Práctica evaluable Redes Neuronales Convolucionales (CNNs)

Deep Learning y sus aplicaciones

Máster en Ingeniería Informática

Autor:

Rebé Martín, Jorge

Fecha: 10 de enero de 2024

1. Introducción

En este documento se describe un trabajo consistente en la resolución de un problema de clasificación de imágenes utilizando técnicas de Deep Learning, en concreto, redes convolucionales.

Se trata de clasificar correctamente la especie de un ave que aparece en la imagen a clasificar, con un total de 7 especies de aves a distinguir. Estas especies de aves son cacatúas, loros, milanos negros, periquitos, cigüeñas, garzas, buitres leonados o alimoches.

2. Objetivos

El objetivo de este trabajo es la obtención de un modelo de deep learning que obtenga el *accuracy* más alto posible para un conjunto de test no conocido.

Para lograrlo, se utilizará alguno de los modelos de clasificación de imágenes de fastai[1] (PyTorch) preentrenados, se entrenarán con un *dataset* que se obtendrá utilizando técnicas de Image Scraping y se exportará el modelo que se utilizará para la evaluación sobre ese conjunto de test no conocido.

3. Creación del Dataset

Para entrenar el modelo, necesitamos un conjunto de datos. Para este caso de clasificación de especies de aves no existe ningún dataset público, òr lo que será creado desde cero. Para ello, se utilizarán técnicas de Web Scraping, después, se limpiarán los datos eliminando aquellas imágenes que no correspondan con ninguna clase y, por último, se dividirán las imágenes en conjuntos de entrenamiento, validación y prueba (*train, validation, test*).

3.1. Web Scraping Images

Para cada especie de ave, se realizará una búsqueda automática en Google Imágenes en la que se incluirá como búsqueda el nombre de la especie en inglés. Se obtendrán las 100 primeras y se guardarán en subcarpetas con el nombre de la especie en español, que será la etiqueta (*label*) de la imagen.

Las imágenes se han recogido utilizando técnicas de *image scraping* [2], utilizando la herramienta Selenium [3].

3.2. Limpieza de Datos

Debido a que las imágenes para el dataset han sido obtenidas a través de búsquedas automáticas en Google Imágenes, es natural que algunas de esas imágenes no correspondan con la clase, por lo que hay que realizar una limpieza sobre dichas imágenes y borrar las que no queremos utilizar.

Para ello, se ha utilizado la herramienta Image Cleaner[4], que muestra las imágenes por directorios y permite eliminarlas, pudiendo realizar así una limpieza sobre el dataset de forma bastante rápida.

3.3. División del Dataset en Entrenamiento-Validación-Test

Una vez obtenido el dataset, se ha dividido en entrenamiento, conjunto y prueba. El 70 % de las imágenes se han dedicado al entrenamiento, el 15 % a validación y el 15 % a test. La razón de esta división es porque de esta manera tendremos un subconjunto sobre el que entrenar el modelo (entrenamiento), otro subconjunto sobre el que evaluar durante el entrenamiento cómo de bien está generalizando el modelo (validación) y, por último, un subconjunto para ver cómo de bueno es el modelo una vez entrenado (test).

La división se ha realizado utilizando la biblioteca de Python split-folders [5], que divide de forma aleatoria un conjunto de datos en los tres conjuntos mencionados con el ratio deseado (en este caso: 0.70 para entrenamiento, 0.15 para validación y 0.15 para test), aportando una semilla para que la división sea reproducible.

4. Creación del modelo

Una vez creado el dataset y dividido en entrenamiento, validación y prueba, se ha procedido a elegir una arquitectura de redes neuronales convoluciones que se entrenará con nuestro dataset para resolver nuestro problema de clasificación de aves.

4.1. Arquitecturas de CNNs utilizadas

Se han probado varios modelos preentrenados aportados por fastai (PyTorch) [6], y a continuación se muestran algunos de ellos junto con su pérdida con los datos de test y la accuracy obtenida tras el entrenamiento y el fine tuning.

Modelo	Pérdida (test loss)	Accuracy
resnet152	0.31549	0.92500
resnext101_64x4d	0.29956	0.90833
resnext101_32x8d	0.22138	0.91666
resnext50_32x4d	0.20954	0.92500

Tabla 1: Algunas arquitecturas de CNNs utilizadas para entrenar el modelo, junto con su pérdida y accuracy sobre el conjunto de test

Por supuesto, se han utilizado más arquitecturas, como por ejemplo la de Google o VGGNet. Sin embargo, arquitecturas como esas son muy profundas y, por tanto, tienen un tiempo de entrenamiento por *epoch* extremadamente alto, lo cual hace muy difícil realizar varios entrenamientos variando tasa de aprendizaje y algún otro parámetro para poder obtener un modelo que funcione bien en varios de esos entrenamientos.

4.2. Elección de arquitectura de CNN

La arquitectura del modelo modelo entregado ha sido resnext50_32x4d [7, 8, 9], que es parecida a la de ResNet, pero se añade una nueva dimensión: la cardinalidad, que define el tamaño del conjunto de las transformaciones. Utilizando aproximadamente el mismo número de parámetros que una ResNet, consigue mejores resultados.

No hay nada especial con esta arquitectura, pero tras probar unas cuantas, con esta se obtuvieron unos resultados razonablemente buenos sobre el conjunto de test y, al entrenar bastante rápido (unos dos minutos por epoch), se pudieron realizar varios entrenamientos distintos para comprobar que los buenos resultados no eran casualidad.

5. Entrenamiento del modelo

Una vez elegido un tipo de CNN, es el momento de entrenarlo con el dataset que hemos creado. Durante el primer entrenamiento, el detallado en esta sección, se entrenan los pesos de la última capa. Después, habrá una etapa de *fine-tuning* en la que se ajustarán los pesos de todas las capas, en la Sección 6.

En la Figura 1 se muestra la primera fase de entrenamiento, en la que se entrena utilizando la función fit_one_cycle. Con este entrenamiento sólo se modifican los pesos de la última capa, mientras que para el resto de las capas se mantienen con los valores preentrenados.

Entrenamiento del modelo

```
learn = cnn_learner(dls, models.resnext50_32x4d, metrics=accuracy)

cbs_train = [EarlyStoppingCallback(monitor='valid_loss', patience=3), ReduceLROnPlateau(monitor='valid_loss', min_delta=0.10, factor=3.5)]

learn.fit_one_cycle(10, cbs=cbs_train)
```

epoch	train_loss	valid_loss	accuracy	time
0	2.845170	1.167606	0.575472	01:23
1	1.911437	0.277185	0.915094	01:23
2	1.361806	0.350570	0.886792	01:23
3	1.024081	0.334124	0.905660	01:22
4	0.798929	0.344430	0.886792	01:22

```
Better model found at epoch 0 with valid_loss value: 1.1676058769226074.
Better model found at epoch 1 with valid_loss value: 0.27718546986579895.
Epoch 2: reducing lr to 0.000284118704360637
Epoch 3: reducing lr to 0.00026324607492628097
Epoch 4: reducing lr to 0.00022155734906567835
No improvement since epoch 1: early stopping
```

Figura 1: Resultados del primer entrenamiento

5.1. Callbacks

Los modelos preentrenados de fastai son capaces de entrenar muy rápidamente y es posible que en pocos epochs los valores de la función de pérdida sobre el conjunto de validación y la accuracy alcancen un valor muy bueno, y en los siguientes, comience a empeorar.

Por esto, es necesario utilizar métodos que permitan la modificación de hiperparámetros durante el aprendizaje para evitar el sobreajuste. Estos métodos son los llamados Callbacks [10]. En esta

práctica se han utilizado los siguientes:

- **EarlyStoppingCallback**: Para el entrenamiento cuando una métrica ha dejado de mejorar durante un determinado número de epochs. En el caso del entrenamiento, se monitoriza el valor de la función de pérdida sobre el conjunto de validación, y una paciencia de 3 epochs. Esto quiere decir que si durante 3 epochs seguidos no mejora ese valor, se parará el entrenamiento.
- **SaveModelCallback**: Guarda el mejor modelo durante el entrenamiento según un valor que se monitoriza. En el caso del entrenamiento, se guardará el modelo que tenga el menor valor de la función de pérdida sobre el conjunto de validación.
- **ReduceLROnPlateau**: Divide la tasa de aprendizaje por un factor cuando una métrica que se está monitorizando no ha mejorado una cantidad determinada durante un número de epochs.

6. Fine Tuning

Para obtener el mejor modelo posible no es suficiente con modificar los pesos de la última capa, sino que nos interesa actualizar los parámetros del resto de capas también. Para ello, utilizaremos la función fine_tune[11, 12] de fastai.

Esta función entrena por defecto durante 1 epoch, y después, desbloquea los parámetros del resto de capas y ajusta los parámetros del resto de capas con una tasa de aprendizaje más pequeña.

En la Figura 2 se muestran los resultados del fine-tuning sobre el modelo.

Fine tuning

```
learn.load('trainmodel')

cbs_test = [EarlyStoppingCallback(monitor='valid_loss', patience=3), ReduceLROnPlateau(monitor='valid_loss', min_delta=0.05, factor=7.5)]

learn.fine_tune(10, cbs=cbs_test)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.570766	0.280880	0.915094	01:29

```
Better model found at epoch 0 with valid_loss value: 0.2808797359466553.
```

epoch	train_loss	valid_loss	accuracy	time
0	0.525178	0.282170	0.924528	01:51
1	0.411254	0.248094	0.915094	01:51
2	0.347964	0.254963	0.933962	01:51
3	0.311151	0.282073	0.905660	01:51
4	0.270180	0.373946	0.896226	01:51

```
Better model found at epoch 0 with valid_loss value: 0.2821703851222992.
Better model found at epoch 1 with valid_loss value: 0.2480938732624054.
Epoch 1: reducing lr to 9.948484887789789e-05
Epoch 2: reducing lr to 0.00013273764413621942
Epoch 3: reducing lr to 0.00012846116636853667
Epoch 4: reducing lr to 0.00011148694561507197
No improvement since epoch 1: early stopping
```

Figura 2: Resultados del primer entrenamiento

6.1. Callbacks

Los callbacks utilizados para el fine tuning en esta etapa de entrenamiento de fine-tuning son los mismos que los utilizados en la Subsección 5.1, pero modificando algunos parámetros.

7. Prueba del Modelo

Una vez entrenado el modelo, se ha evaluado contra el subconjunto destinado para test (el 15 % del dataet, como se explicó en la Sección 3). Se ha evaluado el modelo sobre este conjunto dos veces: una tras el el primer entrenamiento (ver Sección 5) y otra tras el fine-tuning (ver Sección 6).

En la Figura 3 se puede ver la matriz de confusión sobre el subconjunto de test tras el primer entrenamiento del modelo (sólo ajuste de los pesos de la última capa).

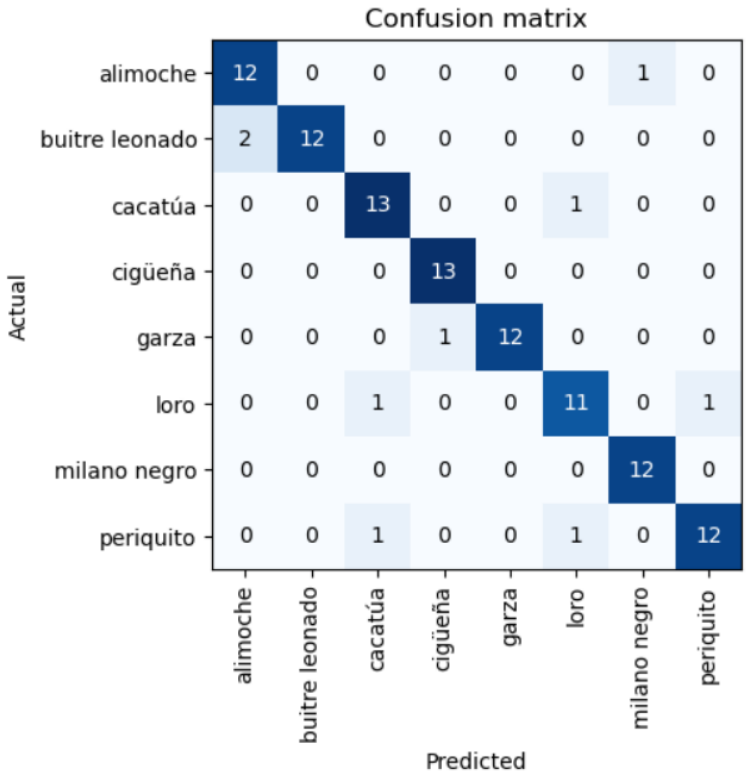

Figura 3: Matriz de confusión en la clasificación del subconjunto de test tras el primer entrenamiento.
Test loss: 0.27585041522979736 - Accuracy: 0.9166666865348816

En la Figura 4 se puede ver la matriz de confusión sobre el subconjunto de test tras el fine-tuning (ajuste de los pesos de todas las capas).

Los resultados tras el fine tuning son peores para algunas clases, como se puede ver en la matriz de confusión, pero en general la accuracy mejora y también lo hace el valor de la función de pérdida, por lo que el modelo entregado será el obtenido tras el fine tuning.

8. Conclusiones

A lo largo de este trabajo se ha desarrollado un pequeño proyecto de Deep Learning desde cero, incluyendo la creación del conjunto de datos utilizando técnicas de image scraping. Después, se han utilizado varias arquitecturas de redes neuronales convolucionales preentrenadas, y se ha aprendido sobre ellas además de comprobado su efectividad clasificando imágenes con pocos epochs de entrenamiento.

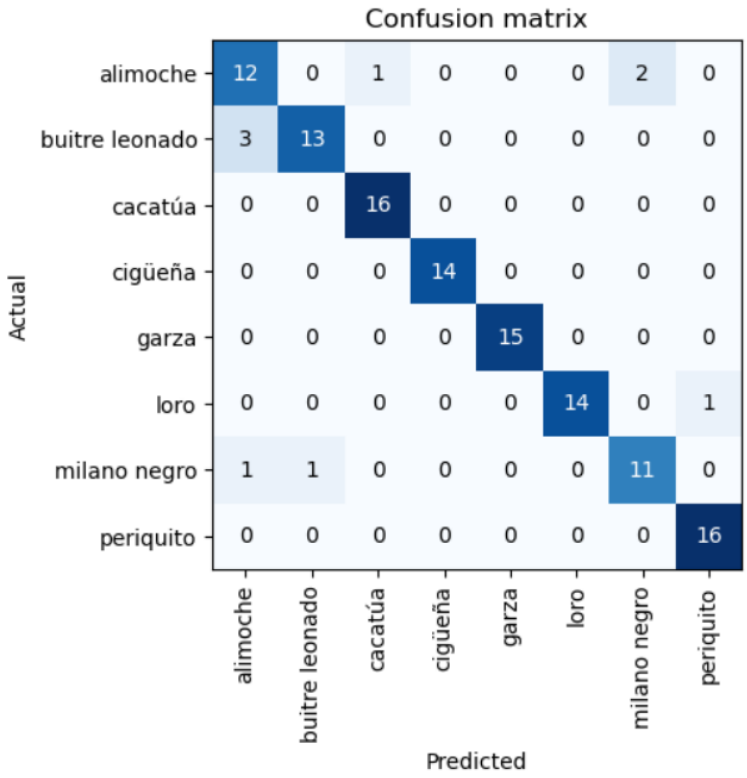

Figura 4: Matriz de confusión en la clasificación del subconjunto de test tras el fine-tuning.
Test loss: 0.20954863727092743 - Accuracy: 0.925000011920929

Referencias

[1] fastai. Welcome to fastai. https://docs.fast.ai/. Último acceso el 09/01/2024.

[2] Deepnote. Web Scraping Images with Python and Selenium. https://deepnote.com/@dennis-dfa8/Web-Scraping-Images-5e2baaa2-d24f-4d47-a124-687639818a7f. Último acceso el 03/01/2024.

[3] Selenium. Web Scraping Images with Python and Selenium. https://www.selenium.dev/documentation/webdriver/. Último acceso el 09/01/2024.

[4] Joe Dockrill. jmd_imagescraper.imagecleaner. https://joedockrill.github.io/jmd_imagescraper/imagecleaner.html. Último acceso el 03/01/2024.

[5] Johannes Filter. split-folders. https://github.com/jfilter/split-folders. Último acceso el 03/01/2024.

[6] PyTorch. Models and pre-trained Weights. https://pytorch.org/vision/stable/models.html. Último acceso el 07/01/2024.

[7] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017.

[8] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *CoRR*, abs/1901.06032, 2019.

[9] Jeremy Jordan. Common architectures in convolutional neural networks. https://www.jeremyjordan.me/convnet-architectures/. Último acceso el 09/01/2024.

[10] fastai. Tracking callbacks. https://docs.fast.ai/callback.tracker.html. Último acceso el 03/01/2024.

[11] fastai. Hyperparam schedule - fine_tune. https://docs.fast.ai/callback.schedule.html#learner.fine_tune. Último acceso el 09/01/2024.

[12] fastai. Hyperparam schedule - fine_tune. https://github.com/fastai/fastai/blob/master/fastai/callback/schedule.py#L161. Último acceso el 09/01/2024.