Visión por Computador

Práctica 2: CNN

Antonio Coín Castro

Curso 2019-20

Estructura del código

El código desarrollado se divide en dos archivos: `p2_cifar.py` y `p2_resnet.py`, correspondientes a las dos partes en las que se divide la práctica. Junto a estos archivos se proporcionan otros de idéntico nombre pero en formado *notebook*, listos para ser ejecutados en *Google Colab*. Hay una serie de consideraciones genéricas a tener en cuenta:

- Tras mostrar una gráfica, el programa quedará a la espera de una pulsación de tecla para continuar con su ejecución (incluso si se cierra manualmente la ventana del *plot*).
- Hay algunos parámetros globales (editables) al inicio del programa para modificar ciertos comportamientos de algunas funciones.
- El programa desarrollado en cada caso va ejecutando desde una función `main` los distintos apartados de la práctica uno por uno, llamando a las funciones que sean necesarias para mostrar ejemplos de funcionamiento.
- Todas las funciones están comentadas y se explica el significado de los parámetros.

Apartado 1: red convolucional básica con CIFAR100

En este apartado perseguimos desarrollar una red convolucional básica para clasificar imágenes del conjunto de datos CIFAR100, disponible en *Keras*. Utilizamos la función `load_data` para cargar en memoria los datos de entrenamiento y test en el formato adecuado. Es importante notar que tratamos los vectores de clases como matrices binarias (mediante la función `to_categorical`) en las que hay un 1 en la posición de la clase correspondiente, y un 0 en el resto de posiciones. Aprovechamos para quedarnos solo con las **25 primeras clases** de las 100 disponibles, reduciendo así el tamaño del problema.

Para trabajar con las imágenes de entrada haremos uso de la clase `ImageDataGenerator`, creando cuando sea necesario un *datagen* que permitirá el flujo de imágenes. Es en esta construcción cuando podemos reservar un 10 % de datos de entrenamiento para validación, mediante el argumento `validation_split = 0.1`.

Disponemos de una función `compile` que se encarga de definir el optmizador y compilar un modelo. El optmizador por defecto será *Stochastic Gradient Descent (SGD)* con un *learning rate* de 0.01, y la función de pérdida a minimizar será la función

`categorical_crossentropy`. Este término es otra forma de referirse a la pérdida logarítmica multi-clase [**1**], que mide el desempeño de un modelo de clasificación en el que las predicciones son una probabilidad.

Para entrenar un modelo tenemos la función `train`, cuya implementación se puede resumir en el siguiente trozo de código:

```
# Dividimos en datos de entrenamiento y de validación
train_gen = datagen.flow(x_train, y_train,
                         batch_size = BATCH_SIZE,
                         subset = 'training')
val_gen = datagen.flow(x_train, y_train,
                       batch_size = BATCH_SIZE,
                       subset = 'validation')

# Entrenamos el modelo
return model.fit_generator(train_gen,
                    epochs = EPOCHS,
                    steps_per_epoch = len(x_train) * 0.9 / BATCH_SIZE,
                    validation_data = val_gen,
                    validation_steps = len(x_train) * 0.1 / BATCH_SIZE,
                    callbacks = callbacks_list)
```

El parámetro `batch_size` controla el número de imágenes que se procesan antes de actualizar el modelo (podríamos decir que es la unidad mínima de aprendizaje), que por defecto es 64. Los parámetros `steps_per_epoch` y `validation_steps` controlan el número de *batches* que se procesan antes de finalizar una época y antes de concluir la validación, respectivamente.

Hemos añadido dos *callbacks* de la clase `EarlyStopping`, que nos permiten detener el entrenamiento cuando no se haya mejorado el error o la precisión de validación en un número de épocas determinado (controlado por el parámetro `PATIENCE`, con valor por defecto de 10), recuperando los mejores pesos obtenidos hasta el momento:

```
early_stopping = EarlyStopping(monitor = 'val_loss', # igual con val_acc
                               patience = PATIENCE,
                               restore_best_weights = True)
```

También se añade un *callback* de la clase `ModelCheckpoint` para ir guardando los mejores pesos obtenidos hasta el momento en un fichero si se quiere.

Para evaluar el modelo utilizamos la función `evaluate`, a la que le pasamos los datos de test. También existe una función `execute` que se encarga de compilar, entrenar y evaluar un modelo. Por último, las funciones `show_stats` y `show_evolution` nos permiten mostrar gráficas y estadísticas del entrenamiento y la evaluación del modelo.

Notamos que damos la posibilidad guardar los pesos del modelo con el parámetro `save_w`, y también el historial de entrenamiento y las estadísticas de evaluación con `save_h` y `save_s`. Esto nos permitiría seguir entrenando un modelo a partir de unos pesos precargados, o comparar dos modelos sin tener que volver a entrenarlos y evaluarlos. De hecho, disponemos de una función `compare` que realiza justamente esta función.

Modelo basenet

Definimos el modelo de referencia BaseNet que nos piden, cuyo bloque convolucional tiene la estructura `conv-maxpool-conv-maxpool`.

```
def basenet_model():
    model = Sequential()
    model.add(Conv2D(6,
                     kernel_size = (5, 5),
                     activation = 'relu',
                     input_shape = (32, 32, 3)))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Conv2D(16,
                     kernel_size = (5, 5),
                     activation='relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Flatten())
    model.add(Dense(50,
                    activation = 'relu'))
    model.add(Dense(25,
                    activation = 'softmax'))

    return model, "basenet"
```

Se trata de una red neuronal convolucional cuya salida es un vector de 25 entradas, donde la entrada i-ésima representa la probabilidad de que la imagen evaluada pertenezca a la clase i-ésima. Si ejecutamos el modelo, obtenemos los siguientes resultados:

```
---------- BASENET MODEL EVALUATION ----------
Test loss: 2.017395290565491
Test accuracy: 0.4452
```

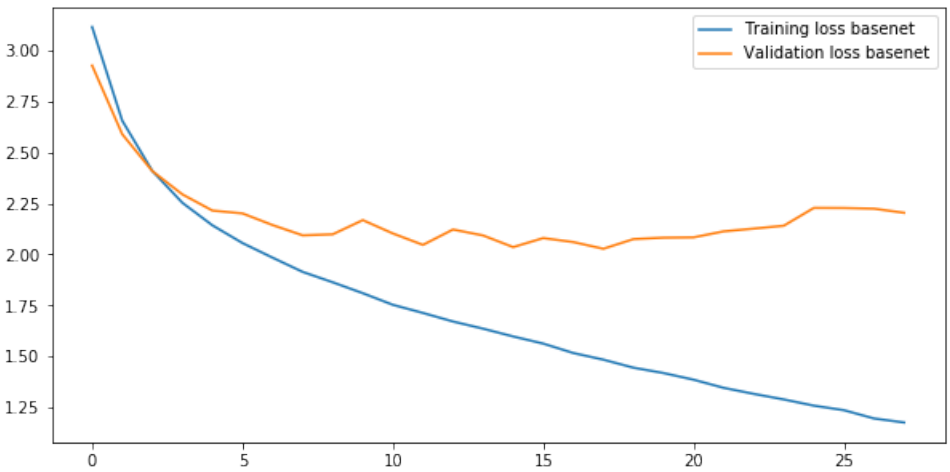


Figura 1: Estadísticas de pérdida en basenet.

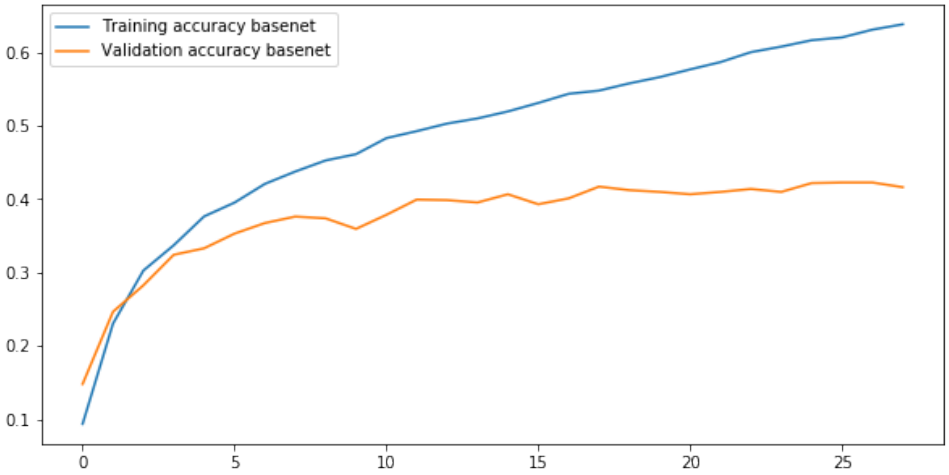


Figura 2: Estadísticas de precisión en basenet.

El modelo se ha entrenado durante 26 épocas, obteniéndose los mejores pesos en la época 16. A partir de esta época, el error de validación comienza a subir, mientras que la precisión en validación se mantiene constante o incluso disminuye. Observamos que se produce el fenómeno de *overfitting*, pues tanto la pérdida como la precisión llegan a ser mucho mejores en el entrenamiento.

Nota: al ejecutar en local pueden aparecer gráficas de pérdida en validación inesperadas, por un problema en la versión de Keras 2.3.1.

Modelo basenet mejorado

En este caso perseguimos modificar el modelo anterior para obtener una mayor precisión en la clasificación de las imágenes. Se proponen una serie de mejoras, que analizamos a continuación. En todos los caso utilizamos la técnica de *early-stopping* para detener el entrenamiento en el punto en que el error de validación comienze a subir o la precisión en validación comience a disminuir.

Normalización inicial de los datos

Queremos en primer lugar conseguir datos con la misma media (0) y la misma varianza (1), disminuyendo así la complejidad del problema. Para ello, añadimos al *datagen* de entrenamiento los parámetros `featurewise_center = True` y `featurewise_std_normalization = True`, para tipificar los datos de entrada restándoles su media y dividiendo por su desviación típica. Posteriormente, debemos aplicar la misma normalización a los datos de test.

```
# Estandarizamos datos de entrenamiento y test
datagen.fit(x_train)
datagen.standardize(x_test)
```

Si volvemos a ejecutar el modelo de referencia con este preprocesamiento de datos, obtenemos una precisión en la evaluación similar, e incluso un poco por debajo. Sin embargo, es conveniente siempre tener los datos normalizados.

```
---------- BASENET MODEL EVALUATION ----------
Test loss: 2.038418185043335
Test accuracy: 0.4252
```

Capas de normalización

La siguiente mejora consiste en añadir capas `BatchNormalization` después de cada capa convolucional y totalmente conectada (excepto la última), para ir normalizando la salida en cada etapa e intentar mantener la media cercana a 0 y la varianza cercana a 1. Estas capas tipifican los datos de cada *batch* (en cada componente), y después realizan una transformación lineal:

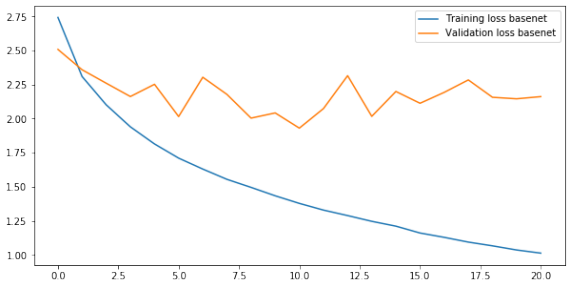
$$y_i^{(k)} = \gamma^{(k)} \widetilde{x}_i^{(k)} + \beta^{(k)},$$

donde el superíndice (k) denota la componente k-ésima, x_i el dato $i-$ésimo del *batch*, $\widetilde{x}_i^{(k)}$ la tipificación en la k-ésima componente del dato i-ésimo, y $\gamma^{(k)}, \beta^{(k)}$ son parámetros que se aprenden durante el entrenamiento.

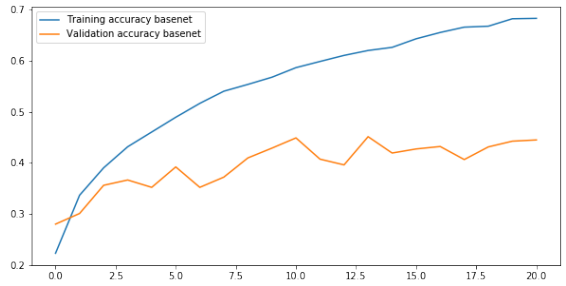
El modelo queda como sigue:

```
def basenet_model():
    model = Sequential()
    model.add(Conv2D(6, kernel_size = (5, 5),
                     use_bias = False,
                     input_shape = (32, 32, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Conv2D(16, kernel_size = (5, 5),
                     use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Flatten())
    model.add(Dense(50, use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(25,
                    activation = 'softmax'))

    return model, "basenet"
```



(a) Pérdida en entrenamiento y validación



(b) Precisión en entrenamiento y validación

Figura 3: Estadísticas de entrenamiento de basenet con capas de normalización.

Hemos añadido las capas de normalización **antes de la activación**, pero también podrían haberse puesto después. Se han hecho pruebas de ambas formas y se ha observado que no hay diferencias significativas, por lo que se ha optado por la primera variante. El resultado obtenido al ejecutar el modelo mejora el que teníamos sin las capas de normalización:

```
---------- BASENET MODEL EVALUATION ----------
Test loss: 1.8317416580200194
Test accuracy: 0.4784
```

Un detalle a tener en cuenta es el empleo del parámetro `use_bias = False` en las capas que preceden a la normalización. Le decimos a estas capas que no añadan el término de sesgo o *bias* (una constante aditiva), pues en las capas de normalización ya se contempla este término (el parámetro β). De hecho, como las capas de normalización trasladan los valores por su media, cualquier constante aditiva previa se cancelaría.

Data augmentation

De nuevo para evitar que nuestra red se ajuste demasiado a los datos de entrenamiento, introducimos la técnica de *data augmentation*. Se trata de sustituir aleatoriamente algunas imágenes de entrenamiento por otras con ligeras perturbaciones y/o transformaciones aleatorias. Por ejemplo, podemos considerar imágenes con rotaciones, traslaciones, cizallas, zoom, etc.

Notamos que esto solo lo hacemos para las imágenes de entrenamiento, pues pretendemos aumentar la capacidad del modelo de generalizar y aprender características que no sean propias de las imágenes concretas con las que se entrena. Para implementarlo, utilizamos

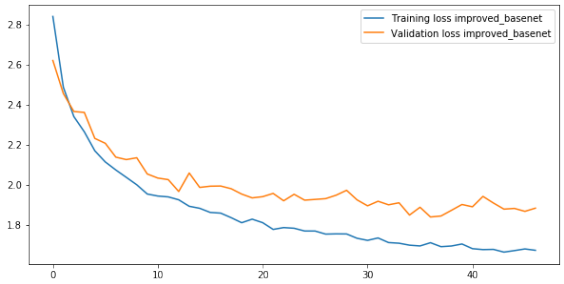
la clase `ImageDataGenerator` para crear un *datagen* de entrenamiento con los parámetros que queremos (contando la normalización inicial que ya teníamos):

```
datagen = ImageDataGenerator(featurewise_center = True,
                             featurewise_std_normalization = True,
                             width_shift_range = 0.1,
                             height_shift_range = 0.1,
                             zoom_range = 0.2,
                             horizontal_flip = True,
                             validation_split = 0.1)
```

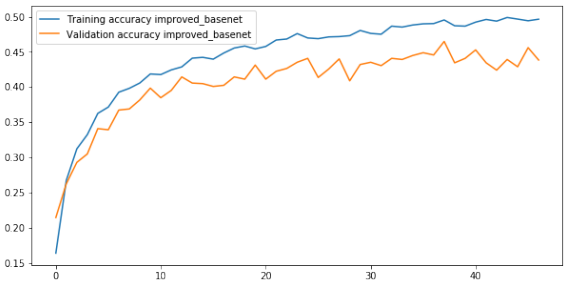
En este caso hemos permitido traslaciones horizontales y verticales, volteos horizontales y zoom. Los parámetros numéricos controlan la "intensidad" de la transformación. Por ejemplo, con `zoom_range = 0.2` estamos haciendo un aumento aleatorio en el intervalo [1 - 0.2, 1 + 0.2].

```
---------- BASENET MODEL EVALUATION ----------
Test loss: 1.741819490814209
Test accuracy: 0.482
```

Observamos a continuación como, a parte de mejorar la precisión en la evaluación, conseguimos como pretendíamos evitar el *overfitting*: las curvas de pérdida y precisión en validación se pegan más a las de entrenamiento.



(a) Pérdida en entrenamiento y validación



(b) Precisión en entrenamiento y validación

Figura 4: Estadísticas de entrenamiento de basenet con *data augmentation*.

Aumentar la profundidad del modelo

Añadimos finalmente más capas convolucionales, con un número de canales de salida creciente, y más capas totalmente conectadas. Evitamos hacer *max pooling* después de

cada capa convolucional para no perder excesiva infomación. En concreto, tenemos dos bloques `conv-conv-maxpool` y dos capas totalmente conectadas antes de la última capa de activación *softmax*.

Hemos decidido mantener el tamaño de las imágenes en la primera convolución de cada bloque mediante el parámetro `padding = 'same'`, para no perder rápidamente dimensión en las imágenes. Además, reducimos el tamaño de los *kernels* de convolución a 3×3, ya que ahora tenemos dos convoluciones seguidas.

```
def improved_basenet_model():
    """Devuelve el modelo BaseNet mejorado."""

    model = Sequential()

    model.add(Conv2D(32, padding = 'same',
                     kernel_size = (3, 3),
                     use_bias = False,
                     input_shape = (32, 32, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(32, kernel_size = (3, 3),
                     use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, padding = 'same',
                     kernel_size = (3, 3),
                     use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(64, kernel_size = (3, 3),
                     use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(512, use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(256, use_bias = False))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dropout(0.5))
    model.add(Dense(25,
                    activation = 'softmax'))

    return model, "improved_basenet"
```

Aunque no se mencionaba en el guión, también se han añadido capas `Dropout`. Estas capas desactivan una fracción de neuronas en el entrenamiento de forma aleatoria, para obligar a la red a individualizar el aprendizaje y prevenir el *overfitting*. Se trata pues de una técnica de regularización. El parámetro que aceptan estas capas es la fracción de unidades de entrada que se desactivan (siendo 0 ninguna y 1 todas) en la capa siguiente.

Podemos visualizar nuestro modelo final en la siguiente tabla:

Layer No.	Layer Type	Kernel size	Input|Output dim.	Input|Output channels
1	Conv2D	3	32|32	3|32
2	BatchNorm	-	32|32	-
3	Relu	-	32|32	-
4	Conv2D	3	32|30	32|32
5	BatchNorm	-	30|30	-
6	Relu	-	30|30	-
7	MaxPooling2D	2	30|15	-
8	Dropout (0.25)	-	15|15	-
9	Conv2D	3	15|15	32|64
10	BatchNorm	-	15|15	-
11	Relu	-	15|15	-
12	Conv2D	3	15|13	64|64
13	BatchNorm	-	13|13	-
14	Relu	-	13|13	-
15	MaxPooling2D	2	13|6	-
16	Dropout (0.25)	-	6|6	-
17	Dense	-	2304|512	-
18	BatchNorm	-	512|512	-
19	Relu	-	512|512	-
20	Dense	-	512|256	-
21	BatchNorm	-	256|256	-
22	Relu	-	256|256	-
23	Dropout (0.5)	-	256|256	-
24	Dense	-	256|25	-

Juntando todas las mejoras anteriores y entrenando este nuevo modelo durante 84 épocas, obtenemos los mejores pesos en la época 74, que nos dan las siguientes estadísticas de entrenamiento y evaluación:

```
---------- IMPROVED_BASENET MODEL EVALUATION ----------
Test loss: 0.9160149734497071
Test accuracy: 0.7256
```

Vemos como la función de pérdida toma valores más pequeños que antes, tanto en entrenamiento como en validación y test. Del mismo modo, la precisión aumenta en todas las etapas, obteniendo finalmente un nada desdeñable 72 % de precisión en la clasificación de imágenes (considerando la simplicidad del modelo y el tamaño de la base de datos).

Podemos concluir que las mejoras realizadas a nuestro modelo han surtido efecto sin aumentar notablemente el tiempo de entrenamiento (se ha pasado de unos 2-3s por época

a 10-12s por época). Además, hemos logrado reducir el *overfitting*, pues vemos como las gráficas de entrenamiento y validación no difieren tanto como antes.

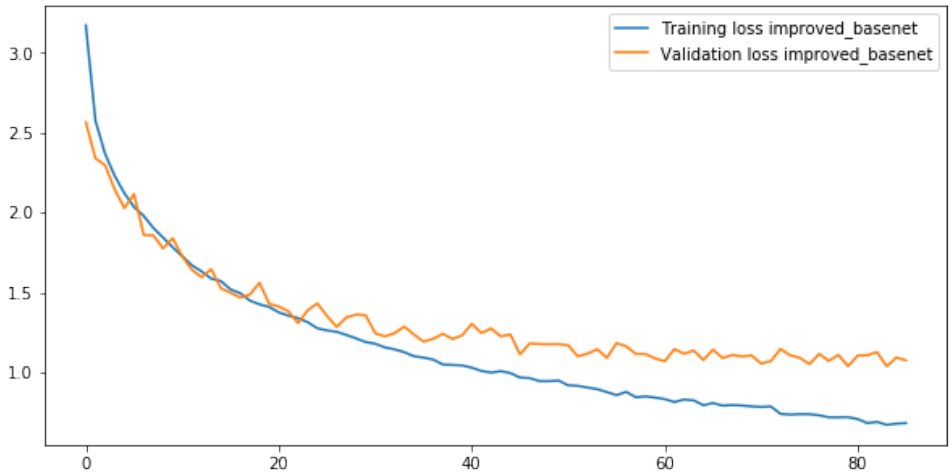


Figura 5: Estadísticas de pérdida en basenet mejorado.

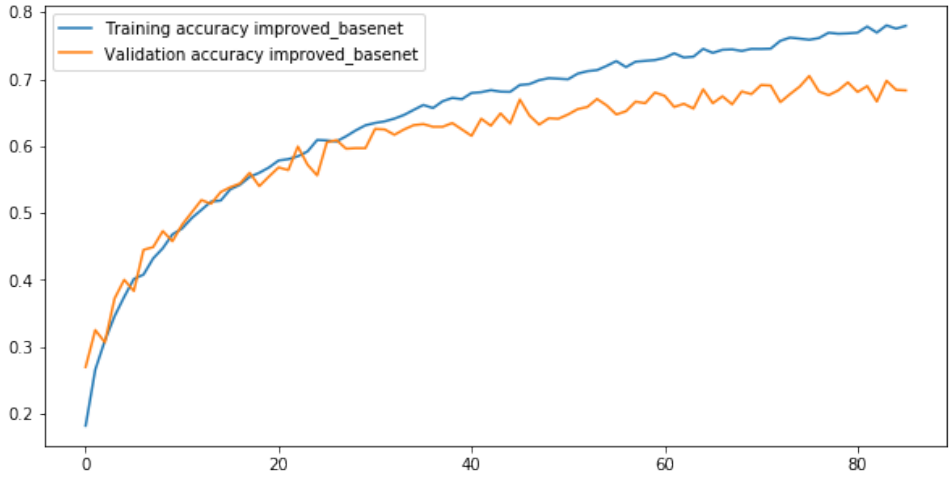


Figura 6: Estadísticas de precisión en basenet mejorado.

Comparación de modelos

Si guardamos las estadísticas de entrenamiento y evaluación de nuestros dos modelos, podemos realizar gráficas comparativas en validación de ambos mediante la función `compare`, que lee los datos de fichero y construye dichas gráficas. El número de épocas en ambos modelos es distinto debido al *early-stopping*, y apreciamos que se alcanza el "punto de saturación" en épocas muy distintas.

Como ya vimos, el segundo modelo alcanza una precisión bastante mayor, y también consigue disminuir el valor de la función de error. Incluso si entrenásemos durante el mismo número de épocas, se puede comprobar que el modelo mejorado seguiría siendo superior en todos los sentidos.

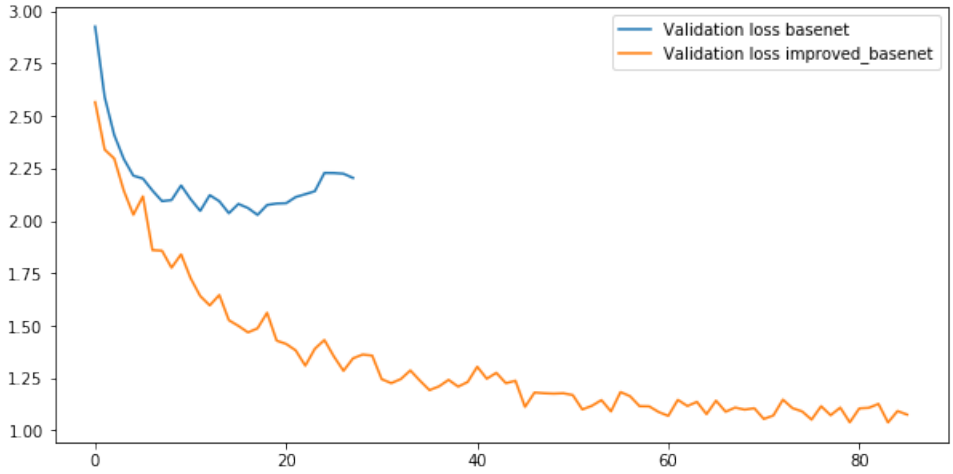


Figura 7: Comparación de pérdida en validación de basenet y basenet mejorado.

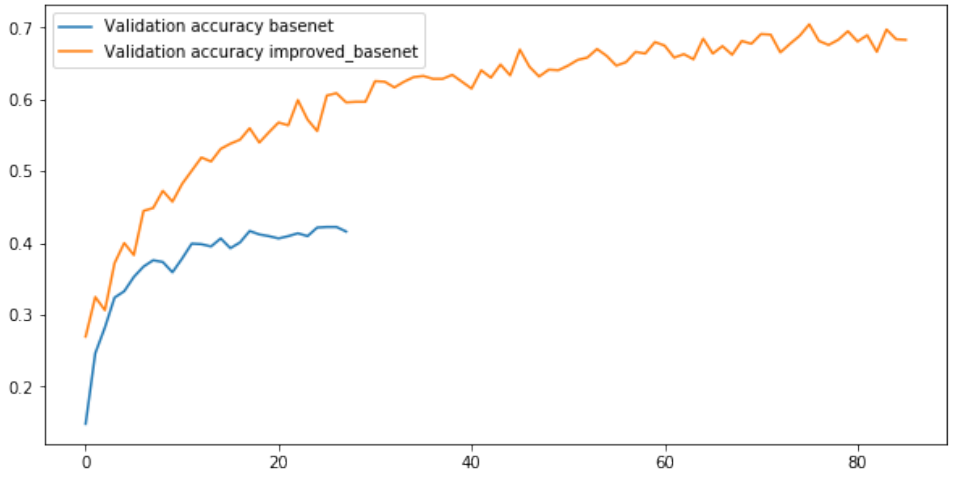


Figura 8: Comparación de precisión en validación de basenet y basenet mejorado.

Apartado 2: red preentrenada ResNet50 con Caltech-UCSD

En este apartado cambiamos el enfoque, y lo que pretendemos es usar una red preentrenada y adaptarla a nuestro problema de clasificación. Elegimos la red ResNet50 [**2**] y trabajamos sobre el conjunto de datos Caltech-UCSD [**3**], que consta de 3000 imágenes de entrenamiento y 3033 imágenes de test, con 200 clases distintas.

La filosofía detrás de usar una red preentrenada es que las capas iniciales consiguen capturar características generales de las imágenes, y son las últimas capas las que se adaptan al problema concreto de clasificación. Así, si eliminamos las últimas capas de una red preentrenada y concatenamos un modelo de nuestra elección, podemos ahorrar tiempo de entrenamiento y obtener una buena precisión en la clasificación.

Tenemos la siguiente función para realizar predicciones (pasar las imágenes por la red) sobre un modelo, usando un *datagen*:

```
def predict_gen(model, datagen, x):
    return model.predict_generator(datagen.flow(x,
                                                batch_size = 1,
                                                shuffle = False),
                                   verbose = 1,
                                   steps = len(x))
```

También disponemos de una función análoga a `evaluate` para evaluar un modelo usando un *datagen*, llamada `evaluate_gen`.

Extracción de características

Como primer enfoque utilizamos ResNet50 como **extractor de características**. Esto quiere decir que eliminamos su última capa de activación *softmax*, y utilizamos la salida de la capa anterior como entrada a un nuevo modelo. Podemos ver dicha salida como una colección de características de las imágenes que la red ha extraído.

El proceso es el siguiente: obtenemos el modelo de ResNet50 preentrenado en la base de datos de ImageNet, le quitamos la última capa, y pasamos nuestras imágenes de entrenamiento y test por la red, obteniendo como salida en cada caso un vector de 2048 características. Una vez tenemos estas características, las usamos como entrada a un modelo muy simple con dos o tres capas. Por último, compilamos, entrenamos y ejecutamos nuestro modelo.

Un detalle a tener en cuenta es que al ejecutar el modelo de ResNet50 sobre nuestras imágenes, debemos preprocesar los datos de forma adecuada. Esto se hace al construir el *datagen* de imágenes, pasándole la función `keras.applications.resnet50.preprocess_input`.

El proceso más básico sería añadir únicamente al final una capa totalmente conectada de tamaño 200 y activación *softmax*:

```
def base_model():
    model = Sequential()
    model.add(Dense(200, activation = 'softmax'))
    return model
```

```
def feature_extraction_basic():
    # Cargamos los datos
    x_train, y_train, x_test, y_test = load_data()

    # Creamos un generador para entrenamiento y otro para test
    datagen_train = ImageDataGenerator(preprocessing_function = preprocess_input)
    datagen_test = ImageDataGenerator(preprocessing_function = preprocess_input)

    # Usamos ResNet50 preentrenada en ImageNet sin la última capa
    resnet = ResNet50(weights = 'imagenet',
                      include_top = False,
                      pooling = 'avg')

    # Extraemos características de las imágenes con el modelo anterior
    features_train = predict_gen(resnet, datagen_train, x_train)
    features_test = predict_gen(resnet, datagen_test, x_test)

    # Ejecutamos un modelo con las características extraídas como entrada
    score, hist = execute(base_model, 10,
                          features_train, y_train, features_test, y_test)
```

Notamos que realizamos un *Global pooling average* para que la salida de ResNet50 (ya sin la última capa) sea un vector de características. Otra opción sería usar `Flatten()`.

Con el objetivo de comparar los nuevos modelos que definamos, ejecutamos el ejemplo anterior entrenando durante 10 épocas únicamente la capa nueva que hemos añadido, y obtenemos los siguientes resultados:

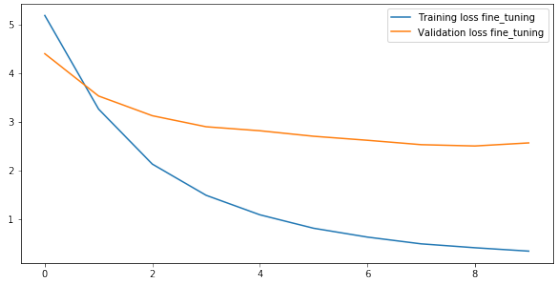
```
---------- BASE_MODEL EVALUATION ----------
Test loss: 2.897768755805652
Test accuracy: 0.3003626772172766
```

Vemos en la figura siguiente que no tiene mucho sentido entrenar más épocas, porque ya hay un gran sobreajuste y el error y la precisión de validación comienzan a empeorar.

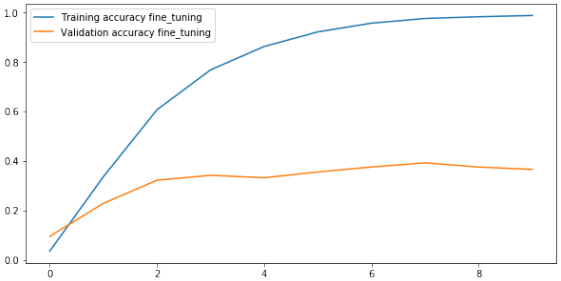
Ahora probamos dos alternativas para definir nuestro modelo: una en la que añadimos dos capas totalmente conectadas, y otro en el que además añadimos una capa convolucional. El primero de ellos sería el siguiente, en el que añadimos una capa de `Dropout` bastante agresivo para prevenir que el sobreajuste se produzca muy rápidamente.

```
def fc_model():
    model = Sequential()
    model.add(Dense(1024, activation = 'relu',
                    input_shape = (2048,)))
    model.add(Dense(512, activation = 'relu'))
    model.add(Dropout(0.7))
    model.add(Dense(200, activation = 'softmax'))

    return model
```



(a) Pérdida en entrenamiento y validación



(b) Precisión en entrenamiento y validación

Figura 9: Estadísticas de entrenamiento del modelo básico.

También podemos analizarlo en forma de tabla:

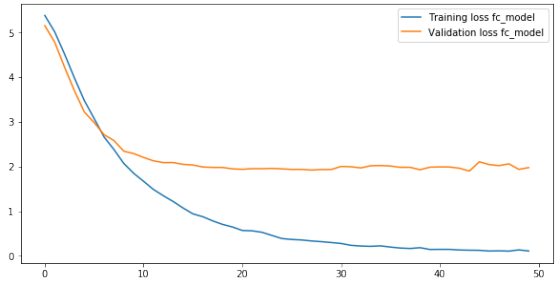
Layer No.	Layer Type	Kernel size	Input|Output dim.	Input|Output channels
1	Dense	-	2048|1024	-
2	Relu	-	1024|1024	-
3	Dense	-	1024|512	-
4	Relu	-	512|512	-
5	Dropout (0.7)	-	512|512	-
6	Dense	-	512|200	-

Se ha evitado añadir las capas de regularización de `BatchNorm` porque las etapas finales del modelo podemos suponer que los datos vienen más o menos normalizados. Si las añadimos, el desempeño del modelo era bastante peor. Si ejecutamos este modelo durante 50 épocas usando el código mostrado anteriormente, obtenemos los siguientes resultados:

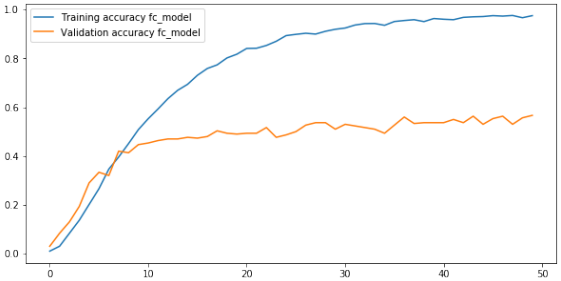
```
---------- FC_MODEL EVALUATION ----------
Test loss: 2.418268212735987
Test accuracy: 0.45268710853241473
```

Como vemos, se mejora en todos los sentidos al modelo básico, obteniendo una precisión 15 puntos por encima. No podemos evitar que acabe produciéndose un sobreajuste, pero sí que obtenemos una precisión aceptable en un tiempo de entrenamiento despreciable (cada época tarda unos pocos milisegundos).

La interpretación que hacemos de esta mejora es que el módulo de clasificación que hemos añadido (dos capas totalmente conectadas) aprovecha las características extraídas para aprender rasgos concretos del problema que estamos tratando, y no está tan alejado del mismo como el modelo básico. Sin embargo, el precio que pagamos es que al estar preentrenada en una base de datos muy extensa se produce rápidamente un sobreajuste.



(a) Pérdida en entrenamiento y validación



(b) Precisión en entrenamiento y validación

Figura 10: Estadísticas de entrenamiento del modelo totalmente conectado.

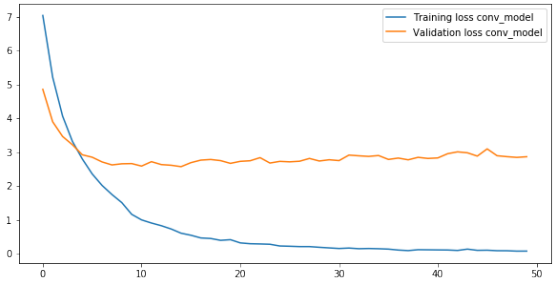
El segundo modelo que probamos es el siguiente, visto tanto en código como en tabla:

```
def conv_model():
    model = Sequential()
    model.add(Conv2D(64, kernel_size = (3, 3),
                     use_bias = False,
                     activation = 'relu',
                     input_shape = (7, 7, 2048)))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(1024, activation = 'relu',
                    use_bias = False))
    model.add(BatchNormalization())
    model.add(Dropout(0.7))
    model.add(Dense(200, activation = 'softmax'))
    return model
```

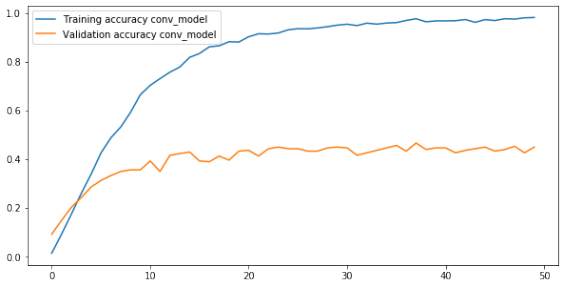
Layer No.	Layer Type	Kernel size	Input|Output dim.	Input|Output channels
1	Conv2D	3	7|5	2048|64
2	Relu	-	5|5	-
3	BatchNorm	-	5|5	-
4	Dropout (0.5)	-	5|5	-
5	Dense	-	1600|1024	-
6	Relu	-	1024|1024	-
7	BatchNorm	-	1024|1024	-
8	Dropout (0.7)	-	1024|1024	-
9	Dense	-	1024|200	-

A la hora de ejecutar este modelo, debemos pasar el parámetro `pooling = None` para construir el modelo de ResNet50, pues queremos mantener el tamaño de $(7, 7, 2048)$ para poder hacer una convolución. Ejecutando ahora este modelo, obtenemos:

```
---------- CONV_MODEL EVALUATION ----------
Test loss: 3.1692161699195056
Test accuracy: 0.38806462246633544
```



(a) Pérdida en entrenamiento y validación



(b) Precisión en entrenamiento y validación

Figura 11: Estadísticas de entrenamiento del modelo convolucional.

Vemos que en este caso no conseguimos mejorar la precisión en la evaluación, y seguimos viendo el sobreajuste en unas pocas épocas. Además, el tiempo por cada época es un poco superior al del modelo totalmente conectado, luego preferimos ese a este convolucional.

Fine-tuning

En este apartado lo que queremos hacer es similar a lo que hicimos en el anterior, pero en vez de extraer características de las imágenes usando Resnet50 y construir un modelo posterior, consideramos todo como un único modelo que entrenaremos completamente unas pocas épocas. Aprovechamos que la red ya está preentrenada y tendrá unos pesos bastante buenos, y al entrenarla entera movemos un poco estos pesos para adaptarnos a nuestro problema concreto.

El modelo que vamos a añadir tras quitar la última capa de ResNet50 es el siguiente modelo con una capa totalmente conectada y un `Dropout` antes de la capa final de activación *softmax* (para prevenir el sobreajuste). Hay que tener en cuenta que se trata de una instancia de `Model` y no de `Sequential`, por lo que la sintaxis cambia ligeramente.

```
def fc_model_2(x):
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.5)(x)
    output = Dense(200, activation='softmax')(x)

    return output
```

Visto en forma de tabla, sería:

Layer No.	Layer Type	Kernel size	Input|Output dim.	Input|Output channels
1	Dense	-	2048|1024	-
2	Relu	-	1024|1024	-
3	Dropout (0.5)	-	1024|1024	-
4	Dense	-	1024|200	-

Para ejecutar este modelo la construcción es un poco diferente al apartado anterior:

```
def resnet_fine_tuning(show = True, save_w = False, load_w = False):
    # Cargamos los datos
    x_train, y_train, x_test, y_test = load_data()

    # Creamos un generador para entrenamiento y otro para test
    datagen_train = ImageDataGenerator(preprocessing_function = preprocess_input,
                                       validation_split = 0.1)
    datagen_test = ImageDataGenerator(preprocessing_function = preprocess_input)

    # Usamos ResNet50 preentrenada en ImageNet sin la última capa
    resnet = ResNet50(weights = 'imagenet',
                      include_top = False,
                      pooling = 'avg')

    # Definimos un nuevo modelo a partir de ResNet50
    output = fc_model_2(resnet.output)
    model = Model(inputs = resnet.input, outputs = output)

    # Compilamos el modelo
    compile(model)

    # Entrenamos el modelo completo
    hist = train_gen(model, datagen_train, 15, x_train, y_train)

    # Evaluamos el modelo
    score = evaluate_gen(model, datagen_test, x_test, y_test)
```

La diferencia es que ahora tomamos la salida de ResNet50 directamente como base, y añadimos al modelo las capas que queremos. Definimos nuestro modelo como una instancia de `Model` que acepta como entrada la misma que ResNet50, y que proporciona como salida la salida completa del modelo con las nuevas capas.

Entrenamos este modelo durante 15 épocas, y al evaluarlo obtenemos lo siguiente:

```
---------- FINE_TUNING EVALUATION ----------
Test loss: 2.397986556532407
Test accuracy: 0.4497197494230135
```

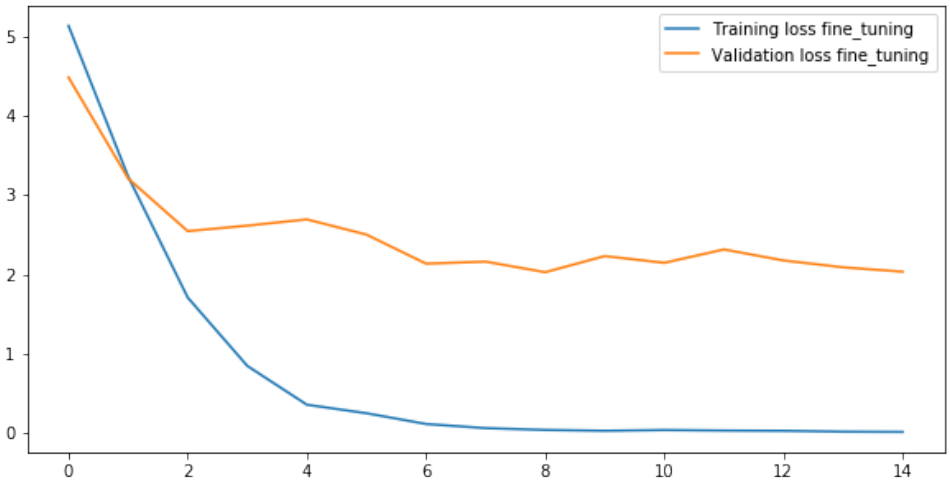


Figura 12: Estadísticas de pérdida en fine-tuning.

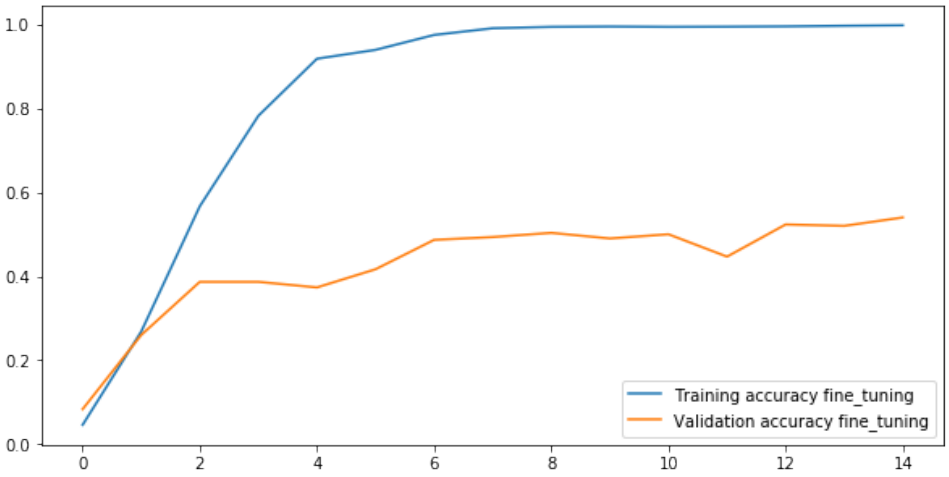


Figura 13: Estadísticas de precisión en fine-tuning.

Observamos que la precisión en la evaluación que obtenemos es similar a la del apartado anterior. También es similar el sobreajuste que se produce, posiblemente por el mismo motivo que se comentó anteriormente: la red está preentrenada y los pesos iniciales son "buenos", luego no hay mucho lugar para el aprendizaje. Sin embargo, en este caso con menos épocas obtenemos un resultado similar.

Como conclusión, hemos visto dos métodos para transferir el aprendizaje de una red preentrenada (*transfer learning*) a otra red que se adapte a un problema concreto. Hemos elegido un modelo final bastante sencillo, formado por un par de capas totalmente conectadas (salvo la variación convolucional). También se han hecho pruebas añadiendo más

capas y más diversas, pero en todos los casos probados el desempeño era peor. Nos ha faltado hacer pruebas quitando más capas al final de ResNet, quizás quitando el último bloque convolucional y entrenando un bloque residual completo definido por nosotros, para adaptarnos mejor al problema que tenemos.

En nuestro caso ambos métodos tienen un desempeño parecido, pero probando en distintos conjuntos de datos podríamos ver que hay veces que uno de los dos es más adecuado para un problema dado.

Referencias

[**1**] Log loss

[**2**] Deep Residual Learning for Image Recognition

[**3**] Caltech-UCSD Birds 200