de activación seguida de una suma ponderada de las entradas.

b) Una suma ponderada de las entradas seguida de una función de activación.

c) Una función de pérdida, definida sobre el target.

d) Ninguna de las anteriores es correcta

b) Una suma ponderada de las entradas seguida de una función de activación.

Cuestión 6: ¿Cuál de estas funciones de activación no debería usarse en una capa oculta (hidden layer)?

a) `sigmoid`

b) `tanh`

c) `relu`

d) `linear`

d) linear

Cuestión 7: ¿Cuál de estas técnicas es efectiva para combatir el overfitting en una red con varias capas ocultas? Ponga todas las que lo sean.

a) Dropout

b) Regularización L2.

c) Aumentar el tamaño del test set.

d) Aumentar el tamaño del validation set.

e) Reducir el número de capas de la red.

f) Data augmentation.

a) Dropout

b) Regularización L2

e) Reducir el número de capas de la red

f) Data augmentation

Cuestión 8: Supongamos que queremos entrenar una red para un problema de clasificación de imágenes con las siguientes clases: {'perro','gato','persona'}. ¿Cuántas neuronas y que función de activación debería tener la capa de salida? ¿Qué función de pérdida (loss function) debería usarse?

Las neuronas en la capa de salida va a tener que ser 3, ya que tu intención es clasificar 3 elementos.

La función de activación de la capa de salida tendrá que ser una softmax.

La función de pérdida denería ser una Categorical Cross-Entropy.

Actividad 2: Redes Convolucionales

Vamos a usar el dataset cifar-10, que son 60000 imágenes de 32x32 a color con 10 clases diferentes. Para realizar mejor la práctica puede consultar Introduction_to_CNN.ipynb.

Puntuación:

- Cuestión 1: 1 pt
- Cuestión 2: 1.5 pt
- Cuestión 3: 0.5 pts
- Cuestión 4: 0.25 pts
- Cuestión 5: 0.25 pts
- Cuestión 6: 0.25 pts
- Cuestión 7: 0.25 pts

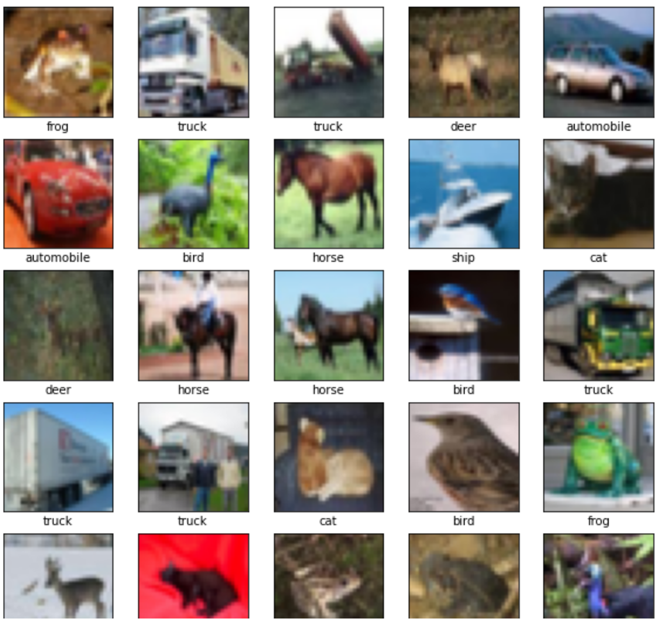
Puede normalizar las imágenes al principio o usar la capa Rescaling:

```
tf.keras.layers.experimental.preprocessing.Rescaling(
    scale, offset=0.0, name=None, **kwargs
)
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
y_train = y_train.flatten()
y_test = y_test.flatten()
```

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i])
    plt.xlabel(class_names[y_train[i]])
plt.show()
```




```
print('x_train, y_train shapes:', x_train.shape, y_train.shape)
print('x_test, y_test shapes:', x_test.shape, y_test.shape)
```

```
x_train, y_train shapes: (50000, 32, 32, 3) (50000,)
x_test, y_test shapes: (10000, 32, 32, 3) (10000,)
```

Cuestión 1: Cree una red convolucional con la API funcional con al menos dos capas convolucionales y al menos dos capas de pooling. Utilize sólo [Average Pooling] y no añada ninguna regularización.

```
inputs = tf.keras.Input(shape=(32,32,3), name='input')
reescaling = layers.experimental.preprocessing.Rescaling(1. / 255)(inputs)

# Conv Layer 1
conv_1 = layers.Conv2D(512, 3, padding='valid', activation='relu',
                       name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(3,3), name='pool_1')(conv_1)

# Conv Layer 2
conv_2 = layers.Conv2D(256, 3, padding='valid', activation='relu',
                       name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(3,3), name='pool_2')(conv_2)

# Fully-connected
# Flattening
flat = layers.Flatten(name='flatten')(pool_2)
dense = layers.Dense(64, activation='relu', name='dense')(flat)
outputs = layers.Dense(10, activation='softmax', name='output')(dense)

model = keras.Model(inputs=inputs, outputs=outputs, name='cnn_example')
```

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

```
history = model.fit(x_train, y_train, epochs=25, batch_size=64,
                    validation_split=0.15)
```

```
Epoch 1/25
665/665 [==============================] - 12s 17ms/step - loss: 1.7982 - accuracy: 0.3417 - val_loss: 1.5972 - val_accuracy: 0.409
Epoch 2/25
665/665 [==============================] - 12s 18ms/step - loss: 1.4750 - accuracy: 0.4659 - val_loss: 1.4176 - val_accuracy: 0.484
Epoch 3/25
665/665 [==============================] - 12s 18ms/step - loss: 1.3463 - accuracy: 0.5170 - val_loss: 1.3148 - val_accuracy: 0.533
Epoch 4/25
665/665 [==============================] - 14s 21ms/step - loss: 1.2495 - accuracy: 0.5556 - val_loss: 1.2490 - val_accuracy: 0.559
Epoch 5/25
665/665 [==============================] - 11s 17ms/step - loss: 1.1768 - accuracy: 0.5860 - val_loss: 1.1797 - val_accuracy: 0.591
Epoch 6/25
665/665 [==============================] - 11s 16ms/step - loss: 1.1171 - accuracy: 0.6089 - val_loss: 1.2110 - val_accuracy: 0.579
Epoch 7/25
665/665 [==============================] - 11s 17ms/step - loss: 1.0723 - accuracy: 0.6262 - val_loss: 1.0764 - val_accuracy: 0.626
Epoch 8/25
665/665 [==============================] - 11s 17ms/step - loss: 1.0317 - accuracy: 0.6392 - val_loss: 1.0625 - val_accuracy: 0.634
Epoch 9/25
665/665 [==============================] - 11s 17ms/step - loss: 0.9973 - accuracy: 0.6547 - val_loss: 1.0564 - val_accuracy: 0.636
Epoch 10/25
665/665 [==============================] - 11s 16ms/step - loss: 0.9658 - accuracy: 0.6644 - val_loss: 1.0406 - val_accuracy: 0.643
Epoch 11/25
665/665 [==============================] - 11s 17ms/step - loss: 0.9353 - accuracy: 0.6748 - val_loss: 0.9979 - val_accuracy: 0.658
Epoch 12/25
665/665 [==============================] - 11s 16ms/step - loss: 0.9062 - accuracy: 0.6868 - val_loss: 1.0170 - val_accuracy: 0.657
Epoch 13/25
665/665 [==============================] - 11s 17ms/step - loss: 0.8893 - accuracy: 0.6936 - val_loss: 0.9905 - val_accuracy: 0.662
Epoch 14/25
665/665 [==============================] - 11s 16ms/step - loss: 0.8638 - accuracy: 0.7015 - val_loss: 0.9903 - val_accuracy: 0.661
Epoch 15/25
665/665 [==============================] - 11s 16ms/step - loss: 0.8397 - accuracy: 0.7098 - val_loss: 1.0056 - val_accuracy: 0.660
Epoch 16/25
665/665 [==============================] - 12s 17ms/step - loss: 0.8176 - accuracy: 0.7173 - val_loss: 0.9676 - val_accuracy: 0.672
Epoch 17/25
665/665 [==============================] - 11s 17ms/step - loss: 0.8022 - accuracy: 0.7231 - val_loss: 0.9668 - val_accuracy: 0.671
Epoch 18/25
665/665 [==============================] - 11s 16ms/step - loss: 0.7792 - accuracy: 0.7305 - val_loss: 0.9414 - val_accuracy: 0.682
Epoch 19/25
665/665 [==============================] - 11s 16ms/step - loss: 0.7648 - accuracy: 0.7350 - val_loss: 0.9334 - val_accuracy: 0.687
Epoch 20/25
665/665 [==============================] - 11s 17ms/step - loss: 0.7442 - accuracy: 0.7418 - val_loss: 1.0212 - val_accuracy: 0.660
Epoch 21/25
665/665 [==============================] - 11s 16ms/step - loss: 0.7300 - accuracy: 0.7470 - val_loss: 0.9703 - val_accuracy: 0.674
Epoch 22/25
665/665 [==============================] - 11s 16ms/step - loss: 0.7109 - accuracy: 0.7542 - val_loss: 0.9382 - val_accuracy: 0.683
Epoch 23/25
```

```
665/665 [==============================] - 11s 17ms/step - loss: 0.6925 - accuracy: 0.7602 - val_loss: 0.9327 - val_accuracy: 0.687
Epoch 24/25
665/665 [==============================] - 11s 16ms/step - loss: 0.6802 - accuracy: 0.7639 - val_loss: 1.0052 - val_accuracy: 0.673
Epoch 25/25
665/665 [==============================] - 11s 16ms/step - loss: 0.6661 - accuracy: 0.7694 - val_loss: 0.9936 - val_accuracy: 0.679
```

```
results = model.evaluate(x_test, y_test, verbose=0, batch_size=1000)
print('Test Loss: {}'.format(results[0]))
print('Test Accuracy: {}'.format(results[1]))
```

```
Test Loss: 1.012425422668457
Test Accuracy: 0.6723999977111816
```

Cuestión 2: Cree un modelo con la API funcional con un máximo de 2 capas convolucionales y un máximo de 2 capas de pooling. Utilize [Max Pooling](#) o [Average Pooling](#) y añada la regularización que quiera. Debe obtener un `Test accuracy > 0.68`

```
inputs = tf.keras.Input(shape=(32,32,3), name='input')
reescaling = layers.experimental.preprocessing.Rescaling(1. / 255)(inputs)

# Conv Layer 1
conv_1 = layers.Conv2D(512, 3, padding='valid', activation='relu',
                       name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(3,3), name='pool_1')(conv_1)
pool_1 = layers.Dropout(0.4)(pool_1)

# Conv Layer 2
conv_2 = layers.Conv2D(512, 3, padding='valid', activation='relu',
                       name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(3,3), name='pool_2')(conv_2)
pool_2 = layers.Dropout(0.4)(pool_2)

# Fully-connected
# Flattening
flat = layers.Flatten(name='flatten')(pool_2)
dense = layers.Dense(64, activation='relu', name='dense')(flat)
outputs = layers.Dense(10, activation='softmax', name='output')(dense)

model = keras.Model(inputs=inputs, outputs=outputs, name='cnn_example')
```

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

```
es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1
)

history = model.fit(x_train, y_train, epochs=100, batch_size=64,
                    validation_split=0.15, callbacks=[es_callback])
```

```
Epoch 1/100
665/665 [==============================] - 15s 21ms/step - loss: 1.8394 - accuracy: 0.3148 - val_loss: 1.5667 - val_accuracy: 0.4
Epoch 2/100
665/665 [==============================] - 14s 21ms/step - loss: 1.5138 - accuracy: 0.4484 - val_loss: 1.4399 - val_accuracy: 0.4
Epoch 3/100
665/665 [==============================] - 14s 22ms/step - loss: 1.3865 - accuracy: 0.5007 - val_loss: 1.2796 - val_accuracy: 0.5
Epoch 4/100
665/665 [==============================] - 14s 22ms/step - loss: 1.3048 - accuracy: 0.5375 - val_loss: 1.2289 - val_accuracy: 0.5
Epoch 5/100
665/665 [==============================] - 14s 21ms/step - loss: 1.2445 - accuracy: 0.5571 - val_loss: 1.2027 - val_accuracy: 0.5
Epoch 6/100
665/665 [==============================] - 14s 21ms/step - loss: 1.1956 - accuracy: 0.5780 - val_loss: 1.1564 - val_accuracy: 0.5
Epoch 7/100
665/665 [==============================] - 14s 22ms/step - loss: 1.1568 - accuracy: 0.5945 - val_loss: 1.1148 - val_accuracy: 0.6
Epoch 8/100
665/665 [==============================] - 14s 22ms/step - loss: 1.1279 - accuracy: 0.6036 - val_loss: 1.0558 - val_accuracy: 0.6
Epoch 9/100
665/665 [==============================] - 15s 22ms/step - loss: 1.0915 - accuracy: 0.6188 - val_loss: 1.0521 - val_accuracy: 0.6
Epoch 10/100
665/665 [==============================] - 14s 22ms/step - loss: 1.0684 - accuracy: 0.6248 - val_loss: 1.0329 - val_accuracy: 0.6
Epoch 11/100
665/665 [==============================] - 14s 21ms/step - loss: 1.0416 - accuracy: 0.6376 - val_loss: 1.0179 - val_accuracy: 0.6
Epoch 12/100
665/665 [==============================] - 14s 21ms/step - loss: 1.0175 - accuracy: 0.6458 - val_loss: 0.9728 - val_accuracy: 0.6
```

```
Epoch 13/100
665/665 [==============================] - 14s 21ms/step - loss: 1.0011 - accuracy: 0.6509 - val_loss: 0.9881 - val_accuracy: 0.6
Epoch 14/100
665/665 [==============================] - 14s 21ms/step - loss: 0.9784 - accuracy: 0.6580 - val_loss: 0.9372 - val_accuracy: 0.6
Epoch 15/100
665/665 [==============================] - 14s 21ms/step - loss: 0.9565 - accuracy: 0.6657 - val_loss: 0.9455 - val_accuracy: 0.6
Epoch 16/100
665/665 [==============================] - 14s 21ms/step - loss: 0.9418 - accuracy: 0.6714 - val_loss: 0.9162 - val_accuracy: 0.6
Epoch 17/100
665/665 [==============================] - 14s 21ms/step - loss: 0.9228 - accuracy: 0.6778 - val_loss: 0.9014 - val_accuracy: 0.6
Epoch 18/100
665/665 [==============================] - 14s 21ms/step - loss: 0.9113 - accuracy: 0.6802 - val_loss: 0.8912 - val_accuracy: 0.6
Epoch 19/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8918 - accuracy: 0.6903 - val_loss: 0.9043 - val_accuracy: 0.6
Epoch 20/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8829 - accuracy: 0.6927 - val_loss: 0.9200 - val_accuracy: 0.6
Epoch 21/100
665/665 [==============================] - 14s 21ms/step - loss: 0.8680 - accuracy: 0.6969 - val_loss: 0.8763 - val_accuracy: 0.7
Epoch 22/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8513 - accuracy: 0.6993 - val_loss: 0.8939 - val_accuracy: 0.6
Epoch 23/100
665/665 [==============================] - 15s 22ms/step - loss: 0.8413 - accuracy: 0.7064 - val_loss: 0.8793 - val_accuracy: 0.6
Epoch 24/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8277 - accuracy: 0.7105 - val_loss: 0.8653 - val_accuracy: 0.7
Epoch 25/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8263 - accuracy: 0.7110 - val_loss: 0.8508 - val_accuracy: 0.7
Epoch 26/100
665/665 [==============================] - 14s 21ms/step - loss: 0.8054 - accuracy: 0.7193 - val_loss: 0.8445 - val_accuracy: 0.7
Epoch 27/100
665/665 [==============================] - 14s 22ms/step - loss: 0.8002 - accuracy: 0.7205 - val_loss: 0.8421 - val_accuracy: 0.7
Epoch 28/100
```

```
results = model.evaluate(x_test, y_test, verbose=0, batch_size=1000)
print('Test Loss: {}'.format(results[0]))
print('Test Accuracy: {}'.format(results[1]))
```

```
Test Loss: 0.7985534071922302
Test Accuracy: 0.7347999811172485
```

Cuestión 3: Añada data augmentation al principio del modelo

```
data_augmentation = keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip(),
    layers.experimental.preprocessing.RandomRotation(0.25),
    layers.experimental.preprocessing.RandomZoom(0.25),
  ]
)
```

```
inputs = tf.keras.Input(shape=(32,32, 3), name='input')
data_aug= data_augmentation(inputs)
reescaling = layers.experimental.preprocessing.Rescaling(1. / 255)(data_aug)

# Conv Layer 1
conv_1 = layers.Conv2D(512, 3, padding='valid', activation='relu',
                       name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(3,3), name='pool_1')(conv_1)
pool_1 = layers.Dropout(0.4)(pool_1)

# Conv Layer 2
conv_2 = layers.Conv2D(512, 3, padding='valid', activation='relu',
                       name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(3,3), name='pool_2')(conv_2)
pool_2 = layers.Dropout(0.4)(pool_2)

# Fully-connected
# Flattening
flat = layers.Flatten(name='flatten')(pool_2)
dense = layers.Dense(64, activation='relu', name='dense')(flat)
outputs = layers.Dense(10, activation='softmax', name='output')(dense)

model = keras.Model(inputs=inputs, outputs=outputs, name='cnn_example')
```

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

```
es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
```

```
    patience=5,
    verbose=1
)

history = model.fit(x_train, y_train, epochs=100, batch_size=64,
                    validation_split=0.15, callbacks=[es_callback])
```

```
Epoch 1/100
665/665 [==============================] - 49s 72ms/step - loss: 2.0170 - accuracy: 0.2510 - val_loss: 1.9527 - val_accuracy: 0.2
Epoch 2/100
665/665 [==============================] - 45s 67ms/step - loss: 1.8704 - accuracy: 0.3200 - val_loss: 1.7817 - val_accuracy: 0.3
Epoch 3/100
665/665 [==============================] - 46s 69ms/step - loss: 1.8038 - accuracy: 0.3458 - val_loss: 1.7158 - val_accuracy: 0.3
Epoch 4/100
665/665 [==============================] - 45s 67ms/step - loss: 1.7556 - accuracy: 0.3614 - val_loss: 1.7340 - val_accuracy: 0.3
Epoch 5/100
665/665 [==============================] - 53s 80ms/step - loss: 1.7239 - accuracy: 0.3768 - val_loss: 1.6381 - val_accuracy: 0.4
Epoch 6/100
665/665 [==============================] - 46s 70ms/step - loss: 1.6970 - accuracy: 0.3853 - val_loss: 1.6133 - val_accuracy: 0.4
Epoch 7/100
665/665 [==============================] - 47s 71ms/step - loss: 1.6753 - accuracy: 0.3953 - val_loss: 1.6063 - val_accuracy: 0.4
Epoch 8/100
665/665 [==============================] - 48s 72ms/step - loss: 1.6588 - accuracy: 0.4028 - val_loss: 1.5964 - val_accuracy: 0.4
Epoch 9/100
665/665 [==============================] - 45s 68ms/step - loss: 1.6390 - accuracy: 0.4106 - val_loss: 1.5833 - val_accuracy: 0.4
Epoch 10/100
665/665 [==============================] - 44s 67ms/step - loss: 1.6252 - accuracy: 0.4127 - val_loss: 1.5571 - val_accuracy: 0.4
Epoch 11/100
665/665 [==============================] - 45s 67ms/step - loss: 1.6103 - accuracy: 0.4235 - val_loss: 1.5371 - val_accuracy: 0.4
Epoch 12/100
665/665 [==============================] - 44s 67ms/step - loss: 1.5991 - accuracy: 0.4252 - val_loss: 1.5438 - val_accuracy: 0.4
Epoch 13/100
665/665 [==============================] - 45s 68ms/step - loss: 1.5855 - accuracy: 0.4325 - val_loss: 1.4808 - val_accuracy: 0.4
Epoch 14/100
665/665 [==============================] - 47s 71ms/step - loss: 1.5765 - accuracy: 0.4336 - val_loss: 1.5551 - val_accuracy: 0.4
Epoch 15/100
665/665 [==============================] - 45s 67ms/step - loss: 1.5638 - accuracy: 0.4374 - val_loss: 1.4757 - val_accuracy: 0.4
Epoch 16/100
665/665 [==============================] - 44s 66ms/step - loss: 1.5519 - accuracy: 0.4440 - val_loss: 1.4848 - val_accuracy: 0.4
Epoch 17/100
665/665 [==============================] - 44s 66ms/step - loss: 1.5407 - accuracy: 0.4460 - val_loss: 1.4507 - val_accuracy: 0.4
Epoch 18/100
665/665 [==============================] - 44s 66ms/step - loss: 1.5323 - accuracy: 0.4491 - val_loss: 1.4258 - val_accuracy: 0.4
Epoch 19/100
665/665 [==============================] - 44s 65ms/step - loss: 1.5181 - accuracy: 0.4577 - val_loss: 1.4441 - val_accuracy: 0.4
Epoch 20/100
665/665 [==============================] - 45s 67ms/step - loss: 1.5162 - accuracy: 0.4564 - val_loss: 1.4478 - val_accuracy: 0.4
Epoch 21/100
665/665 [==============================] - 46s 69ms/step - loss: 1.5078 - accuracy: 0.4572 - val_loss: 1.4296 - val_accuracy: 0.4
Epoch 22/100
665/665 [==============================] - 43s 65ms/step - loss: 1.5001 - accuracy: 0.4613 - val_loss: 1.4198 - val_accuracy: 0.5
Epoch 23/100
665/665 [==============================] - 43s 65ms/step - loss: 1.4997 - accuracy: 0.4626 - val_loss: 1.3803 - val_accuracy: 0.5
Epoch 24/100
665/665 [==============================] - 43s 65ms/step - loss: 1.4865 - accuracy: 0.4701 - val_loss: 1.3996 - val_accuracy: 0.5
Epoch 25/100
665/665 [==============================] - 43s 65ms/step - loss: 1.4798 - accuracy: 0.4709 - val_loss: 1.4162 - val_accuracy: 0.4
Epoch 26/100
665/665 [==============================] - 44s 66ms/step - loss: 1.4742 - accuracy: 0.4739 - val_loss: 1.4311 - val_accuracy: 0.4
Epoch 27/100
665/665 [==============================] - 44s 66ms/step - loss: 1.4657 - accuracy: 0.4738 - val_loss: 1.4141 - val_accuracy: 0.5
Epoch 28/100
665/665 [==============================] - 43s 65ms/step - loss: 1.4619 - accuracy: 0.4754 - val_loss: 1.3760 - val_accuracy: 0.5
Epoch 29/100
```

```
results = model.evaluate(x_test, y_test, verbose=0, batch_size=1000)
print('Test Loss: {}'.format(results[0]))
print('Test Accuracy: {}'.format(results[1]))
```

```
Test Loss: 1.3555670976638794
Test Accuracy: 0.5159000158309937
```

Cuestión 4: Cree el mismo modelo de manera secuencial. No es necesario compilar ni entrenar el modelo

```
img_shape = (32,32,3)
model_seq = tf.keras.models.Sequential()

# Código aquí

model_seq.add(layers.Conv2D(filters=512, kernel_size=(3,3), input_shape=img_shape, activation='relu'))
model_seq.add(layers.AveragePooling2D(pool_size=(3,3), strides=(2, 2)))
```

```
model_seq.add(layers.Dropout(0.4))

model_seq.add(layers.Conv2D(filters=512, kernel_size=(3,3),activation='relu'))
model_seq.add(layers.AveragePooling2D(pool_size=(3,3), strides=(2, 2)))
model_seq.add(layers.Dropout(0.4))

...

model_seq.add(layers.Flatten())
model_seq.add(layers.Dense(64, activation='relu'))
model_seq.add(layers.Dense(10, activation = 'softmax'))

model_seq.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_9 (Conv2D)           (None, 30, 30, 512)       14336     
                                                                 
 average_pooling2d_6 (Averag  (None, 14, 14, 512)      0         
 ePooling2D)                                                     
                                                                 
 dropout_26 (Dropout)        (None, 14, 14, 512)       0         
                                                                 
 conv2d_10 (Conv2D)          (None, 12, 12, 512)       2359808   
                                                                 
 average_pooling2d_7 (Averag  (None, 5, 5, 512)        0         
 ePooling2D)                                                     
                                                                 
 dropout_27 (Dropout)        (None, 5, 5, 512)         0         
                                                                 
 flatten_3 (Flatten)         (None, 12800)             0         
                                                                 
 dense_21 (Dense)            (None, 64)                819264    
                                                                 
 dense_22 (Dense)            (None, 10)                650       
                                                                 
=================================================================
Total params: 3,194,058
Trainable params: 3,194,058
Non-trainable params: 0
_________________________________________________________________
```

Cuestión 5: Si tenenemos una una imagen de entrada de 300 x 300 a color (RGB) y queremos usar una red densa. Si la primera capa oculta tiene 100 neuronas, ¿Cuántos parámetros tendrá esa capa (sin incluir el bias) ?

Tiene 27000000 de parámetros.

Cuestión 6 Ponga las verdaderas ventajas de las redes convolucionales respecto a las densas

a) Reducen el número total de parámetros, reduciendo así el overfitting.

b) Permiten utilizar una misma 'función' en varias localizaciones de la imagen de entrada, en lugar de aprender una función diferente para cada pixel.

c) Permiten el uso del transfer learning.

d) Generalmente son menos profundas, lo que facilita su entrenamiento.

a) Reducen el número total de parámetros, reduciendo así el overfitting

b) Permiten utilizar una misma 'función' en varias localizaciones de la imagen de entrada, en lugar de aprender una función diferente para cada pixel.

Cuestión 7: Para el procesamiento de series temporales las redes convolucionales no son efectivas, habrá que usar redes recurrentes.

- **Verdadero**
- **Falso**

Falso, las redes convolucionales también tienen muy buenos resultados con las series temporales.

Actividad 3: Redes Recurrentes

- Cuestión 1: 0.5 pt
- Cuestión 2: 0.5 pt
- Cuestión 3: 0.5 pts
- Cuestión 4: 0.25 pts
- Cuestión 5: 0.25 pts

Vamos a usar un dataset de las temperaturas mínimas diarias en Melbourne. La tarea será la de predecir la temperatura mínima en dos días. Puedes usar técnicas de series temporales vistas en otras asignaturas, pero no es necesario.

```
dataset_url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-min-temperatures.csv'
data_dir = tf.keras.utils.get_file('daily-min-temperatures.csv', origin=dataset_url)
```

```
Downloading data from https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-min-temperatures.csv
67921/67921 [==============================] - 0s 0us/step
```

```
df = pd.read_csv(data_dir, parse_dates=['Date'])
df.head()
```

	Date	Temp
0	1981-01-01	20.7
1	1981-01-02	17.9
2	1981-01-03	18.8
3	1981-01-04	14.6

```
temperatures = df['Temp'].values
print('number of samples:', len(temperatures))
train_data = temperatures[:3000]
test_data = temperatures[3000:]
print('number of train samples:', len(train_data))
print('number of test samples:', len(test_data))
print('firsts trainn samples:', train_data[:10])
```

```
number of samples: 3650
number of train samples: 3000
number of test samples: 650
firsts trainn samples: [20.7 17.9 18.8 14.6 15.8 15.8 15.8 17.4 21.8 20. ]
```

```
train_data
```

```
array([20.7, 17.9, 18.8, ..., 15. , 17.1, 17.3])
```

Cuestión 1: Convierta `train_data` y `test_data` en ventanas de tamaño 5, para predecir el valor en 2 días

En la nomenclatura de Introduction_to_RNN_Time_Series.ipynb

```
past, future = (5, 2)
```

Para las primeras 10 muestras de train_data `[20.7, 17.9, 18.8, 14.6, 15.8, 15.8, 15.8, 17.4, 21.8, 20. ]` el resultado debería ser:

```
x[0] : [20.7, 17.9, 18.8, 14.6, 15.8] , y[0]: 15.8
x[1] : [17.9, 18.8, 14.6, 15.8, 15.8] , y[1]: 17.4
x[2] : [18.8, 14.6, 15.8, 15.8, 15.8] , y[2]: 21.8
x[3] : [14.6, 15.8, 15.8, 15.8, 17.4] , y[3]: 20.
```

```
import random

def convert2matrix(data_arr, past, future, shuffle=False):
    X, Y = [], []
    size = len(data_arr)
    for i in range(size - future - past + 1):
        d = i + past
        y_ind = i + past + future - 1
```

```
        X.append(data_arr[i:d])
        Y.append(data_arr[y_ind])
    if shuffle:
        c = list(zip(X, Y))
        random.shuffle(c)
        X, Y = zip(*c)
    return np.array(X), np.array(Y)
```

```
past, future = (5, 2)

X_train, y_train = convert2matrix(train_data, past, future, shuffle=True)
X_test, y_test = convert2matrix(test_data, past, future)
```

Cuestión 2: Cree un modelo recurrente de dos capas GRU para predecir con las ventanas de la cuestión anterior.

```
inputs = keras.layers.Input(shape=(past, 1))

...


GRU_layer_1 = keras.layers.GRU(32, return_sequences=True)(inputs)

GRU_layer_2 = keras.layers.GRU(32, return_sequences=False)(GRU_layer_1)

outputs = keras.layers.Dense(1)(GRU_layer_2)

...

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=keras.optimizers.Adam(), loss="mse")

model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 5, 1)]            0

 gru (GRU)                   (None, 5, 32)             3360

 gru_1 (GRU)                 (None, 32)                6336

 dense_23 (Dense)            (None, 1)                 33

=================================================================
Total params: 9,729
Trainable params: 9,729
Non-trainable params: 0
_________________________________________________________________
```

```
es_callback = keras.callbacks.EarlyStopping(
    monitor="val_loss", min_delta=0, patience=10)

history = model.fit(
    X_train, y_train,
    epochs=200,
    validation_split=0.2, shuffle=True, batch_size = 64, callbacks=[es_callback]
)
```

```
Epoch 1/200
38/38 [==============================] - 4s 21ms/step - loss: 90.3729 - val_loss: 46.5354
Epoch 2/200
38/38 [==============================] - 0s 7ms/step - loss: 29.9855 - val_loss: 26.6112
Epoch 3/200
38/38 [==============================] - 0s 6ms/step - loss: 20.6453 - val_loss: 22.2309
Epoch 4/200
38/38 [==============================] - 0s 6ms/step - loss: 17.8497 - val_loss: 20.3558
Epoch 5/200
38/38 [==============================] - 0s 7ms/step - loss: 16.6874 - val_loss: 19.4867
Epoch 6/200
38/38 [==============================] - 0s 6ms/step - loss: 15.9523 - val_loss: 18.3597
Epoch 7/200
38/38 [==============================] - 0s 6ms/step - loss: 14.3989 - val_loss: 15.8739
Epoch 8/200
38/38 [==============================] - 0s 6ms/step - loss: 12.3192 - val_loss: 13.8075
Epoch 9/200
38/38 [==============================] - 0s 6ms/step - loss: 11.0567 - val_loss: 12.7364
Epoch 10/200
```

```
38/38 [==============================] - 0s 6ms/step - loss: 10.2130 - val_loss: 12.0337
Epoch 11/200
38/38 [==============================] - 0s 7ms/step - loss: 9.7394 - val_loss: 11.5746
Epoch 12/200
38/38 [==============================] - 0s 5ms/step - loss: 9.4024 - val_loss: 11.1690
Epoch 13/200
38/38 [==============================] - 0s 6ms/step - loss: 9.1306 - val_loss: 11.0227
Epoch 14/200
38/38 [==============================] - 0s 6ms/step - loss: 8.9312 - val_loss: 11.0189
Epoch 15/200
38/38 [==============================] - 0s 6ms/step - loss: 8.8810 - val_loss: 10.6201
Epoch 16/200
38/38 [==============================] - 0s 7ms/step - loss: 8.7610 - val_loss: 10.7896
Epoch 17/200
38/38 [==============================] - 0s 7ms/step - loss: 8.6357 - val_loss: 10.4188
Epoch 18/200
38/38 [==============================] - 0s 6ms/step - loss: 8.6130 - val_loss: 10.4900
Epoch 19/200
38/38 [==============================] - 0s 6ms/step - loss: 8.5443 - val_loss: 10.4792
Epoch 20/200
38/38 [==============================] - 0s 6ms/step - loss: 8.4310 - val_loss: 10.7081
Epoch 21/200
38/38 [==============================] - 0s 5ms/step - loss: 8.3859 - val_loss: 10.3056
Epoch 22/200
38/38 [==============================] - 0s 5ms/step - loss: 8.4030 - val_loss: 10.3590
Epoch 23/200
38/38 [==============================] - 0s 6ms/step - loss: 8.3539 - val_loss: 10.2837
Epoch 24/200
38/38 [==============================] - 0s 6ms/step - loss: 8.3718 - val_loss: 10.2401
Epoch 25/200
38/38 [==============================] - 0s 7ms/step - loss: 8.2715 - val_loss: 10.4304
Epoch 26/200
38/38 [==============================] - 0s 6ms/step - loss: 8.3341 - val_loss: 10.2711
Epoch 27/200
38/38 [==============================] - 0s 7ms/step - loss: 8.2625 - val_loss: 10.2072
Epoch 28/200
38/38 [==============================] - 0s 6ms/step - loss: 8.2292 - val_loss: 10.1711
Epoch 29/200
```

```
results = model.evaluate(X_test, y_test, verbose=1)
print('Test Loss: {}'.format(results))
```

```
21/21 [==============================] - 0s 3ms/step - loss: 7.1065
Test Loss: 7.106459617614746
```

Cuestión 3: Añada más features a la series temporal, por ejemplo `portion_year`. Cree un modelo que mejore al anterior.

```
## Puede añadir más features
df['portion_year'] = df['Date'].dt.dayofyear / 365.0
df_multi = df[['Temp', 'portion_year']].copy()

## train - test split
train_data = df_multi.iloc[:3000].copy()
test_data = df_multi.loc[3000:, :].copy()
```

```
def convert2matrix_multi(df, past, future, target, shuffle=False):
    X, Y = [], []
    size = len(df)
    for i in range(size - future - past + 1):
        d = i + past
        y_ind = i + past + future - 1
        X.append(df.iloc[i:d, :].values)
        Y.append(df.iloc[y_ind][target])
    if shuffle:
        c = list(zip(X, Y))
        random.shuffle(c)
        X, Y = zip(*c)
    return np.array(X), np.array(Y)
```

```
## Create windows

X_train, y_train = convert2matrix_multi(train_data, past, future,target='Temp', shuffle=True)
X_test, y_test = convert2matrix_multi(test_data, past, future, target='Temp')
```

```
inputs = keras.layers.Input(shape=(past, 2))

...

```

```
GRU_layer_1 = keras.layers.GRU(32, return_sequences=True)(inputs)

GRU_layer_2 = keras.layers.GRU(32, return_sequences=False)(GRU_layer_1)

outputs = keras.layers.Dense(1)(GRU_layer_2)

...

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=keras.optimizers.Adam(), loss="mse")

model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 5, 2)]            0

 gru_2 (GRU)                 (None, 5, 32)             3456

 gru_3 (GRU)                 (None, 32)                6336

 dense_24 (Dense)            (None, 1)                 33

=================================================================
Total params: 9,825
Trainable params: 9,825
Non-trainable params: 0
_________________________________________________________________
```

```
es_callback = keras.callbacks.EarlyStopping(
    monitor="val_loss", min_delta=0, patience=10)

history = model.fit(
    X_train, y_train,
    epochs=200,
    validation_split=0.2, shuffle=True, batch_size = 64, callbacks=[es_callback]
)
```

```
Epoch 1/200
38/38 [==============================] - 3s 22ms/step - loss: 80.2641 - val_loss: 44.5195
Epoch 2/200
38/38 [==============================] - 0s 6ms/step - loss: 33.0398 - val_loss: 28.1451
Epoch 3/200
38/38 [==============================] - 0s 7ms/step - loss: 24.0968 - val_loss: 22.1943
Epoch 4/200
38/38 [==============================] - 0s 7ms/step - loss: 20.0992 - val_loss: 19.2634
Epoch 5/200
38/38 [==============================] - 0s 7ms/step - loss: 18.1654 - val_loss: 17.6836
Epoch 6/200
38/38 [==============================] - 0s 10ms/step - loss: 16.9349 - val_loss: 16.2380
Epoch 7/200
38/38 [==============================] - 0s 11ms/step - loss: 14.7866 - val_loss: 14.0108
Epoch 8/200
38/38 [==============================] - 0s 11ms/step - loss: 12.9759 - val_loss: 12.4250
Epoch 9/200
38/38 [==============================] - 0s 8ms/step - loss: 11.7557 - val_loss: 11.2979
Epoch 10/200
38/38 [==============================] - 0s 6ms/step - loss: 10.9749 - val_loss: 10.6121
Epoch 11/200
38/38 [==============================] - 0s 6ms/step - loss: 10.4308 - val_loss: 10.3607
Epoch 12/200
38/38 [==============================] - 0s 7ms/step - loss: 9.9694 - val_loss: 9.7115
Epoch 13/200
38/38 [==============================] - 0s 6ms/step - loss: 9.6755 - val_loss: 9.4460
Epoch 14/200
38/38 [==============================] - 0s 6ms/step - loss: 9.4734 - val_loss: 9.3949
Epoch 15/200
38/38 [==============================] - 0s 7ms/step - loss: 9.3287 - val_loss: 9.3038
Epoch 16/200
38/38 [==============================] - 0s 6ms/step - loss: 9.2191 - val_loss: 9.0740
Epoch 17/200
38/38 [==============================] - 0s 6ms/step - loss: 9.0834 - val_loss: 8.9714
Epoch 18/200
38/38 [==============================] - 0s 6ms/step - loss: 8.9925 - val_loss: 8.8928
Epoch 19/200
38/38 [==============================] - 0s 5ms/step - loss: 8.9303 - val_loss: 8.8132
Epoch 20/200
38/38 [==============================] - 0s 6ms/step - loss: 8.9419 - val_loss: 8.7892
Epoch 21/200
38/38 [==============================] - 0s 6ms/step - loss: 8.8565 - val_loss: 8.7773
Epoch 22/200
38/38 [==============================] - 0s 7ms/step - loss: 8.7928 - val_loss: 8.7043
Epoch 23/200
38/38 [==============================] - 0s 5ms/step - loss: 8.8201 - val_loss: 8.6821
Epoch 24/200
```

```
38/38 [==============================] - 0s 6ms/step - loss: 8.7976 - val_loss: 8.7087
Epoch 25/200
38/38 [==============================] - 0s 6ms/step - loss: 8.7054 - val_loss: 8.6222
Epoch 26/200
38/38 [==============================] - 0s 6ms/step - loss: 8.7295 - val_loss: 8.5671
Epoch 27/200
38/38 [==============================] - 0s 6ms/step - loss: 8.6874 - val_loss: 8.6506
Epoch 28/200
38/38 [==============================] - 0s 6ms/step - loss: 8.6785 - val_loss: 8.5051
Epoch 29/200
38/38 [==============================] - 0s 6ms/step - loss: 8.6218 - val_loss: 8.6372
```

```
results = model.evaluate(X_test, y_test, verbose=1)
print('Test Loss: {}'.format(results))
```

```
21/21 [==============================] - 0s 3ms/step - loss: 6.2271
Test Loss: 6.227080345153809
```

Cuestión 4: ¿En cuáles de estas aplicaciones se usaría un arquitectura 'many-to-one'?

a) Clasificación de sentimiento en textos

b) Verificación de voz para iniciar el ordenador.

c) Generación de música.

d) Un clasificador que clasifique piezas de música según su autor.

a) Clasificación de sentimiento en textos

b) Verificación de voz para iniciar el ordenador.

Cuestión 5: ¿Qué ventajas aporta el uso de word embeddings?

a) Permiten reducir la dimensión de entrada respecto al one-hot encoding.

b) Permiten descubrir la similaridad entre palabras de manera más intuitiva que con one-hot encoding.

c) Son una manera de realizar transfer learning en nlp.

d) Permiten visualizar las relaciones entre palabras con métodos de reducción de dimensioones como el PCA.

a) Permiten reducir la dimensión de entrada respecto al one-hot encoding.

b) Permiten descubrir la similaridad entre palabras de manera más intuitiva que con one-hot encoding.

c) Son una manera de realizar transfer learning en nlp.

d) Permiten visualizar las relaciones entre palabras con métodos de reducción de dimensioones como el PCA.