9-1-2024

Red Neuronal

Cristóbal Jesús Sarmiento Rodríguez / 42263227Z
Arhamis Gutiérrez Caballero / 54164656R

Índice

- **Introducción**

Para este trabajo de redes neuronales, se nos ha pedido desarrollar desde cero una arquitectura neuronal para una clasificación multiclase. En este documento se resume brevemente nuestro proyecto, comentando qué hace cada parte del código y el cómo hemos llegado hasta ese punto.

Además, habrá dos partes, una con un proyecto hecho desde cero, y la otra, con un proyecto hecho con un modelo preentrenado.

- **Generadores**

Primero, elegimos un dataset multiclase, en nuestro caso, escogimos una clasificación de animales en diez clases distintas. Luego, sería recortado hasta cinco clases debido a que la red no era capaz de seguir mejorando.

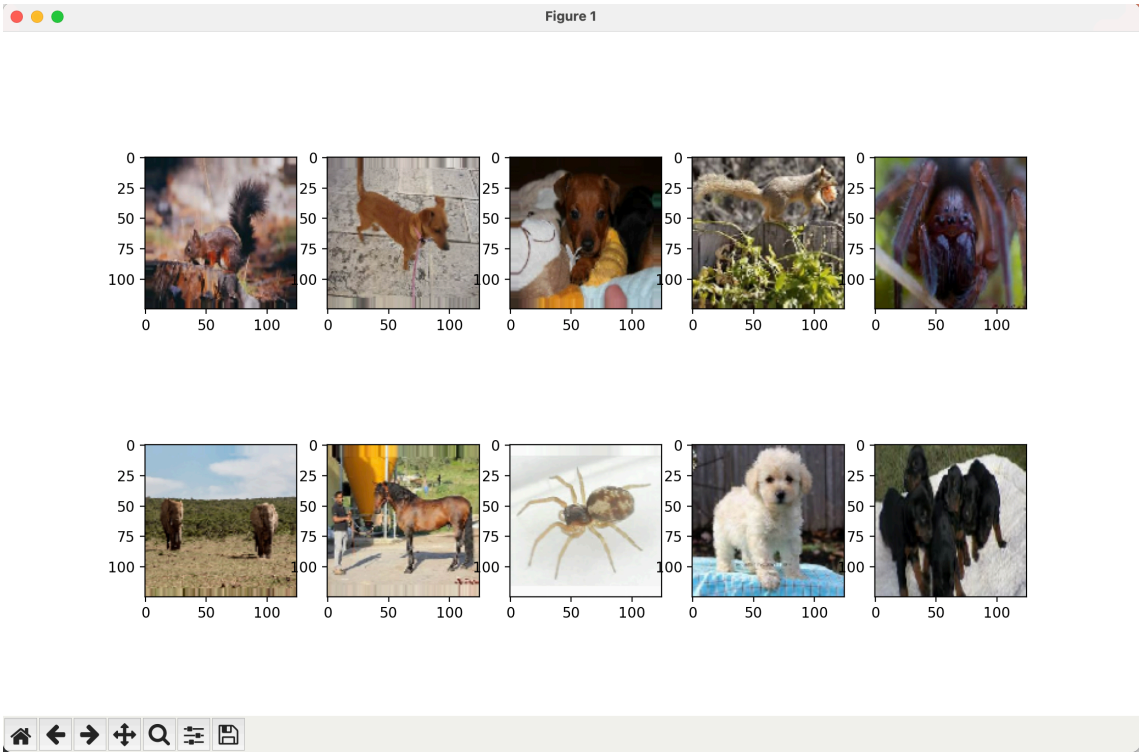
Además, eliminamos unas cuantas imágenes a mano de las clases que quedaban ya que algunas eran muy complicadas para la clasificación.

Una vez elegido el dataset, teníamos que crear las funciones que generan las imágenes que le pasaremos a la red.

```
train_data_dir = '/Users/cristobal/Desktop/archive4/raw-img/training'                    ⚠ 6  ⚠ 32  ✓ 34  ^

train_datagen = ImageDataGenerator(
    rescale = rescale_factor,  # Normalizar los valores de los píxeles
    shear_range = 0.2,  # Rango para las transformaciones aleatorias
    zoom_range = 0.2,  # Rango para el zoom aleatorio
    horizontal_flip = True,  # Activar el giro horizontal aleatorio
    validation_split = 0.2)  # Establecer el porcentaje de imágenes para el conjunto de validación)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,  # Directorio con datos
    target_size = (125, 125),  # Cambiar el tamaño de las imágenes a 50x50
    batch_size = batch_size,
    shuffle = True,
    class_mode = 'categorical',  # 'binary' para clasificación binaria, 'categorical' para multiclase
    subset = 'training')  # Seleccionar solo el conjunto de entrenamiento

validation_generator = train_datagen.flow_from_directory(
    train_data_dir,  # Directorio con datos
    target_size = (125, 125),  # Cambiar el tamaño de las imágenes a 50x50
    batch_size = batch_size,
    shuffle = True,
    class_mode = 'categorical',  # 'binary' para clasificación binaria, 'categorical' para multiclase
    subset = 'validation')  # Seleccionar solo el conjunto de entrenamiento

for imagen, etiqueta in train_generator:
    for i in range(10):

        plt.subplot(2, 5, i+1)
        plt.xticks()
        plt.yticks()
        plt.imshow(imagen[i])

    break
```

Estas funciones lo que hacen es generar variaciones de las imágenes iniciales como añadirle zoom o cambiarlas de orientación, cargar las que usaremos en el entrenamiento, y por último, cargar las imágenes de validación para comprobar si la red está clasificando bien los datos.

Además, añadimos un bucle al final de las funciones generadoras que nos enseña diez imágenes aleatorias que estará usando el generador para pasárselas a la red para que las clasifique, obteniendo así un ejemplo de las imágenes que usaremos.



- **Modelo**

En segundo lugar, teníamos que crear el modelo, para el proyecto creado desde cero, hemos terminado usando esta arquitectura de neuronas (en pirámide), ya que ha sido donde mejores resultados hemos obtenido.

Luego, en la función de activación, nos decidimos por la “softmax”, ya que realizando pruebas con otras como la “sigmoid” o poniendo el campo a “null” no conseguíamos mejores resultados. Además, la función softmax es la más usada comúnmente en redes neuronales para problemas de clasificación múltiple.

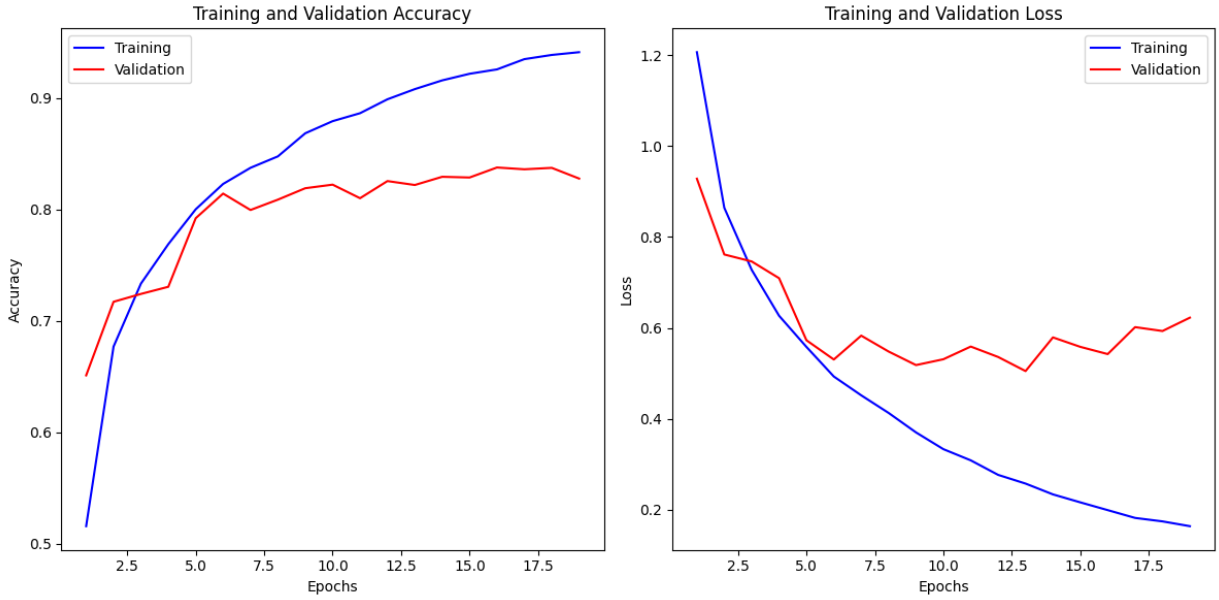
Finalmente, para el optimizador, hemos utilizamos el “Nadam”, ya que fue el que mejores resultados nos dio en comparación a otros como el “Adam”, “SGD” o “Adagrad” que no llegaron a mejorar los resultados.

```
#Modelo

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(125, 125, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

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

model.add(Dense(5, activation='softmax'))

model.compile(
    optimizer='nadam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])

model.summary()
```

Para el entrenamiento pusimos un máximo de 30 épocas y añadimos un EarlyStopping, que sirve para detener el entrenamiento automáticamente cuando la red deje de mejorar.

```
epochs = 30

es = EarlyStopping(monitor='val_accuracy', mode='max', verbose=1, patience=3, restore_best_weights=True)

history_of_train = model.fit_generator(
        train_generator,
        epochs=epochs,
        validation_data=validation_generator,
        callbacks=[es]
)

generatePlot(history_of_train)
```

Por último, generamos la gráfica para visualizar su comportamiento.



Por otro lado, para el proyecto donde usamos una red preentrenada, en la primera línea elegimos la opción de que el modelo haya sido preentrenado con más de un millón de imágenes.

Luego, creamos otro modelo con las capas de salida que le queramos añadir a la red preentenada, y juntamos estos dos modelos antes de compilar.

Por último, usamos la misma función de activación que en la red creada desde cero (softmax), pero con la diferencia de que el optimizador, en este caso, es Adam.

```python
#Uso del modelo vgg16 preentrenado
base_model = applications.VGG16(weights='imagenet', include_top=False, input_shape=(image_size, image_size, 3))

add_model = Sequential()
add_model.add(Flatten(input_shape=base_model.output_shape[1:]))
add_model.add(Dense(256, activation='relu'))
add_model.add(Dense(5, activation='softmax'))

model = Model(inputs=base_model.input, outputs=add_model(base_model.output))

# Compilar el modelo
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(learning_rate=0.0001),
              metrics=['accuracy'])

# Resumen del modelo
model.summary()
```

Para el entrenamiento, en este caso también usamos el EarlyStopping para dejar de entrenar la red cuando no siga mejorando. Además, entrenamos al modelo poniendo un máximo de cincuenta épocas.

```python
from keras.callbacks import EarlyStopping

# Configurar Early Stopping
early_stopping =EarlyStopping(monitor='val_accuracy', mode='max', verbose=1, patience=10, restore_best_weights=True)
```

+ Código + Markdown

```python
# Entrenar el modelo con Early Stopping

# Entrenar el modelo con Early Stopping
history_of_train =model.fit(
    train_generator,
    epochs=50,
    validation_data=validation_generator

)
```

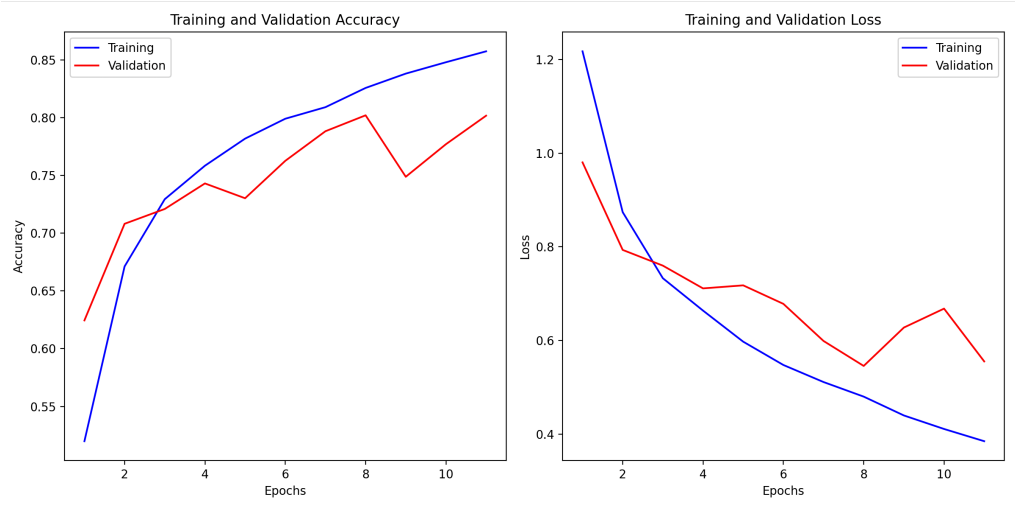
Por último, generamos la gráfica para ver el comportamiento de la red.

- **Observaciones**

En general, habiendo realizado este trabajo de dos formas diferentes, nos ha resultado más sencillo usar un modelo preentrenado y luego modificar las capas de salida para que se adapte a nuestro dataset, que crear el modelo desde cero e ir probando que arquitectura, optimizador y función de activación funcionan mejor para el dataset elegido, ya que estos últimos parámetros varían dependiendo del tipo de dataset que hayamos elegido. Además, observando las dos gráficas generadas, vemos que en el modelo preentrenado, obtenemos mejores resultados.

Por último, dejamos otras pruebas que hemos hechos con los dos proyectos donde hemos usado unos optimizadores diferentes y arquitecturas.

Proyecto desde cero con optimizador Adam:



Proyecto modelo preentrenado con optimizador Nadam(learning_rate=0.002):