2023

MEMORIA CASO MÉDICO

INFORMÁTICA MÉDICA, UPV
MANUEL SÁNCHEZ PICAZO, JOSE MIGUEL MARTINEZ MARTIN,
ADRIÁN FERNÁNDEZ CEBRIÁ, ÁNGEL GIMÉNEZ GONZÁLEZ

1. INTRODUCCIÓN: JUSTIFICACIÓN BIBLIOGRÁFICA Y ASPECTOS GENERALES

La Resonancia Magnética Nuclear (RMN) es una técnica de imagenología médica que se utiliza para visualizar los tejidos y estructuras internas del cuerpo humano de forma no invasiva. Las imágenes de RMN pueden proporcionar información detallada sobre la anatomía del cuerpo y se utilizan comúnmente en la evaluación del cerebro, la médula espinal y otras partes del cuerpo.

Con el creciente uso de la imagenología médica, la interpretación de las imágenes de RMN se ha convertido en una tarea cada vez más desafiante. El proceso de análisis de las imágenes de RMN es largo y requiere de expertos altamente capacitados para su interpretación, lo que puede llevar a retrasos en el diagnóstico y tratamiento de los pacientes.

El aprendizaje profundo (Deep Learning) ha demostrado ser una herramienta poderosa en el campo de la imagenología médica, y se ha utilizado en una variedad de aplicaciones, incluyendo la segmentación de imágenes, la clasificación y el análisis de patrones. En este trabajo, se propone el desarrollo de un método basado en Deep Learning para identificar el tipo de imagen de RMN para su uso en la plataforma VolBrain.

El objetivo principal de este trabajo es crear un método automatizado para identificar el tipo de imagen de RMN, lo que puede ayudar a reducir el tiempo necesario para el análisis de las imágenes y mejorar la eficiencia del diagnóstico. Se espera que los resultados de este trabajo contribuyan al desarrollo de soluciones innovadoras en el campo de la imagenología médica y mejoren la atención al paciente.

2. MATERIALES

Los materiales que vamos a usar para el desarrollo de nuestro proyecto es un conjunto de imágenes RMN y varios códigos python en su versión 3.6 utilizando las librerías tensorflow y pillow.

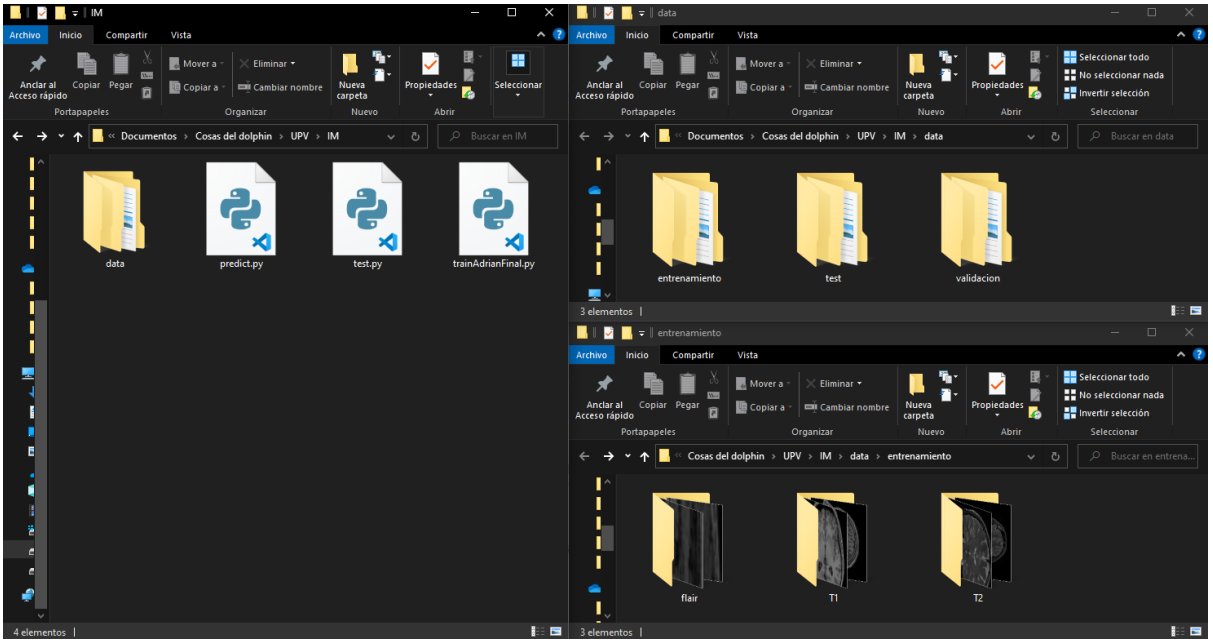
El código base lo hemos sacado del siguiente enlace AQUÍ , el código original estaba pensado para clasificar gatos, perros y gorilas, nos venía genial puesto que nosotros también necesitamos clasificar en 3 categorías

2.1 DATOS

Las más de 6400 imágenes de RMN proporcionadas están ya ordenadas, revisadas y separadas en su tipo de imagen, para que podamos utilizarlas para nuestro programa.
El código dividido en 3 secciones, la parte de train, encargada de adaptar y modificar lo que se debe priorizar en el modelo para poder tener la mayor posibilidad de éxito; la parte de predict, encargada de intentar predecir el tipo de una imagen sin mirarlo antes y la parte de test, encargada de comprobar las predicciones hechas por el modelo y indicar como de bueno es el mismo.

2.2 PREPROCESO ANTERIOR A EXPERIMENTOS

Antes de intentar ejecutar el código tendremos que descargar la versión correcta de python (3.8 o superior) e instalar el pip y las librerías Tensorflow, Keras y Pillow para que puedan funcionar los códigos, además de configurar la sesión de Keras y limpiar sesiones previas.
También tenemos que asegurarnos de tener las carpetas con las imágenes descomprimidas, organizadas en entrenamiento, test y validación y localizadas en la misma carpeta que los códigos a ejecutar.



3. MÉTODOS

Los 3 Métodos de python són Predict, Test y Train.
El más importante y con el que empezaremos será el train, encargado de entrenar a la red y guardar los datos, el predict servirá para ver que funciona correctamente, pasándole una imagen y que nos devuelva su tipo. Por último el test se encargará de cargar la red y evaluarla para todos los datos de test, viendo realmente su potencial. Los datos y resultados se mostrarán junto a las mejoras con el fin de comparar.

3.1 CÓDIGO BASE

TRAIN

```
import sys
import os
import tensorflow as tf
from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout, Flatten, Dense, Activation
from tensorflow.python.keras.layers import  Convolution2D, MaxPooling2D, Conv2D
from tensorflow.python.keras import backend as K
from keras.optimizers import Adam
from tensorflow.python.keras.callbacks import ModelCheckpoint, EarlyStopping
import numpy as np

K.clear_session()

#tienes que instalar esta lib
#pip install pillow

print("---------------------------------")

# paths relativos mejor para portar codigo
path=os.getcwd()
data_entrenamiento = path+os.sep+'data/entrenamiento'
data_validacion =  path+os.sep+'data/validacion'
data_test =  path+os.sep+'data/test'
```

->Este código se encarga del entrenamiento de la red neuronal, la entrenará y guardará los pesos.

Primero, importamos los paquetes necesarios, incluyendo TensorFlow y Keras. Además, configuramos la sesión de Keras y limpiamos cualquier sesión previa que pueda existir. Y después establecemos los directorios de entrenamiento, validación y prueba.

```
Parameters
"""
epocas=100
longitud, altura = 150, 150
batch_size = 32
pasos = 90
validation_steps = 30
filtrosConv1 = 32
filtrosConv2 = 64
tamano_filtro1 = (3, 3)
tamano_filtro2 = (2, 2)
tamano_pool = (2, 2)
clases = 3
```

->Definimos algunos parámetros importantes, como la longitud y la altura de la imagen, el batch size, el número de pasos, el número de filtros convolucionales y las dimensiones de los filtros y las capas de agrupamiento.

```
##Preparamos nuestras imagenes

entrenamiento_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    preprocessing_function=lambda x: x + np.random.normal(0.0, 0.1, x.shape))

val_datagen = ImageDataGenerator(rescale=1. / 255)
test_datagen = ImageDataGenerator(rescale=1. / 255)

entrenamiento_generador = entrenamiento_datagen.flow_from_directory(
    data_entrenamiento,
    target_size=(altura, longitud),
    batch_size=batch_size,
    class_mode='categorical')

validacion_generador = val_datagen.flow_from_directory(
    data_validacion,
    target_size=(altura, longitud),
    batch_size=batch_size,
    class_mode='categorical')


test_generador = test_datagen.flow_from_directory(
    data_test,
    target_size=(altura, longitud),
    batch_size=batch_size,
    class_mode='categorical')
```

->Esta parte se encarga de preparar las imágenes para ser utilizadas en el modelo de aprendizaje. Primero, se define un objeto ImageDataGenerator que realiza una serie de transformaciones en las imágenes de entrenamiento. Esto se hace para aumentar la cantidad de datos disponibles y evitar el sobreajuste del modelo.

Luego, se definen tres objetos flow_from_directory que generan lotes de imágenes a medida que se necesitan. Cada objeto se utiliza para cargar imágenes de entrenamiento, validación y pruebas respectivamente desde sus respectivas carpetas.

```
cnn = Sequential()
cnn.add(Conv2D(filtrosConv1, tamano_filtro1, padding ="same", input_shape=(longitud, altura, 3), activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Flatten())
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(256, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(clases, activation='softmax'))
cnn.summary()
```

Esta parte del código define la arquitectura de la red neuronal convolucional (CNN). Primero se crea un objeto Sequential para definir la secuencia de capas de la red. Luego, se agregan varias capas de convolución con activación relu y pooling, para extraer características de las imágenes de entrada. Finalmente, se agregan varias capas densas y de dropout con un valor de 0.5, lo que significa que se desactivaran la mitad de las neuronas para obligar a la red a aprender de otras formas.

La función summary() se utiliza para mostrar un resumen de la arquitectura de la CNN, incluyendo el número de parámetros entrenables en cada capa.

```python
savemodel =ModelCheckpoint(filepath, monitor='val_loss', save_best_only=True, save_weights_only=True, mode='min', period=1)
early=EarlyStopping(monitor='val_loss', patience=10)

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


cnn.fit(#antes fit_generator
    entrenamiento_generador,
    steps_per_epoch=pasos,
    epochs=epocas,
    validation_data=validacion_generador,
    validation_steps=validation_steps,
    callbacks=[savemodel,early])
```

Después de definir la arquitectura, guardamos el modelo con el callback ModelCheckPoint y añadimos un callback de EarlyStoppig para dejar de entrenar la red si no mejora el val_loss. Se compila la CNN con la función compile(), utilizando la función de pérdida categórica de entropía cruzada, el optimizador Adam y la métrica de precisión (accuracy) para evaluar el rendimiento del modelo.

Luego, se ajusta el modelo a los datos de entrenamiento utilizando la función fit(), utilizando un generador de imágenes para cargar las imágenes de entrenamiento. Además, se utilizan callbacks para detener el entrenamiento antes de que se produzca el sobreajuste y guardar el modelo con los mejores pesos en un archivo .h5.

```python
cnn.load_weights(filepath)

score=cnn.evaluate(test_generador)

print('Test loss:', score[0])
print('Test accuracy:', score[1])


try:
    # Ejecutar normalmente hasta que se interrumpa manualmente
    while True:
        pass
except KeyboardInterrupt:
    # Capturar la excepción y guardar el modelo antes de salir
    cnn.save(filepath)
    print("\nModelo guardado en", os.path.abspath(filepath))
```

Finalmente, se evalúa el rendimiento del modelo en los datos de prueba utilizando la función evaluate() y se imprime la pérdida y la precisión (accuracy) en los datos de prueba.

Se ha añadido un try except que nos permite guardar el modelo correctamente en caso de que interrumpamos el entrenamiento de forma manual.

PREDICT

Tanto el predict como el test tiene la mayor parte del código compartido, imports, paths, la red neuronal… Por lo que vamos a ver solo las parte que nos interesan.

```python
test_datagen = ImageDataGenerator(rescale=1. / 255)
test_generador = test_datagen.flow_from_directory(
    data_test,
    target_size=(altura, longitud),
    batch_size=batch_size,
    class_mode='categorical')
```

En esta sección, preparamos nuestras imágenes de prueba con un generador de datos de imagen. Escalamos las imágenes para que estén en un rango de 0 a 1 y definimos el tamaño de la imagen y el tamaño del lote.

```python
# cargamos el modelo
filepath="modelo.h5"
cnn.load_weights(filepath)

#aplicamos a una imagen
img = Image.open(path+"/data/test/T1/T1_497_3.png")
img = img.resize((longitud,altura),Image.ANTIALIAS)
img = np.asarray(img)
img=img/255
print(img.shape)
x=np.zeros((longitud,altura,3))
x[:,:,0]=img
x[:,:,1]=img
x[:,:,2]=img
x = np.reshape(x,(1,longitud,altura,3))

array = cnn.predict(x)

result = array[0]
answer = np.argmax(result)
if answer == 0:
    print("pred: T1")
elif answer == 1:
    print("pred: T2")
elif answer == 2:
    print("pred: flair")
```

Esta es la parte más característica del predict, Primero se abre la imagen utilizando la biblioteca PIL y se cambia su tamaño para que coincida con el tamaño de entrada esperado por el modelo (línea 6). Luego, se normaliza la imagen dividiendo cada valor de píxel por 255 (línea 7).

A continuación, se crea una matriz que tiene el tamaño esperado de entrada del modelo y se copia la imagen normalizada en cada una de las tres capas (líneas 8 a 10). Después, se reformatea la matriz en un tensor de cuatro dimensiones.

Finalmente, se llama al método predict en el modelo con la imagen reformateada como entrada (línea 13), lo que devuelve una matriz de predicción que contiene las probabilidades de cada clase. La predicción final se determina tomando la clase con la mayor probabilidad (línea 15-21), y se imprime en la pantalla.

TEST

Del test no vamos a mencionar nada, el código es el mismo que el de train, simplemente se obvia la parte de generar los lotes y transformaciones así como la generación del modelo .h5.

3.2 MEJORAS AL CÓDIGO

1) La última instrucción de la parte que se encarga de aplicar las transformaciones y aumentos de datos de las imágenes hemos añadido la función preprocessing la cual se encarga de añadir ruido (en este caso ruido gaussiano) a las imágenes, esto ayudara a crear un código más robusto y también ayudará a controlar el overfitting.

Necesitaremos importar numpy `import numpy as np`

```
entrenamiento_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    preprocessing_function=lambda x: x + np.random.normal(0.0, 0.1, x.shape))
```

2) Hemos añadido otras dos capas de convolución y de agrupación máxima (max pooling) al código original con el objetivo de mejorar su capacidad de aprendizaje y su capacidad para realizar tareas de clasificación de imágenes más complejas aprendiendo patrones más complicados.

```
cnn = Sequential()
cnn.add(Conv2D(filtrosConv1, tamano_filtro1, padding ="same", input_shape=(longitud, altura, 3), activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, padding ="same",activation='relu'))
cnn.add(MaxPooling2D(pool_size=tamano_pool))
```

3) También hemos añadido otras dos capas densas de 256 y 128 neuronas con activación relu y y otra capa de Dropout con una tasa de 0.5, es decir desactivaremos la mitad de las neuronas, con esto mejoraremos aún más la capacidad de prevenir overfitting y ayudar a la mejoría de la generalización, obligando a las neuronas a aprender de otra forma.

```
cnn.add(Flatten())
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(256, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(clases, activation='softmax'))
cnn.summary()
```

4) En la parte del guardado del archivo hemos añadido un Early Stopping con el monitor a val_loss y un patience de 10, esto también nos ayudará a evitar que entrene de más sin necesidad, su forma de actuar es esperar a 10 épocas para ver si el valor de val_loss mejora, en caso de que mejore el código seguirá ejecutándose, si no lo hace entonces pararía, en resumen es como darle un "ultimátum" o mejoras en las próximas 10 épocas o paras y guardas el modelo. Pusimos patience 15 pero eso generaba problemas con el overfitting, los resultados con patience 10 son más que aceptables.

Necesitamos importar la librería

```
from keras.callbacks import EarlyStopping
```

```
savemodel = [EarlyStopping(monitor='val_loss', patience=10),
             ModelCheckpoint(filepath, monitor='val_loss', save_best_only=True, save_weights_only=True, mode='min', period=1)]
```

5) Añadimos un try para que cuando cancelemos el código con Ctrl+C guarde el archivo correctamente y no tengamos ningún problema.

```
try:
    # Ejecutar normalmente hasta que se interrumpa manualmente
    while True:
        pass
except KeyboardInterrupt:
    # Capturar la excepción y guardar el modelo antes de salir
    cnn.save(filepath)
    print("\nModelo guardado en", os.path.abspath(filepath))
```

3.3 EXPERIMENTACIÓN Y EVALUACIÓN

Aunque parezcan pocas mejoras y que el código es muy parecido al código base, los resultado son muy buenos, en muchos casos pasarse con la complejidad sale mal
Actualmente tenemos estos datos cuando la red se deja de entrenar:

```
Epoch 31/100
90/90 [==============================] - 17s 192ms/step - loss: 0.0112
  1081/Unknown - 48s 44ms/step - loss: 0.0014 - accuracy: 0.9990
```

Que pasa si dejamos dos solo:

```
cnn = Sequential()
cnn.add(Conv2D(filtrosConv1, tamano_filtro
cnn.add(MaxPooling2D(pool_size=tamano_pool

cnn.add(Conv2D(filtrosConv2, tamano_filtro
cnn.add(MaxPooling2D(pool_size=tamano_pool

cnn.add(Conv2D(filtrosConv2, tamano_filtro
cnn.add(MaxPooling2D(pool_size=tamano_pool


cnn.add(Flatten())
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(256, activation='relu'))
cnn.add(Dropout(0.5))
```

```
0/90 [==============================] - 17s 187ms/step - loss: 0.0096
  525/Unknown - 23s 43ms/step - loss: 9.4190e-05 - accuracy: 1.0000
```

Accuracy de 1 y un los de casi cero lo que es genial pero el accuracy llega a 1 lo que podía indicar overfitting.

¿Qué pasaría si añadimos más?

Vamos a añadir 1 de max pooling y convolución y dos de Dropout y dense:

```
cnn.add(Conv2D(filtrosConv1, tamano_filtro1, pa
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, pa
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, pa
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, pa
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Conv2D(filtrosConv2, tamano_filtro2, pa
cnn.add(MaxPooling2D(pool_size=tamano_pool))

cnn.add(Flatten())
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(256, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(64, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(clases, activation='softmax'))
```

No mejora, empeora.

```
Epoch 57/100
90/90 [==============================] - 17s 189ms/step - loss: 0.0322
  1860/Unknown - 85s 46ms/step - loss: 0.0020 - accuracy: 0.9989
```

4. RESULTADOS Y CONCLUSIONES

Con los cambios realizados al código base conseguimos crear una red neuronal capaz de identificar la clase de la imágen RMN con una eficacia prácticamente perfecta en cuestión de meros segundos, lo que ayuda en gran medida a diagnosticar posibles anomalías cerebrales rápidamente para poder ofrecer tratamiento lo antes posible en el caso de ser necesario y por ende tener más posibilidades de tratar dichas anomalías exitosamente.
Está claro que el uso de redes neuronales y otros algoritmos similares tienen un potencial altísimo para mejorar la calidad y velocidad de diagnósticos en el sector de salud en general y es simplemente cuestión de tiempo la implementación de este tipo de sistemas en casi todos los aspectos de sanidad, aunque a pesar de ya haber empezado a implementarse, aún le queda mucho por desarrollar e implementar.

REFERENCIAS

Hemos encontrado un par de estudios que nos demuestra la importancia que tiene la clasificación de imágenes de RMN automática:

1. En el estudio sobre segmentación automática de imágenes de RMN cerebrales, los autores utilizaron un enfoque basado en una red neuronal convolucional para segmentar la sustancia gris y blanca en las imágenes de RMN cerebrales. El estudio fue publicado en la revista "Neuroinformatics" en 2020 y se titula "Automated brain tissue segmentation based on deep learning: A systematic review".

2. En el estudio sobre la clasificación de imágenes de RMN de la columna vertebral, los autores utilizaron una red neuronal convolucional para clasificar las imágenes de RMN en tres categorías: normal, degenerativa y hernia de disco. El estudio fue publicado en la revista "Computerized Medical Imaging and Graphics" en 2021 y se titula "Automated classification of lumbar spine MRI using deep learning".