MEMORIA PRÁCTICA 2

REDES NEURONALES: RECONOCIMIENTO DE SEÑALES DE TRÁFICO

FUNDAMENTOS DE LOS SISTEMAS INTELIGENTES

SEBASTIÁN FERNÁNDEZ GARCÍA Y NAHIMA ORTEGA RODRÍGUEZ

INTRODUCCIÓN

En esta memoria, vamos a resumir los principales aspectos de la práctica que hemos realizado. Se trata de una red neuronal que sea capaz de reconocer nueve tipos distintos de señales de tráfico de tipo precaución, o advertencia de peligro.

Nuestra principal motivación se debe a que una gran mayoría de automóviles actuales son capaces de reconocer señales de límite de velocidad y mostrárselas posteriormente al conductor. Esto ha pasado a ser una característica de incluso aquellos vehículos que no son necesariamente de una alta gama. Sin embargo, todavía no son capaces de detectar señales de tráfico que adviertan algún tipo de peligro al conductor, a pesar de la gran importancia que tienen éstas.

Hemos seleccionado un total de 9 categorías distintas:

1. Señal P-50. Otros peligros.
2. Señal P-14b. Curvas peligrosas hacia la izquierda.
3. Señal P-20. Peatones.
4. Señal P-19. Pavimento deslizante.
5. Señal P-18. Obras.
6. Señal P-3. Semáforos.
7. Señal P-15a. Resalto.
8. Señal P-1. Intersección con prioridad.
9. Señal P-24. Paso de animales en libertad.

Las imágenes han sido extraídas en gran parte del dataset GTSRB (German Traffic Sign Recognition Benchmark), aunque también hemos obtenido imágenes de otros datasets o han sido de completa elaboración personal, como es el caso de la séptima y octava señal.

CONFIGURACIÓN DE LOS HIPERPARÁMETROS

A continuación, vamos a realizar una tabla comparativa con los hiperparámetros que hemos probado y su respectivos resultados, ya que nos quedaremos con el mejor de ellos.

	Capas	Batch Size	Epoch	Optimizer	Validation Accuracy
Versión 1	Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Conv2D, 64, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 128, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9874 (datos de train y test distintos al resto)
Versión 2	Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Conv2D, 64, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 128, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9667 (nuevos datos)
Versión 3	Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Conv2D, 64, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 128, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	SGD	0.8667
Versión 3.2	"	"	"	Adagrad	0.8861
Versión 3.3	"	"	"	Nadam	0.9778
Versión 3.4	"	"	"	RMSprop	0.9694
Versión 4	Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.25 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Nadam	0.9694
Versión 4.3	Conv2D, 32, 5x5, relu MaxPooling2D 2x2	32	200	Adam	0.9806

	Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax				
Versión 5	Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9222
Versión 5.2	Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Conv2D, 64, 3x3, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9389
Versión 6	Conv2D, 16, 3x3, relu MaxPooling2D 2x2 Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 64, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9278
Versión 7	Conv2D, 16, 3x3, relu MaxPooling2D 2x2 Conv2D, 32, 3x3, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 64, 3x3, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 64, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9806
Versión 7.2	Conv2D, 16, 5x5, relu MaxPooling2D 2x2 Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 64, relu Dropout 0.5 Dense, 9, softmax	32	200	Adam	0.9750
Versión 8	Conv2D, 16, 5x5, relu MaxPooling2D 2x2	32	200	Adam	0.9833

	Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Dropout 0.25 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax				
Versión 8.2	Conv2D, 16, 5x5, relu MaxPooling2D 2x2 Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	32	200	Nadam	0.9750
Versión 10	Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 128, relu Dropout 0.5 Dense, 9, softmax	64	100	Adam	0.9778
Versión 10.2	"	100	"	"	0.9750
Versión 10.3	"	32	"	"	0.9806
Versión 11	Conv2D, 32, 5x5, relu MaxPooling2D 2x2 Conv2D, 64, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Conv2D, 128, 5x5, relu MaxPooling2D 2x2 Dropout 0.5 Dense, 9, softmax	32	100	Adam	0.9667

Nuestros mejores valores los obtenemos para la versión 7, 8 y 10.3. Sin embargo, en la versión 8 hemos empleado una capa convolutiva más, es decir, hemos hecho la red un poco más compleja, pero no hemos obtenido unos resultados mucho mejores. Por lo tanto, nos quedaremos con la versión 7.

Algunas de las conclusiones que hemos obtenido después de probar con los distintos parámetros es que, para nuestro caso, nos ha funcionado mucho mejor los optimizadores Adam y Nadam. Además, para evitar el sobreajuste hemos incluido una tasa de Dropout de 0.5. Esto ha hecho que también aumente ligeramente el porcentaje de acierto.

Hemos obtenido mejores resultado para tamaños de lote de 32 imágenes y peores resultados para tamaños de lote más altos. Además, disminuyendo el número de capas obtenemos un acierto en torno al 90%, mientras que añadiéndole una tercera capa, este valor se sitúa por encima del 95%. Sin embargo, al añadirle una cuarta capa este valor no aumenta considerablemente. Por lo tanto, creemos que el número de capas que más se adecúa a nuestro objetivo es tres. En cuanto al tamaño del kernel de cada capa, el tamaño de 5x5 nos había dado, en un principio, una ligera mejora. Propagándolo a todas las capas, la diferencia ha sido un poco más significativa. A pesar de ello, hay ciertas configuraciones que hemos probado con el tamaño 3x3 y ha resultado ser más efectivo que 5x5, como es el caso de la séptima versión.

Como se puede apreciar, hemos obtenido valores muy altos de acierto en general. Creemos que esto se ha debido a que las imágenes se presentan siempre en condiciones parecidas. De ahí que, a partir de la versión 2, se comenzara a usar un conjunto de datos de validación elaborado por nosotros manualmente, en el que incluíamos imágenes un poco más distorsionadas, con obstáculos, desde diferentes perspectivas, etc, para asegurarnos que este conjunto no fuera muy sencillo.

CÓDIGO

A continuación vamos a explicar los aspectos más relevantes del código, que ha sido desarrollado en Google Colab.

En una primera parte, debemos conectarnos a Google Drive para poder cargar nuestros datasets de entrenamiento y validación, que habían sido subidos previamente a esta plataforma. En un principio, todas las categorías cuentan, en general, con 200 imágenes. De estas, 40 se han dedicado al set de validación y las 160 restantes, para el entrenamiento.

Montamos nuestra unidad de Google Drive para poder acceder a nuestros ficheros.

```
[1]  from google.colab import drive
     drive.mount('/content/drive')
```

También debemos importar Tensorflow, ya que vamos a trabajar con esta librería para programar nuestra red neuronal. En este caso, comprobamos en este chunk que contamos con una GPU en nuestro cuaderno de Colab, que será muy útil para la aceleración de los cálculos a la hora de entrenar nuestro modelo.

```
[2]  import tensorflow as tf
     tf.test.gpu_device_name()

     '/device:GPU:0'
```

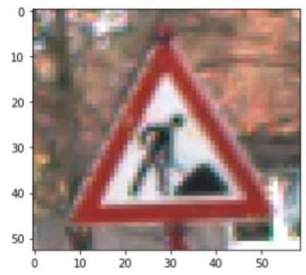
Visualizamos una de las imágenes de nuestro Training Set. Para ello usamos una serie de librerías que hemos importado (matplotlib, numpy y PIL).

```
[ ]  !ls "/content/drive/My Drive/Colab Notebooks/datasets/"

     from matplotlib.pyplot import imshow
     import numpy as np
     from PIL import Image

     %matplotlib inline
     pil_im = Image.open('/content/drive/My Drive/Colab Notebooks/datasets/Traffic-Signs/5/00025_00040_00018.png', 'r')
     imshow(np.asarray(pil_im))
```

```
Traffic-Signs  Traffic-Signs-2  Traffic-Signs-Validation
<matplotlib.image.AxesImage at 0x7f7d4003eb90>
```



También hemos realizado data augmentation sobre el conjunto de datos de entrenamiento.

El principal "problema" que nos hemos encontrado al usar este dataset de señales de tráfico es que, en la mayoría de casos, nos encontramos las señales desde la parte derecha, desde ángulos y distancias similares. Además, estas señales (en comparación con otros datasets

relacionados con objetos que pueden diferir de una imagen a otra, como animales, personas, etc.), tienen la particularidad de que todas presentan las mismas condiciones (mismo tamaño, color, representación), ya que se encuentran reguladas y estandarizadas por las autoridades competentes de cada país. Por todo ello, para contar con la máxima variedad de imágenes posibles, creamos dos imágenes distintas a partir de cada imagen en el set de entrenamiento usando el aumento de datos.

En primer lugar, creamos un ImageDataGenerator, que nos va a permitir crear nuevas imágenes conforme a los parámetros que le pasamos. Indicamos que no queremos que le dé la vuelta de manera horizontal ni vertical a nuestras imágenes, ya que esto no tendría sentido en ningún caso real. Esto se debe a que las señales no suelen ser simétricas ni tampoco las solemos encontrar hacia abajo. Nuestro interés principal es rotarlas y hacerles zoom. Por ello, establecemos un rango de rotación de 25º y un rango de zoom del 20%. Además, damos un rango de corte de 0.20.

Una vez establecidos los parámetros, leemos todas las imágenes de nuestro Training Set, y generamos las nuevas imágenes con la función flow(). Estas imágenes las guardamos en nuestro Google Drive con objetivo de supervisión, para comprobar posteriormente si las imágenes que se están generando cumplen con las características que buscamos.

```
from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
import glob

datagen = ImageDataGenerator(
  horizontal_flip=False,
  vertical_flip=False,
  rotation_range=25,
  zoom_range=0.20,
  shear_range=0.20,
  fill_mode="nearest")

for i in range(1, 10, 1):
  for name in glob.glob("/content/drive/My Drive/Colab Notebooks/datasets/Traffic-Signs/" + str(i) + "/*.png"):
    img = load_img(name)
    x = img_to_array(img)
    x = x.reshape((1,) + x.shape)
    j = 0
    for batch in datagen.flow(x, batch_size=1,
                              save_to_dir="/content/drive/My Drive/Colab Notebooks/datasets/Traffic-Signs/" + str(i) + "/da/",
                              save_prefix="image", save_format='png'):
      j += 1
      if j >= 2:
        break
print("Finalización del aumento de datos")
```

Efectivamente, se perciben cambios notorios que nos ayudan a crear un dataset un poco más variado, como es el siguiente.

Ahora, indicaremos cuáles serán nuestros conjuntos de entrenamiento y validación. Antes, establecemos el tamaño que tendrán nuestras imágenes. Debido a que no queremos un gran resolución, escogemos un tamaño de 150x150. Además, indicamos un tamaño de batch de

32. Esto quiere decir que se escogerán 32 muestras para calcular el gradiente de error antes de que los pesos del modelo sean actualizados. Es decir, las imágenes de entrenamiento se pasarán a la red en lotes de 32 imágenes.

Nuestro conjunto de entrenamiento lo traemos de nuestro directorio de Google Drive, e indicamos los parámetros de tamaño de la imagen y tamaño de los lotes. En el label_mode indicamos categorical porque las etiquetas se van a representar como un vector categórico, ya que vamos a tener más de dos clases. Seguidamente, realizamos el mismo procedimiento con el conjunto de validación.

Por último, indicamos el número máximo de elementos que se van a almacenar en el buffer durante el procedimiento de prefetch.

```
[18] # DATA SOURCE ------------------------------------------------

     image_size = (150, 150)
     batch_size = 32

     train_ds = tf.keras.preprocessing.image_dataset_from_directory(
         "/content/drive/My Drive/Colab Notebooks/datasets/Traffic-Signs",
         image_size=image_size,
         batch_size=batch_size,
         label_mode='categorical'
     )
     val_ds = tf.keras.preprocessing.image_dataset_from_directory(
         "/content/drive/My Drive/Colab Notebooks/datasets/Traffic-Signs-Validation",
         image_size=image_size,
         batch_size=batch_size,
         label_mode='categorical'
     )

     train_ds = train_ds.prefetch(buffer_size=32)
     val_ds = val_ds.prefetch(buffer_size=32)
```

Ahora entramos a la parte de crear el modelo de nuestra red neuronal.

En primer lugar, con Sequential(), indicamos que vamos a comenzar a construir un modelo secuencial, es decir, que las capas se van a agregar en secuencia. Luego, con Rescaling(), reescalamos la entrada en el rango [0, 255] al rango [-1, 1] y también incluimos el tamaño que queremos que tengan las imágenes en nuestra red.

Una vez hecho esto, comenzamos a añadir capaz a la red.

La primera que añadimos tiene 32 filtros, un tamaño de kernel de 5x5 y cuenta con ReLU-activation y max-pooling. Pero, ¿qué significa todo esto?

Por una parte, los filtros se emplean para representar características de la imagen.

Por otra parte, el kernel es un filtro que vamos a pasar por encima de la imagen de entrada para hallar el valor de convolución. Este kernel se utilizará para detectar ciertas características de las imágenes.

Por último, aplicamos la función de activación ReLU y max-pooling para eliminar daros sin reducir en gran medida la información.

El Dropout sirve para para prevenir el sobreentrenamiento, y lo hace poniendo unidades de entrada a 0 en la frecuencia que indiquemos.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Dense, Rescaling, Flatten
from tensorflow.keras.callbacks import EarlyStopping

model = keras.Sequential()
model.add(Rescaling(scale=(1./127.5),
                    offset=-1,
                    input_shape=(150, 150, 3)))
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(9, activation='softmax'))

model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(1e-3),
              metrics=['accuracy'])
```

También vemos cómo usamos como función de pérdida categorical_crossentropy. Esta es una medida de la distancia entre distribuciones de probabilidad. Se emplea en tareas de clasificación en las que contamos con varias clases, y en las que un ejemplo solo puede pertenecer a una de las posibles categorías, teniendo que decidir nuestro modelo cuál de ellas. Es decir, es idóneo para nuestro caso, ya que clasificamos una imagen en las nueve posibles categorías que tenemos. Esta función puede considerar que un ejemplo pertenece a una categoría con probabilidad 1 y a otras categorías con probabilidad 0, de ahí que sea empleada para la clasificación.

Dada una imagen, la predicción es un vector de probabilidad que representa las probabilidades que se predicen para cada una de las clases. Estas predicciones se obtienen normalmente cuando se tiene como función de activación para la última capa la función softmax. Además, tenemos otro vector one-hot, que indica con un 1 en una posición la categoría que representa. Por lo tanto, para una determinada categoría, queremos que la probabilidad sea 1, mientras que para otras clases queremos que sea 0.

Al ser una función de pérdida, cuanto más bajo sea el valor de ésta, mejor. En el entrenamiento, el modelo intenta conseguir la menor pérdida posible.

Para la función cross-entropy, calculamos la pérdida para cada una de las clases de manera independiente y luego las sumamos. La pérdida se computa multiplicando la probabilidad de la clase en el vector one-hot multiplicado por el logaritmo de la clase en el vector predicción. A esta operación se le invierte el signo, para que tienda a infinito o a 0. Para la función Categorical Cross Entropy, olvidamos el resto de clases, y solo calculamos la pérdida para la clase indicada con un 1 en el vector one hot (hot class). Es decir, el valor de la función será el logaritmo de la probabilidad.

Como vemos, esta función no tiene en cuenta si tenemos una señal de precaución por resalto y lo interpreta como una señal de curvas peligrosas con un 70% de probabilidad, sino que tiene en cuenta que quedaría un 30% restante para identificar a la clase correcta, siendo en este caso el valor de pérdida muy alto. Es decir, no se tiene en cuenta a dónde va el resto de la probabilidad, solo se fija en cómo de bien identifica la clase resultado.

Por lo tanto, la pérdida es 0 si la predicción es 1. Por otra parte, la pérdida tiene a infinito si la predicción es 0, que es lo opuesto al resultado de nuestro vector one-hot. Por ello, cuanto más lejos nos encontremos del valor del vector one-hot, más rápido crecerá la función de error.

Una vez explicada la función Categorical Cross Entropy, vemos que usamos un optimizador: "Adam". A lo largo de las versiones, hemos probado con otros optimizadores como Adagrad, que adapta el learning rate a los parámetros llevando a cabo pequeñas actualizaciones para las características más comunes y actualizaciones mucho más grandes para características más raras. Por otro lado, Adadelta intenta reducir la velocidad de aprendizaje agresiva de Adagrad restringiendo la ventana del gradiente acumulado. RMSProp también intenta resolver las tasas de aprendizaje decrecientes de Adagrad usando un promedio del gradiente al cuadrado. Por último, Adam funciona como una combinación. de Adagrad y RMSprop.

A continuación, comenzamos a entrenar nuestra red neuronal. Para ello, definimos el número de épocas y usamos EarlyStopping para prevenir el sobreentrenamiento. El valor de patience indica el número de épocas que se espera en el caso de empeoramiento y restore_best_weights nos permite retornar al valor más alto obtenido.

```
# TRAINING -------------------------------------------------

epochs = 100

es = EarlyStopping(monitor='val_accuracy', mode='max', verbose=1, patience=10, restore_best_weights=True)

h = model.fit(
        train_ds,
        epochs=epochs,
        validation_data=val_ds,
        callbacks = [es]
)
```

Por último, examinamos los valores de acierto que nos da nuestra red neuronal y creamos la matriz de confusión. Adicionalmente, hemos probado con imágenes que la red neuronal no ha visto nunca para comprobar que reconocía correctamente a qué categoría pertenece. Como vemos, las categorías se encuentran numeradas de 1 y al 9 por simplicidad.

Se muestran los resultados que hemos obtenido de manera gráfica mediante la ejecución de este código.

```
[ ]  import matplotlib.pyplot as plt

     plt.plot(h.history['accuracy'])
     plt.plot(h.history['val_accuracy'])
     plt.plot(h.history['loss'])
     plt.title('Model accuracy')
     plt.ylabel('accuracy')
     plt.xlabel('epoch')
     plt.legend(['training', 'validation','loss'], loc='upper right')
     plt.show()
```

Para obtener la matriz de confusión:

```
[ ] import numpy as np
    from sklearn.metrics import classification_report, confusion_matrix
    import seaborn as sns

    results = np.concatenate([(y, model.predict(x=x)) for x, y in val_ds], axis=1)

    predictions = np.argmax(results[0], axis=1)
    labels = np.argmax(results[1], axis=1)

    cf_matrix = confusion_matrix(labels, predictions)

    sns.heatmap(cf_matrix, annot=True, fmt="d", cmap="Blues")

    print(classification_report(labels, predictions, digits = 4))
```

Por último, probamos con imágenes propias.

```
[ ] img = keras.preprocessing.image.load_img(
        '/content/drive/My Drive/Colab Notebooks/imagenes/imagen5.png', target_size=image_size
    )
    img_array = keras.preprocessing.image.img_to_array(img)
    img_array = tf.expand_dims(img_array, 0)  # Create batch axis

    predictions = model.predict(img_array)
    print(np.argmax(predictions[0]))
```

Enlace al repositorio de GitHub: https://github.com/nahimaort/Traffic-Signs-Recognition