Práctica-2: Redes neuronales convolucionales

Visión por Computador.

UNIVERSIDAD DE GRANADA

E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática.
Curso 2019-2020.

Daniel Bolaños Martínez
76592621-E
danibolanos@correo.ugr.es
Grupo 2 - Jueves 11:30h

Índice

1. Introducción.

Para esta práctica, utilizando las herramientas proporcionadas por Keras, implementaremos usando redes neuronales convolucionales una arquitectura de red profunda para clasificar imágenes a partir de la extracción de características basada en técnicas de filtrado lineal.

El objetivo de la práctica será estudiar el comportamiento de un clasificador de imágenes basado en redes neuronales modificando su estructura e implementar una red a partir de una existente que sabemos que funciona bien (usándola como extractor de características o haciendo un ajuste fino de la misma).

2. BaseNet en CIFAR100.

En este apartado trabajaremos con el conjunto de datos *CIFAR100* que consta de imágenes de 100 clases, con 600 imágenes por clase divididas en 50.000 imágenes para entrenamiento y 10.000 imágenes para test. Las imágenes en CIFAR100 son imágenes en color de 3 canales de 32x32 píxeles.

Para reducir el conjunto de datos, las funciones que se nos proporcionan en la plantilla se encargan de reducir el número de clases a 25, y por tanto obtendremos un conjunto de entrenamiento de 12.500 imágenes y uno de prueba de 2.500. Ademaś reservaremos un 10 % del conjunto de entrenamiento para validación.

Además de la función para cargar los datos encargada de dividir las imágenes en train y test, tenemos funciones para calcular el accuracy (que calcula el porcentaje de etiquetas bien clasificadas respecto al total) y mostrar las gráficas de la evolución de *loss* y *accuracy* para los conjuntos de entrenamiento y validación.

Para el primer apartado, se nos pide implementar el modelo de red convolucional **BaseNet** y obtener una precisión de referencia que nos permita compararla con las mejoras posteriores. Para definir el modelo en *Keras*, haremos uso de la clase *Sequential()* que nos permite implementar las capas de la red de forma secuencial. **BaseNet** consta de dos módulos convolucionales

(conv-relu-maxpool) y dos capas lineales. Construiremos el modelo **BaseNet** basándonos en la información de la siguiente tabla:

Layer No.	Layer Type	Kernel size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Conv2D	5	32 | 28	3 | 6
2	Relu	-	28 | 28	-
3	MaxPooling2D	2	28 | 14	-
4	Conv2D	5	14 | 10	6 | 16
5	Relu	-	10 | 10	-
6	MaxPooling2D	2	10 | 5	-
7	Linear	-	400 | 50	-
8	Relu	-	50 | 50	-
9	Linear	-	50 | 25	-

Tabla 1: Esquema modelo BaseNet.

La información necesaria para definir el modelo es:

- **Tipo de capa**: Convolucional (**Conv2D**), Agrupación (**MaxPooling2D**), Totalmente Conectadas (**Dense**), Aplanar (**Flatten**), Normalización (**BatchNormalization**).
- **Tamaño del kernel**: tamaño de la máscara utilizada para la convolución o el pooling 2D.
- **Output channels**: canales de salida para las capas de convolución.
- **Output dimension**: dimensión de salida para el resto de capas.

Las capas de activación pueden construirse añadiendo una capa *Activation()* en cualquier punto de la red a la que se le especifica el tipo de activación o añadiendola detrás de cualquier capa pasando el tipo de activación como parámetro a dicha capa. De forma general, optaré por el segundo caso.

El código que construye el modelo **BaseNet** en *Keras* y que he usado para las pruebas en este apartado es el siguiente:

```
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'))
```

En la primera capa se definen las dimensiones de entrada del modelo, en este caso (32,32,3) para definir imágenes RGB de tamaño 32x32 píxeles. La primera capa implementa **Conv2D** que crea 6 filtros de convolución de tamaño (5x5) que se aplicarán sobre la entrada y generarán el tensor de salidas.

Cada capa de convolución implementará a continuación una capa de activación (pasada como parámetro *activation*) que aportará la no linealidad al modelo. En nuestro caso, *relu* viene definida como $relu(x) = max\{0, x\}$. Permite el paso de todos los valores positivos sin cambiarlos y asigna todos los valores negativos a 0.

La capa de **MaxPooling2D** modificará los valores y dimensiones de cada matriz a partir del valor máximo de cada submatriz de tamaño, en este caso, (2x2). Obtendremos una nueva matriz cuyo tamaño será $\frac{1}{4}$ de la anterior. Esta operación, tendrá como finalidad resaltar la información relevante a la salida de la capa de convolución.

En cada capa de **Conv2D**, aumentaremos el valor del tamaño de los canales de salida. Cada capa añadida hace que el tensor de salidas se haga espacialmente mas estrecho y aumente el número de canales haciendo la red más profunda.

La capa **Flatten** se encarga de pasar el último tensor de salida calculado a un vector unidimensional de tamaño 400 que sea tratable por las capas *fully connected*.

Finalmente tenemos una capa **Dense** que multiplicaa el vector salida de la capa anterior por máscaras cuyo resultado será un vector 1D con dimensión la que se especifique (en este caso 50) y con activación *relu*.

Definiremos como última capa una capa **Dense** con tantas neuronas como

clases tenga el problema (en nuestro caso 25) y una activación *softmax* para transformar las salidas de las neuronas en la probabilidad de pertenecer a cada clase. Se define z como el vector de salida de **Dense** y $softmax(z_j)$ como la probabilidad de la imagen de pertenecer a la clase j:

$$softmax(z_j) = \frac{exp(z_j)}{\sum_{k=1}^{N} exp(z_j)}$$

Una vez definido el modelo, lo compilamos especificando la función de optimización, la de objetivo o pérdida y la métrica que usaremos. En este caso, usaremos la función de optimización de descenso de gradiente estocástico (SGD), la función de objetivo de entropía cruzada y para la métrica la tasa de acierto o accuracy.

Utilizaremos SGD para la optimización del modelo ya que nos ofrece las siguientes ventajas:

- Se reduce el número de cálculos en cada iteración. SGD puede procesar más ejemplos dentro del tiempo de cálculo disponible.
- El gradiente estocástico permite suavizar la función objetivo y reduce el riesgo de converger muy rápidamente o estancarse en mínimos locales.
- En conjuntos grandes de datos donde la función objetivo es la suma de errores para cada punto de datos, la muestra produce un valor muy cercano al de la población.

Utilizaré *categorical_crossentropy* como función de pérdida ya que es la más utilizada en la clasificación multiclase. También usaré los mismos parámetros para el optimizador SGD que los usados en la plantilla de ejemplo *mnist.py* proporcionada en DECSAI.

```
opt = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=opt,
              metrics=['acc'])
```

Una vez tenemos el modelo base, y antes de entrenar, guardaremos los pesos aleatorios con los que empieza la red, para poder reestablecerlos después y comparar resultados entre no usar mejoras y sí usarlas. Esto lo haremos con las siguientes líneas:

```
# Funcion para guardar los pesos
weights = model.get_weights()

# Funcion para restablecer los pesos iniciales
model.set_weights(weights)
```

Una vez tenemos compilado el modelo, pasaremos a entrenarlo. Para ello usaremos la función *fit_generator()* que recibirá como parámetro un generador encargado de crear las imágenes en base a unos parámetros prefijados que nos permitirán especificar entre otras cosas: normalización de los datos, división del conjunto o aumento de datos.

En este primer caso, solo crearemos un generador para el conjunto de entrenamiento, puesto que la única modificación sobre los conjuntos de datos que vamos a hacer es una partición del 10 % del conjunto train como validación. La partición la crearemos haciendo uso del parámetro *validation_split*. Para generar las imágenes en los conjuntos de entrenamiento y validación por separado, usaremos la función *flow()* de *ImageDataGenerator* y le indicaremos el *subset* correspondiente, además del conjunto utilizado (en nuestro caso *x_train*), etiquetas (*y_train*) y tamaño de batch (prefijado en 32 en toda la práctica).

```
datagen_train = ImageDataGenerator(validation_split = 0.1)

train = datagen_train.flow(x_train, y_train, batch_size =
    batch_size, subset = 'training')
validation = datagen_train.flow(x_train, y_train, batch_size =
    batch_size, subset = 'validation')
```

Haremos una llamada a la función *fit_generator()* para entrenar el modelo. A la función le pasaremos los conjuntos de entrenamiento y validación generados y el número de épocas durante las que se entrena la red (*epochs*) en nuestro caso fijadas a 20.

También configuramos *steps_per_epoch* que fija el número de batches de imágenes que se usan antes de terminar una época del entrenamiento y pasar a la siguiente (utilizamos el total imágenes de entrenamiento entre el tamaño de cada batch) y *validation_steps* que fija el número de batches de imágenes de validación que se generan al final de cada época (utilizamos el total imágenes de validación entre el tamaño de cada batch).

```
histograma = model.fit_generator(train, steps_per_epoch = len(
    x_train)*0.9/batch_size, epochs = epochs, validation_data =
    validation, validation_steps = len(x_train)*0.1/batch_size)
```

Finalmente haremos uso de la función *mostrarEvolucion* a la que le pasaremos el *histograma* generado para obtener las gráficas de accuracy y pérdida de los conjuntos de entrenamiento y validación.

Podemos usar la red, una vez que ha terminado de entrenar, para predecir la clase de nuevas imágenes. Para ello usaremos la función *predict()* de la clase *Sequential*. Le pasaremos el conjunto de test (*x_test*) para hacer la predicción y seguidamente calcularemos el accuracy con la predicción y las etiquetas del conjunto de prueba (*y_test*).

Este valor no será decisivo para descartar modificaciones, ya que nos basaremos en el accuracy del conjunto de validación y entrenamiento, pero nos será de utilidad para ver como se comporta nuestra red sobre nuevas muestras de imágenes sobre las que no ha entrenado.

```
preds = model.predict(x_test)
score = calcularAccuracy(y_test, preds)
print("Test accuracy = " + str(score))
```

También calcularemos el accuracy para el conjunto de entrenamiento para obtener los resultados del clasificador para la totalidad de datos del conjunto de entrenamiento.

```
preds = model.predict(x_train)
score = calcularAccuracy(y_train, preds)
print("Train accuracy = " + str(score))
```

A continuación se mostrarán los resultados obtenidos tanto de las gráficas como de los valores de *accuracy* y *loss* de cada conjunto (en el caso de validation usaremos los valores obtenidos en la última época).

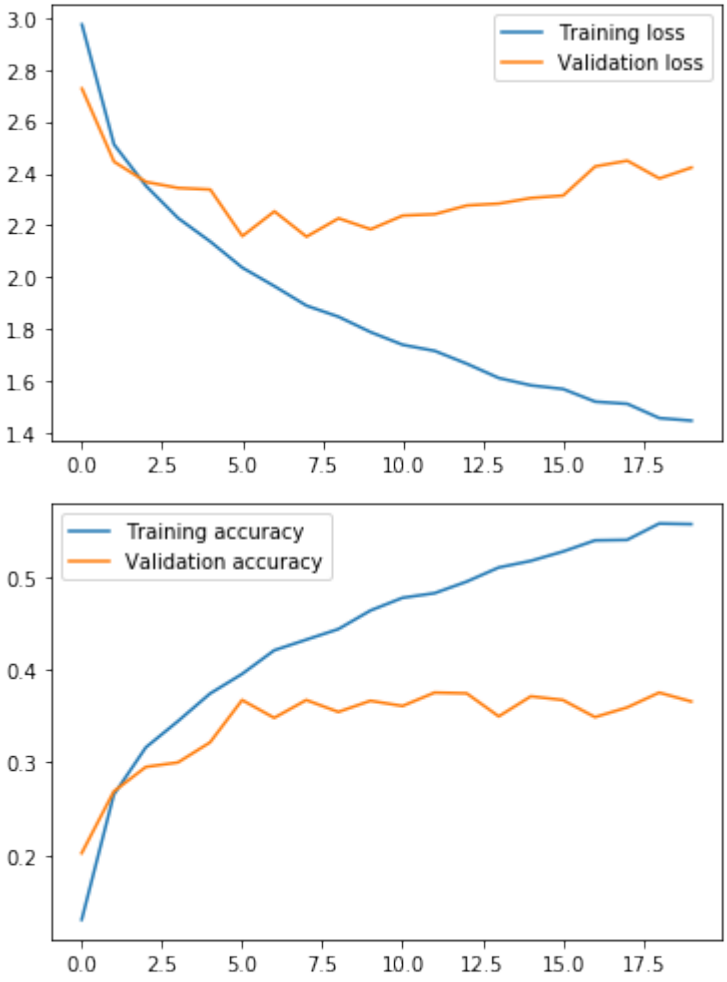

Figura 1: Evolución loss y accuracy en BaseNet.

BaseNet	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
validation_split=0.1	1.4456	0.57824	2.4225	0.3656	0.3724

Tabla 2: Tabla con los valores de loss y accuracy para cada conjunto.

3. Mejora del modelo.

El objetivo de este apartado es crear una red neuronal mejorada modificando las capas y añadiendo nuevas para conseguir una precisión que se acerque al 50 %. Seguiremos las modificaciones propuestas y realizaremos diversas pruebas que plasmaremos junto con las gráficas en una tabla comparativa.

Para evitar una sobrecarga de imágenes en la memoria, solo añadiremos las gráficas de aquellas pruebas que mejoren realmente la modificación o subapartado anterior, aunque plasmaremos todas las modificaciones en las tablas.

3.1. Normalización de datos.

La normalización de los datos de entrada facilita el entrenamiento y lo hace más sólido. Deberemos hacer una normalización de los datos con media 0 y desviación estándar 1. Para ello, usaremos los parámetros de la clase *ImageDataGenerator*, respectivamente: *featurewise_center* y *featurewise_std_normalization* los cuales pondremos a True.

En la documentación de *Keras* se especifica que si ponemos a True alguno de los dos parámetros mencionados anteriormente, tenemos que usar la función *fit()* para ajustar los generadores al conjunto de los datos. En este caso, aunque creamos un generador para normalizar train y otro para test, ajustaremos sobre el conjunto de entrenamiento, ya que no podemos obtener información de *x_test* en el entrenamiento.

Crearemos generadores de imágenes que normalicen los datos y en el caso del generador de train, mantendremos el *validation_split* para realizar la par-

tición. Si un parámetro, en el caso de la normalización, mejora los resultados, no los descartaremos, lo mantendremos en la siguiente mejora.

```
datagen_train_norm = ImageDataGenerator(featurewise_center =
    True, featurewise_std_normalization = True, validation_split
    = 0.1)
datagen_test_norm = ImageDataGenerator(featurewise_center = True
    , featurewise_std_normalization = True)

datagen_train_norm.fit(x_train)
datagen_test_norm.fit(x_train)
```

A partir de aquí el proceso es equivalente, generamos los conjuntos de train y validation, realizamos el entrenamiento de los datos con *fit_generator* y obtendremos el histograma junto con los resultados de las métricas. En este caso, como estamos usando generadores para normalizar los datos, usaremos *predict_generator* para obtener las predicciones que usaremos para calcular el accuracy.

```
preds = model.predict_generator(datagen_test_norm.flow(x_test,
    batch_size = 1, shuffle = False), steps = len(x_test))
score = calcularAccuracy(y_test, preds)
```

Debemos especificar para las predicciones de train y test *batch_size=1* y *shuffle=False* para que las predicciones de las nuevas imágenes estén en el mismo orden que las imágenes de los conjuntos originales, y se puedan comparar con las etiquetas reales.

A continuación en las gráficas obtenidas, podemos observar como la normalización de los datos ha experimentado una leve mejora de los resultados en los tres conjuntos.

BaseNet	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
validation_split=0.1	1.4456	0.57824	2.4225	0.3656	0.3724
validation_split=0.1 + normalización	1.2800	0.6296	2.7775	0.3744	0.3776

Tabla 3: Tabla con los valores de loss y accuracy para cada conjunto.

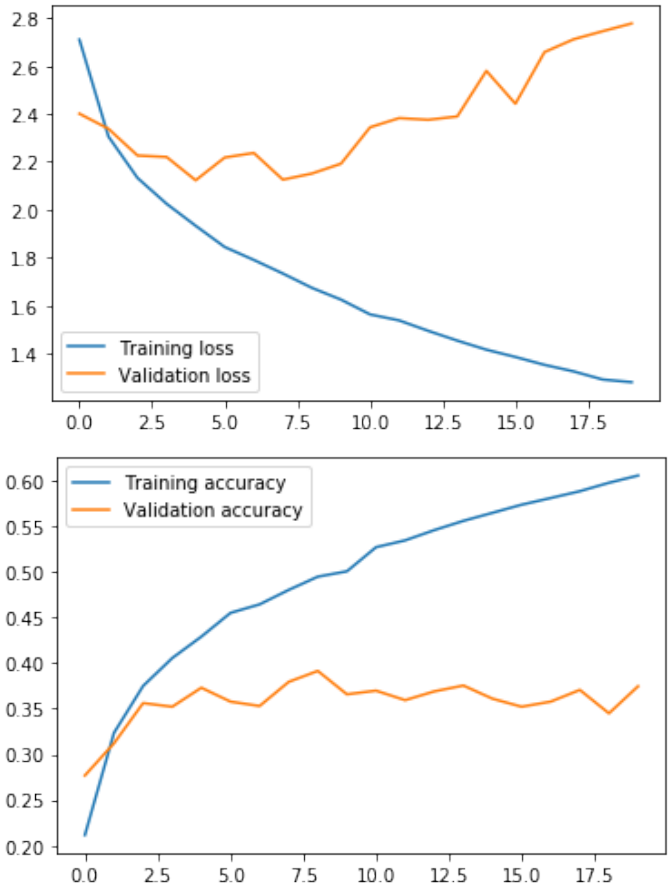

Figura 2: Evolución de loss y accuracy con normalización en BaseNet.

3.2. Aumento de Datos.

Se propone usar algunos de los parámetros de aumento de datos de la clase *ImageDataGenerator*, como *zoom_range* u *horizontal_flip*.

Realizaremos las modificaciones sobre el conjunto de entrenamiento. No aplicaremos ninguna modificación de este tipo sobre el generador de test. Además mantendremos la normalización en ambos conjuntos. Usaremos varios parámetros y veremos cual tiene mejor funcionamiento.

El parámetro horizontal/vertical flip, hace un giro aleatorio horizontal o vertical (dependiendo del caso) a la imagen. En nuestro caso, podemos usarlo ya que un giro de este tipo no cambia la distribución de las imágenes (por el tipo de imágenes de CIFAR100). Por otro lado, *zoom_range* hace un zoom aleatorio en el rango especificado:

$$[lower, upper] = [1 - zoom\_range, 1 + zoom\_range]$$

Probaremos a usar diferentes valores de *zoom_range* con y sin horizontal flip. Valoraremos los resultados sobre los conjuntos de train y validation y nos quedaremos con el que mejor funcione de todos ellos.

El aumento de datos de forma general, debería reducir el overfitting y aumentar el accuracy del conjunto de validación, ya que si añadimos nuevas muestras al conjunto de entrenamiento, reduciremos la tasa de aciertos en el conjunto de *train*, pero podemos generar nuevas muestras que se asemejen a las existentes en *validation*.

Cabe recordar que puede que la mejor modificación de aumento de datos para este modelo no sea la mejor para el modelo que construya en el apartado siguiente, incluso puede que haya algún caso en el que un modelo funcione mejor sin aumentar los datos.

En la tabla 4 podemos observar como conforme aumentamos el valor de *zoom_range* obtenemos peores resultados en ambos conjuntos. Utilizando *horizontal_flip* conseguimos mejores resultados en el validation y peores en el train, esto es bueno ya que conseguimos reducir el overfitting y aumentar la tasa de clasificación del conjunto de validación.

BaseNet	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
horizontal_flip=False + zoom_range=0.0	1.3377	0.59456	2.7121	0.3832	0.3904
horizontal_flip=True + zoom_range=0.0	1.5314	0.54856	2.0523	0.4184	0.4396
horizontal_flip=False + zoom_range=0.2	1.6120	0.53256	2.4019	0.3600	0.4072
horizontal_flip=True + zoom_range=0.2	1.7226	0.47576	2.1051	0.3912	0.4488
horizontal_flip=False + zoom_range=0.5	1.9420	0.44064	2.2501	0.3512	0.44
horizontal_flip=True + zoom_range=0.5	2.0033	0.40728	2.2559	0.3552	0.4128

Tabla 4: Tabla comparativa modificaciones aumento de datos.

Compararemos las versiones con zoom_range=0.0 y 0.2, fijando en ambos horizontal_flip=True. Aunque la versión con 0.2, obtiene mejores resultados en el conjunto de test, dijimos que esto no sería relevante para evaluar las modificaciones. En el caso de los conjuntos de entrenamiento y validación, la diferencia entre los accuracys de train y validation es menor en el caso zoom_range=0.2 y por tanto tendrá una gráfica más ajustada.

Sin embargo, nos decantaremos por el caso zoom_range=0.0, debido a que en el otro caso, un accuracy de 0.47 en el conjunto de entrenamiento podría impedir que el conjunto pudiese aprender más de lo que está capacitado.

Añadiremos los resultados a la tabla general y mostraremos las gráficas obtenidas. (*)También implementa *validation_split=0.1* y normalización.

BaseNet	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
validation_split=0.1	1.4456	0.57824	2.4225	0.3656	0.3724
validation_split=0.1 + normalización	1.2800	0.6296	2.7775	0.3744	0.3776
zoom_range=0.0 + horizontal_flip=True (*)	1.5314	0.54856	2.0523	0.4184	0.4396

Tabla 5: Tabla general con las métricas para cada conjunto.

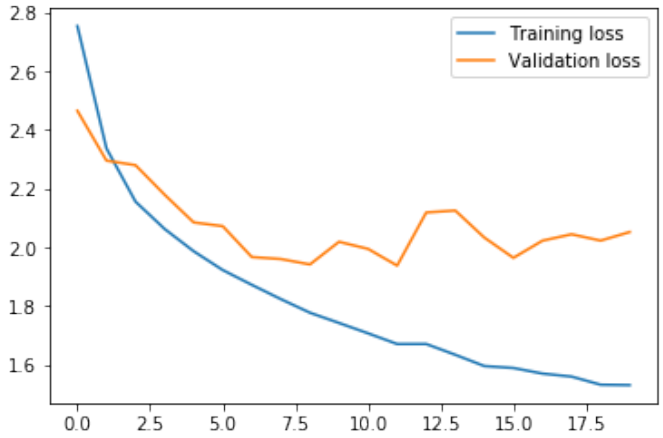

Figura 3: Evolución loss en BaseNet con aumento de datos.

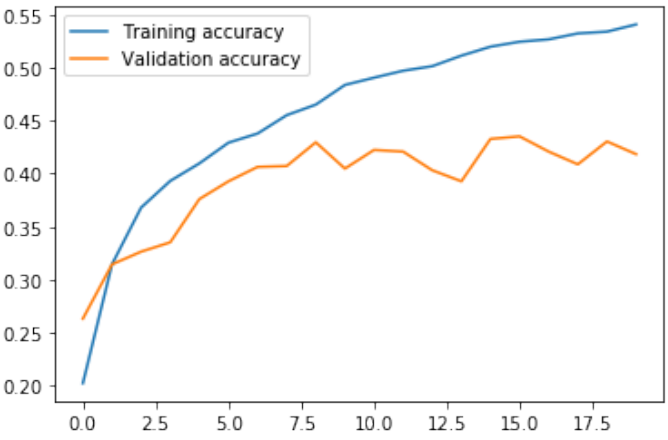

Figura 4: Evolución accuracy en BaseNet con aumento de datos.

3.3. Red más profunda.

Se propone experimentar agregando más capas convolucionales y totalmente conectadas. Evitaremos añadir una capa de **MaxPooling2D** después de cada capa convolucional para no perder demasiada información en cada módulo convolucional y probaremos a añadir capas de regularización **Dropout** que añaden una probabilidad de desactivación de las neuronas durante el entrenamiento, dificultando el entrenamiento y mejorando la validación, lo que se reflejará en una reducción del overfitting y una mejora en la tasa de clasificación en la validación.

Probaremos diferentes combinaciones de capas modificando algunos parámetros y mostraremos los resultados de la versión que mejores resultados obtenga. En este apartado, no trabajaremos con la capa **BatchNormalization**.

Para este apartado, iremos modificando la red a la que añadiremos las capas de forma secuencial y haremos la compilación del modelo tal y como lo hicimos con **BaseNet**. Usaremos normalización de los datos pero no usaremos aumento de los mismos por el momento.

Para la primera versión V1, añadiremos dos capas convolucionales más, aumentaremos de forma creciente el valor de la dimensión de salida y añadiremos una capa de **MaxPooling2D** cada dos convolucionales. Mantendremos el mismo número de capas **Dense** que **BaseNet** y no modificaremos los tamaños de las máscaras convolucionales.

```
# MiRed V1
my_model = Sequential()
my_model.add(Conv2D(16, kernel_size=(5,5), activation = 'relu',
    input_shape=input_shape))
my_model.add(Conv2D(26, kernel_size=(5,5), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Conv2D(36, kernel_size=(5,5), activation='relu'))
my_model.add(Conv2D(46, kernel_size=(5,5), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Flatten())
my_model.add(Dense(50, activation='relu'))
my_model.add(Dense(num_classes, activation='softmax'))
```

Podemos observar en la tabla, cómo la nueva versión, se acerca mucho a los resultados de **BaseNet**, pero obtiene un poco peor resultado en el conjunto de validación y a la larga sobreentrenará los datos, ya que aumenta el accuracy de entrenamiento.

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
BaseNet best version	1.5314	0.54856	2.0523	0.4184	0.4396
MiRed V1	1.3058	0.61288	2.5128	0.3744	0.412

Para la siguiente modificación (V2), aumentaremos el valor de las salidas de las capas convolucionales y el de la capa **Dense** y veremos si este cambio, influirá en los resultados positivamente.

```
# MiRed V2
my_model = Sequential()
my_model.add(Conv2D(32, kernel_size=(5,5), activation = 'relu',
    input_shape=input_shape))
my_model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Conv2D(128, kernel_size=(5,5), activation='relu'))
my_model.add(Conv2D(256, kernel_size=(5,5), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Flatten())
my_model.add(Dense(512, activation='relu'))
my_model.add(Dense(num_classes, activation='softmax'))
```

En este caso, en la tabla se observa como mejora el accuracy pero sobreentrena mucho los datos, por lo que tendremos que ir remediando esto en modificaciones posteriores, implementando capas de regularización.

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
BaseNet best version	1.5314	0.54856	2.0523	0.4184	0.4396
MiRed V1	1.3058	0.61288	2.5128	0.3744	0.412
MiRed V2	0.5674	0.80376	3.3151	0.4208	0.4248

En la modificación V3, disminuiremos el tamaño de la máscara de convolución a (3,3) y veremos como afecta a los resultados. En principio, esta modificación debería aumentar el accuracy de validation, ya que haremos modificaciones más variadas de los valores de entrada en el filtro de convolución y así obtendremos mejores filtros de salida.

```
# MiRed V3
my_model = Sequential()
my_model.add(Conv2D(32, kernel_size=(3,3), activation = 'relu',
    input_shape=input_shape))
my_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Conv2D(128, kernel_size=(3,3), activation='relu'))
my_model.add(Conv2D(256, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Flatten())
my_model.add(Dense(512, activation='relu'))
my_model.add(Dense(num_classes, activation='softmax'))
```

Como podemos ver en la tabla y ya habíamos adelantado, al reducir el tamaño de la máscara hemos mejorado el accuracy del conjunto validation, aún así, tenemos un accuracy en train demasiado alto que debemos de reducir e igualar cuanto más posible al de validation para evitar el overfitting.

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
BaseNet best version	1.5314	0.54856	2.0523	0.4184	0.4396
MiRed V1	1.3058	0.61288	2.5128	0.3744	0.412
MiRed V2	0.5674	0.80376	3.3151	0.4208	0.4248
MiRed V3	0.0600	0.9376	3.8861	0.4992	0.5268

Para reducir el sobreentrenamiento, en la modificación V4 reduciremos el tamaño de la salida de los filtros convolucionales y añadiremos algunas capas de **Dropout** con probabilidad de desactivación de 0.5.

```
# MiRed V4
my_model = Sequential()
```

```
my_model.add(Conv2D(32, kernel_size=(3,3), activation = 'relu',
    input_shape=input_shape))
my_model.add(Dropout(0.5))
my_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
my_model.add(Dropout(0.5))
my_model.add(Conv2D(128, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Flatten())
my_model.add(Dense(512, activation='relu'))
my_model.add(Dense(num_classes, activation='softmax'))
```

En esta versión hemos conseguido reducir el sobreentrenamiento pero hemos perdido un poco de eficiencia en el conjunto de validación.

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
BaseNet best version	1.5314	0.54856	2.0523	0.4184	0.4396
MiRed V1	1.3058	0.61288	2.5128	0.3744	0.412
MiRed V2	0.5674	0.80376	3.3151	0.4208	0.4248
MiRed V3	0.0600	0.9376	3.8861	0.4992	0.5268
MiRed V4	0.4110	0.85944	2.3266	0.4416	0.472

En la versión V5, optaré por añadir una capa **Dense** adicional y capas intermedias **Dropout** entre las totalmente conectadas para intentar reducir aún más el overfitting. Ademas disminuiré la probabilidad de las capas **Dropout** de los módulos convolucionales de 0.5 a 0.25 para evitar peores resultados al añadir más capas de regularización.

```
# MiRed V5
my_model = Sequential()
my_model.add(Conv2D(32, kernel_size=(3,3), activation = 'relu',
    input_shape=input_shape))
my_model.add(Dropout(0.25))
my_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
my_model.add(Dropout(0.25))
```

```
my_model.add(Conv2D(128, kernel_size=(3,3), activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2,2)))
my_model.add(Flatten())
my_model.add(Dense(512, activation='relu'))
my_model.add(Dropout(0.5))
my_model.add(Dense(256, activation='relu'))
my_model.add(Dropout(0.5))
my_model.add(Dense(num_classes, activation='softmax'))
```

Esta última versión obtiene los mejores resultados en accuracy de test y validation y ha conseguido igualar lo más posible al accuracy de train evitando así el sobreentrenamiento.

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
BaseNet best version	1.5314	0.54856	2.0523	0.4184	0.4396
MiRed V1	1.3058	0.61288	2.5128	0.3744	0.412
MiRed V2	0.5674	0.80376	3.3151	0.4208	0.4248
MiRed V3	0.0600	0.9376	3.8861	0.4992	0.5268
MiRed V4	0.4110	0.85944	2.3266	0.4416	0.472
MiRed V5	1.2051	0.74104	1.5410	0.5392	0.5444

Para finalizar el apartado, realizaré un esquema al estilo del proporcionado para **BaseNet** del modelo que mejor ha funcionado **MiRed V5**.

Hemos construido un modelo de 19 capas con 2 modelos convolucionales (conv-relu-drop-maxpool) y dos capas totalmente conectadas. En apartados posteriores, probaremos otras mejoras tales como aumentar el conjunto de datos aunque como ya dijimos, dependerá del modelo y la prueba ya que *Keras* trabaja con aleatoriedad que nos impide conseguir siempre los mismos resultados aunque fijásemos una semilla.

A continuación se mostrará la tabla del modelo **MiRed V5**.

Layer No.	Layer Type	Kernel size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)	Prob. Dropout
1	Conv2D	3	32 | 30	3 | 32	-
2	Relu	-	30 | 30	-	-
3	Dropout	-	30 | 30	-	0.25
4	Conv2D	3	30 | 28	32 | 64	-
5	Relu	-	28 | 28	-	-
6	MaxPooling2D	2	28 | 14	-	-
7	Conv2D	3	14 | 12	64 | 64	-
8	Relu	-	12 | 12	-	-
9	Dropout	-	12 | 12	-	0.25
10	Conv2D	3	12 | 10	64 | 128	-
11	Relu	-	10 | 10	-	-
12	MaxPooling2D	2	10 | 5	-	-
13	Linear	-	3200 | 512	-	-
14	Relu	-	512 | 512	-	-
15	Dropout	-	512 | 512	-	0.5
16	Linear	-	512 | 256	-	-
17	Relu	-	256 | 256	-	-
18	Dropout	-	256 | 256	-	0.5
19	Linear	-	256 | 25	-	-

Tabla 6: Esquema modelo MiRed V5.

A continuación se muestran las gráficas de evolución de las métricas obtenidas por el modelo **MiRed V5**.

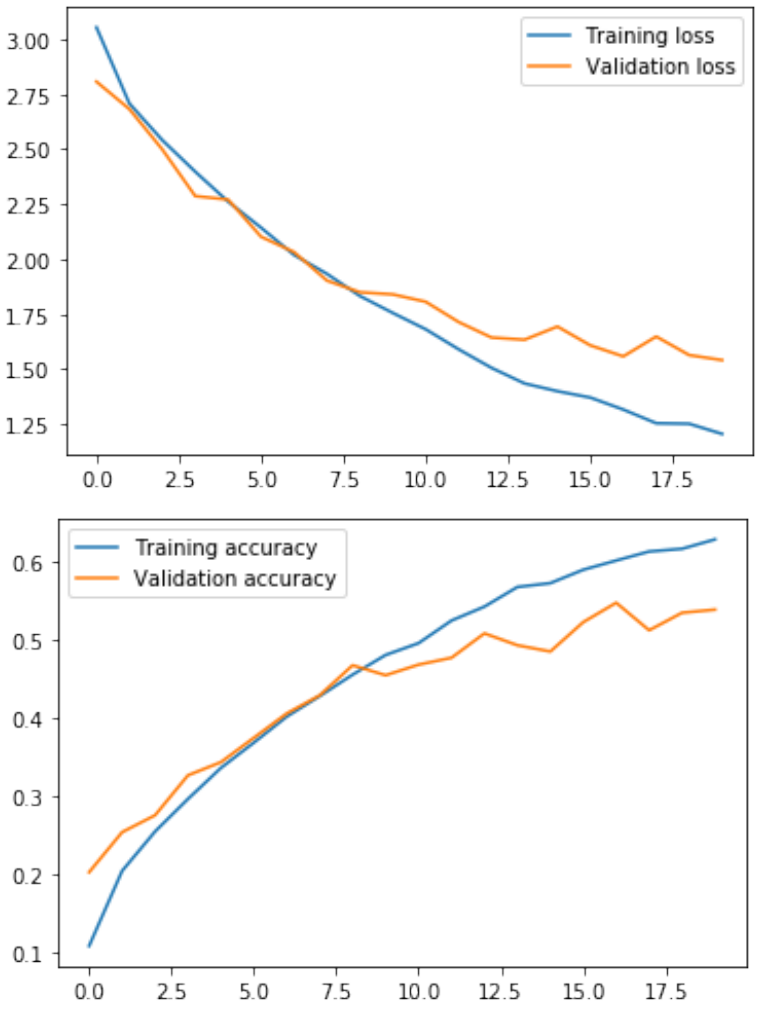

Figura 5: Evolución loss y accuracy en MiRed V5.

3.4. Capas de Normalización.

Las capas de normalización ayudan a reducir el sobreajuste y mejorar el entrenamiento del modelo por lo que añadiremos al modelo anterior una capa **BatchNormalization** detrás de cada capa de convolución.

```
# MiRed + BatchNormalization V1
my_model2 = Sequential()
my_model2.add(Conv2D(32, kernel_size=(3,3), input_shape=
    input_shape))
my_model2.add(Activation('relu'))
my_model2.add(BatchNormalization())
my_model2.add(Dropout(0.25))
my_model2.add(Conv2D(64, kernel_size=(3,3)))
my_model2.add(Activation('relu'))
my_model2.add(BatchNormalization())
my_model2.add(MaxPooling2D(pool_size=(2,2)))
my_model2.add(Conv2D(64, kernel_size=(3,3)))
my_model2.add(Activation('relu'))
my_model2.add(BatchNormalization())
my_model2.add(Dropout(0.25))
my_model2.add(Conv2D(128, kernel_size=(3,3)))
my_model2.add(Activation('relu'))
my_model2.add(BatchNormalization())
my_model2.add(MaxPooling2D(pool_size=(2,2)))
my_model2.add(Flatten())
my_model2.add(Dense(512))
my_model2.add(Activation('relu'))
my_model2.add(Dropout(0.5))
my_model2.add(Dense(256))
my_model2.add(Activation('relu'))
my_model2.add(Dropout(0.5))
my_model2.add(Dense(num_classes, activation='softmax'))
```

Además, haremos dos modificaciones más de la red, una añadiendo capas después de las capas de activación relu y otra añadiendo las capas de normalización antes de las de activación, por lo que tendremos que rediseñar el modelo y añadir las capas de activación de forma manual (*Activation()*), no como parámetro de las convolucionales.

Finalmente, mostraremos una tabla comparativa con las tres modificaciones realizadas, el nuevo esquema del modelo y las gráficas obtenidas para el

mejor caso.

Para evitar saturar la memoria de código, las versiones V2 y V3 de MiRed + BatchNormalization vienen construidas de la siguiente manera:

- **MiRed + BatchNormalization V2**: añade una capa BatchNormalization después de cada capa de activación relu (incluyendo las de las capas **Dense**).
- **MiRed + BatchNormalization V3**: añade una capa BatchNormalization antes de cada capa de activación relu. (incluyendo las de las capas **Dense**).

REDES NEURONALES	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
MiRed+Batch V1	1.2199	0.73984	1.5360	0.5592	0.5716
MiRed+Batch V2	0.9632	0.81912	1.3856	0.5848	0.6056
MiRed+Batch V3	0.9184	0.7912	1.4110	0.6112	0.6124

Tabla 7: Tabla con las métricas por cada tipo de BatchNormalization.

La versión que ha obtenido mejores resultados ha sido la V3, donde añadimos una capa de **BatchNormalization** antes de cada capa de activación relu.

Las capas de activación relu llevan todos los valores negativos a 0, lo que significa una pérdida de información con respecto a los datos obtenidos por las capas anteriores. Si aplicamos una capa de normalización antes de la de activación relu, normalizamos con media 0 y desviación estándar 1, evitando la pérdida de gran parte de la información y mejorarando los resultados.

A continuación se mostrará el esquema final de la red que ha obtenido mejores resultados globales: **MiRed+Batch V3**.

Layer No.	Layer Type	Kernel size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)	Prob. Dropout
1	Conv2D	3	32 | 30	3 | 32	-
2	BatchNormalization	-	30 | 30	-	-
3	Relu	-	30 | 30	-	-
4	Dropout	-	30 | 30	-	0.25
5	Conv2D	3	30 | 28	32 | 64	-
6	BatchNormalization	-	28 | 28	-	-
7	Relu	-	28 | 28	-	-
8	MaxPooling2D	2	28 | 14	-	-
9	Conv2D	3	14 | 12	64 | 64	-
10	BatchNormalization	-	12 | 12	-	-
11	Relu	-	12 | 12	-	-
12	Dropout	-	12 | 12	-	0.25
13	Conv2D	3	12 | 10	64 | 128	-
14	BatchNormalization	-	10 | 10	-	-
15	Relu	-	10 | 10	-	-
16	MaxPooling2D	2	10 | 5	-	-
17	Linear	-	3200 | 512	-	-
18	BatchNormalization	-	512 | 512	-	-
19	Relu	-	512 | 512	-	-
20	Dropout	-	512 | 512	-	0.5
21	Linear	-	512 | 256	-	-
22	BatchNormalization	-	256 | 256	-	-
23	Relu	-	256 | 256	-	-
24	Dropout	-	256 | 256	-	0.5
25	Linear	-	256 | 25	-	-

Tabla 8: Esquema modelo MiRed + BatchNormalization V3.

Obtenemos una red mucho más profunda de 25 capas que funciona mejor que **BaseNet** para el conjunto de imágenes CIFAR100.

También mostraremos las gráficas obtenidas que muestran la evolución de los valores de loss y accuracy para el modelo creado.

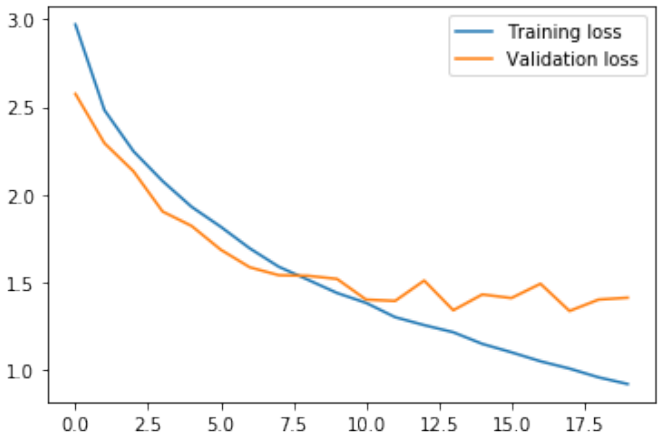

Figura 6: Evolución loss en MiRed+BatchNormalization V3.

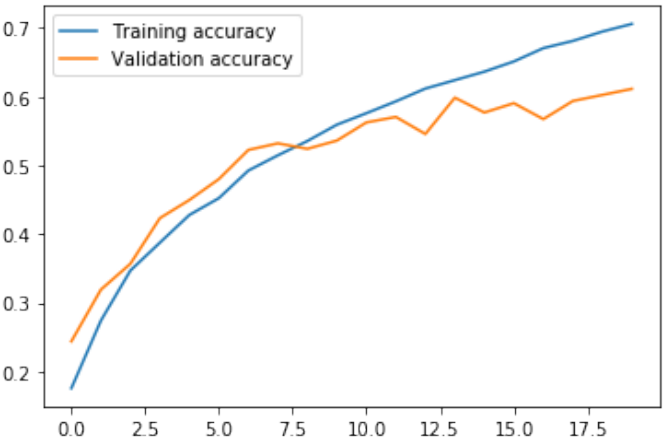

Figura 7: Evolución accuracy en MiRed+BatchNormalization V3.

3.5. Early Stopping.

En este apartado, se nos pide establecer un valor de épocas a partir del cual el valor del accuracy para validation se estabiliza y el de train comienza a aumentar considerablemente.

Una buena forma de realizar este estudio, es fijar un número de épocas alto, digamos 40 (el doble de las que estábamos usando) y observar el momento en el que *val_acc* deje de aumentar. También podemos establecer el punto de parada fijándonos en la gráfica de pérdida y parar cuando la gráfica de loss validation empiece a crecer.

Otra forma, es utilizar la función de *keras.callbacks* **EarlyStopping** que puede monitorear un valor, en nuestro caso, val_acc y hacer que pare de entrenar cuando deje de mejorar durante un periodo de N épocas. A este periodo se le denomina *patiente* y en mi caso lo prefijaré a 4.

Para llamar a la función **EarlyStopping** lo tenemos que hacer en la función *fit_generator()* de la forma siguiente:

```
epochs = 40
histograma = my_model2.fit_generator(train_final,
    steps_per_epoch = len(x_train)*0.9/batch_size, epochs =
    epochs, validation_data = validation_final, validation_steps
    = len(x_train)*0.1/batch_size, callbacks = [EarlyStopping(
    monitor = 'val_acc', patience = 4, restore_best_weights =
    True)])
```

El parámetro *restore_best_weights=True* hace que se restauren los pesos del modelo desde la época con mejor valor de *val_acc*.

Fijaré el número de épocas a 40 y dejaremos que la función deje de entrenar cuando se cumplan las condiciones especificadas. En la primera prueba, se puede observar que el accuracy del conjunto de entrenamiento se dispara. Probaré a aumentar los datos en la nueva red utilizando varias versiones dando preferencia a los parámetros que mejores resultados obtuvieron anteriormente.

(*)También implementa *validation_split=0.1* y normalización.

	Train		**Validation**		**Test**	**Epochs**
Modificación	loss	accuracy	loss	accuracy	accuracy	
validation_split=0.1 + normalización	0.5253	0.91736	1.4150	0.6200	0.6424	31
zoom_range=0.0 + horizontal_flip=True (*)	0.9216	0.7656	1.2949	0.6168	0.644	27
zoom_range=0.2 + horizontal_flip=True (*)	1.0373	0.7376	1.2883	0.6016	0.672	34

Tabla 9: Early Stopping en MiRed+BatchNormalization V3

Podemos observar como el aumento de datos, reduce considerablemente el sobreentrenamiento y tiene unos valores similares a los obtenidos cuando no usamos aumento, consiguiendo además resultados similares en menor número de épocas por lo que es bastante buena idea usar aumento de datos en estas condiciones.

Nos quedaremos con la tercera versión, que en este caso es diferente a la que nos ofrecía mejores resultados en **BaseNet**. He decidido quedarme con esta versión puesto que presenta un mejor equilibrio entre los accuracy de train y validation en la gráfica y además parece que es buena en media en la clasificación de nuevos datos.

A continuación mostraremos las gráficas de evolución de las métricas de esta versión.

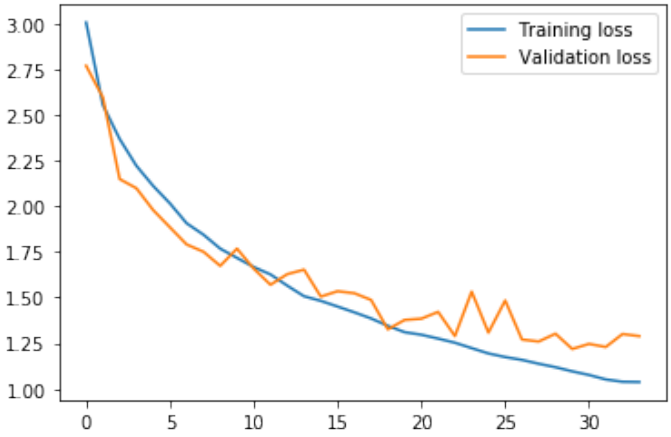

Figura 8: Evolución loss en MiRed+Norm V3 + Aumento de datos.

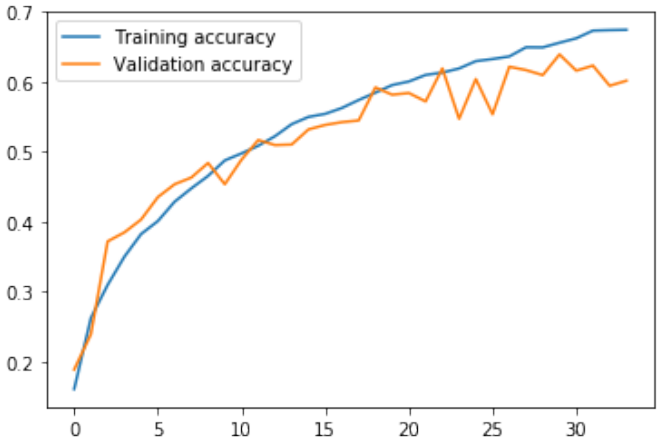

Figura 9: Evolución accuracy en MiRed+Norm V3 + Aumento de datos.

4. Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD.

El conjunto de datos Caltech-UCSD se compone de 6.033 imágenes de 200 especies de pájaros. Cuenta con 200 clases con 3.000 imágenes en el conjunto de entrenamiento y 3.033 en el de prueba. De nuevo, se dejará un 10 % del conjunto de entrenamiento para validación.

Se nos pide utilizar una red que ya viene creada en *Keras* como es **ResNet50** como extractor de características preentrenando previamente en *ImageNet* y realizar un fine-tunning de la red, añadiendo algunas capas más a la misma. Al igual que en los apartados anteriores, se comentará el código y se mostrarán tablas y gráficas con los resultados para ambas versiones.

Las pruebas en este apartado se han realizado en Google Colab. Para agilizar el proceso de descarga de las imágenes, se ha exportado en un fichero zip a Google Drive y se realiza la descarga del fichero directamente desde el repositorio.

```
from google.colab import drive
drive.mount('/content/drive')
! unzip /content/drive/My\ Drive/VC/images.zip -d /content/
```

Utilizaremos las funciones para cargar datos, calcular accuracy y mostrar gráficas de la evolución de las métricas aportadas en DECSAI en la plantilla proporcionada para este apartado.

Para utilizar **ResNet50** como extractor de características, primero necesitamos crear los generadores de las imágenes de train y test con los parámetros necesarios para el preprocesado de la red. Esto lo hacemos usando la función *preprocess_input()* que se agrega como parámetro al generador.

```
datagen_train = ImageDataGenerator(preprocessing_function=
    preprocess_input)
datagen_test = ImageDataGenerator(preprocessing_function=
    preprocess_input)
```

Ahora debemos definir nuestro modelo **ResNet** preentrenado en *ImageNet* y sin la última capa, ya que no queremos que nos devuelva un vector con las probabilidades de la imagen, queremos el vector de características.

```
resnet50 = ResNet50(include_top=False, weights='imagenet',
    pooling='avg')
```

Quitando la última capa con *include_top=False* hemos conseguido que nuestro modelo tenga una última capa con 2048 neuronas. Por tanto, podemos considerar que estamos transformando cada imagen en un vector de 2048 características. Para ello usaremos la función *predict_generator* sobre los generadores de train y test.

```
train = datagen_train.flow(x_train, y_train, batch_size = 1,
    shuffle = False)
test = datagen_test.flow(x_test, batch_size = 1, shuffle = False
    )
train_features = resnet50.predict_generator(train)
test_features = resnet50.predict_generator(test)
```

Es importante especificar el parámetro *shuffle=False* para que a la hora de calcular las predicciones se mantenga el orden original de las imágenes de los conjuntos iniciales.

Las características extraídas serán la entrada de un modelo donde la última capa contará con una activación *softmax* que clasificará las clases de UCSD con 200 clases. El modelo lo compilaremos y entrenaremos de la misma forma que hemos hecho hasta ahora. Fijaremos el número de épocas a 20.

He creado diferentes modelos para comparar cómo mejoran o empeoran los resultados utilizando **ResNet** como extractor de características. Las versiones implementan las siguientes modificaciones:

- **V1**: Modelo de dos capas **Dense** tal y como se aconseja en el enunciado del problema.

  ```
  model = Sequential()
  model.add(Dense(512, activation='relu', input_shape=(2048,))
  ```

```
model.add(Dense(200, activation='softmax'))
```

- **V2**: Añadimos entre las capas *fully connected* una capa de regularización con probabilidad 0.75.

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(2048,))
model.add(Dropout(0.75))
model.add(Dense(200, activation='softmax'))
```

- **V3**: Añadimos una capa **Dense** extra con activación relu con tamaño de salida 256 y se añaden dos capas de regularización con probablidad 0.5 (no mantenemos 0.75 para evitar que el accuracy de train deje de mejorar).

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(2048,))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu')
model.add(Dropout(0.5))
model.add(Dense(200, activation='softmax'))
```

Los resultados de las tres ejecuciones se mostrarán en una tabla donde se discutirá qué versión obtiene mejores resultados y será de la cual mostremos sus gráficas. Se prevee que la que mejores resultados obtendrá será la **V3** ya que añade dos capas de regularización que reducirán el overfitting, el cual es demasiado alto cuando usamos **ResNet** como extractor de características y aumentará la tasa de acierto de validation al contar con dos capas **Dense** con activación relu.

Los resultados obtenidos tardan muy poco en ejecutarse puesto que ya contamos con la red preentrenada y con las características extraidas de la misma. En la tabla 10 observamos en general un valor de accuracy bajo y podemos ver como el modelo sobreentrena los datos en gran medida.

ResNet como extractor	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
Modelo V1	0.0325	0.946	2.1900	0.4800	0.4247
Modelo V2	3.0521	0.7163	2.6553	0.3300	0.3168
Modelo V3	1.7077	0.844	2.3460	0.4133	0.3653

Tabla 10: Resultados utilizando ResNet50 como extractor de características

Nos quedaremos con la versión V3 que es la que de media mejores resultados obtiene y viendo que la gráfica de pérdida tiende a decrecer y la de accuracy a aumentar, usaremos **EarlyStopping** para estudiar cuanto mejor puede ser este modelo aumentando el número de épocas. Usaremos *val_acc* como variable a monitorizar y fijaremos la variable *patiente* a 4. Fijaremos las épocas a 40 esta vez y veremos hasta qué época mejora.

```
epochs = 40
histograma = model.fit(train_features, y_train, epochs=epochs,
    validation_split=0.1, callbacks = [EarlyStopping(monitor = '
    val_acc', patience = 4, restore_best_weights = True)])
```

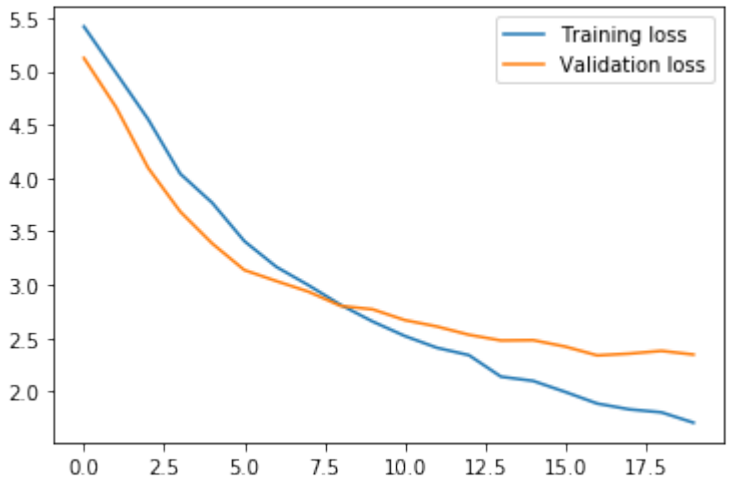

Figura 10: Evolución loss con ResNet como extractor en V3 (epochs=20).

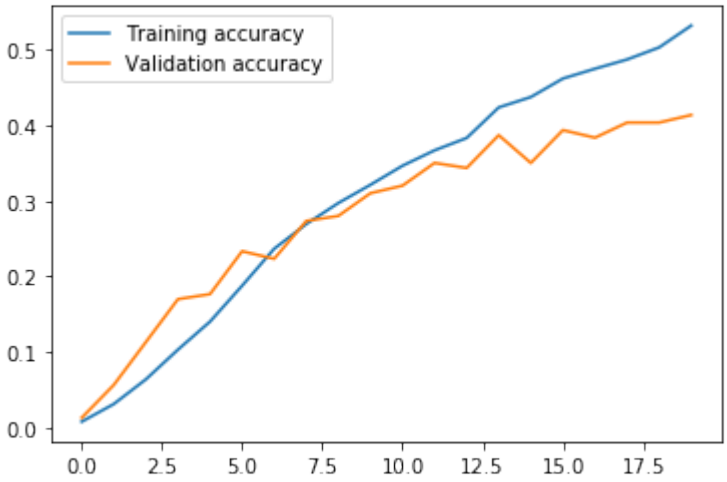

Figura 11: Evolución acc. con ResNet como extractor en V3 (epochs=20).

ResNet como extractor	**Train**		**Validation**		**Test**	**Epochs**
Modificación	loss	accuracy	loss	accuracy	accuracy	
Modelo V1	0.0325	0.946	2.1900	0.4800	0.4247	20
Modelo V2	3.0521	0.7163	2.6553	0.3300	0.3168	20
Modelo V3	1.7077	0.844	2.3460	0.4133	0.3653	20
Modelo V1 + EarlyStopping	0.0340	0.94333	2.0657	0.4867	0.4135	14
Modelo V3 + EarlyStopping	1.2944	0.8817	2.3807	0.4200	0.3769	28

Tabla 11: Tabla comparativa modelos usando ResNet como extractor

He agregado además una versión de V1 con **EarlyStopping** ya que era la que mejores resultados obtenía en el validation accuracy de las tres. Pero podemos observar que aunque obtenga buen resultado en el validation accuracy, el overfitting sigue siendo muy alto y el número de épocas muy pequeño por lo que podemos concluir en que la convergencia de este modelo es muy rápida.

A continuación mostraremos las gráficas de la versión V3 con **EarlyStopping**. Podemos ver que ambas se comienza a estabilizar por lo que no esperamos que obtengan mejores resultados a lo largo del tiempo.

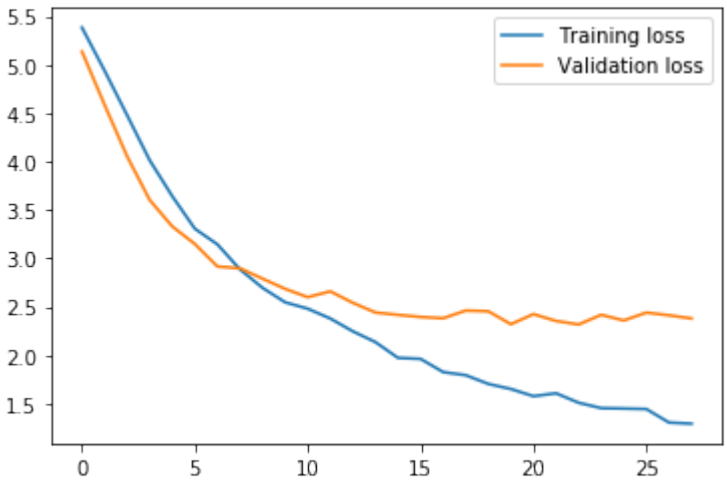

Figura 12: Evolución loss con ResNet como extractor en V3 (epochs=28).

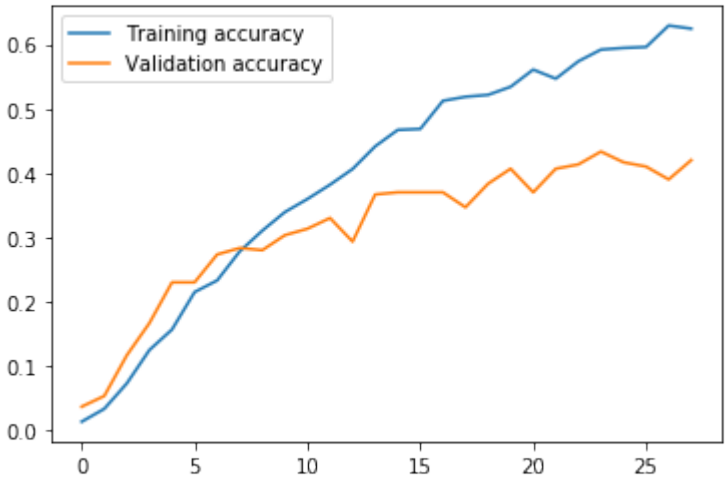

Figura 13: Evolución acc. con ResNet como extractor en V3 (epochs=28).

Para la siguiente parte, realizaremos un ajuste fino de la red **ResNet50** y realizaremos el entrenamiento sobre la red completa a la que le añadiremos un par de capas más totalmente conectadas. Como mínimo, necesitamos añadir una capa que contenga la capa de activación *softmax* que permita clasificar el problema, pero al igual que antes y con el objetivo de mejorar los resultados, añadiremos una capa de **Dropout** y otra **Dense** con activación relu.

Utilizaremos la clase *Model* para añadir el resto de capas al final de la capa **ResNet**.

```
# V1
x = resnet50.output
x = Dense(512, activation='relu')(x)
x = Dropout(0.75)(x)
last = Dense(num_classes, activation='softmax')(x)
new_model = Model(inputs = resnet50.input, outputs = last)
```

Definiremos los generadores de datos usando el parámetro *preprocess_input* y *validation_split*. Para usar la función de preprocesamiento de **ResNet** sobre cada entrada y realizar la división de train y validation respectivamente.

```
datagen_train = ImageDataGenerator(preprocessing_function=
    preprocess_input, validation_split=0.1)
datagen_test = ImageDataGenerator(preprocessing_function=
    preprocess_input)
```

Compilaremos el modelo y lo entrenaremos sobre los conjuntos de entrenamiento y validación. Haremos en este apartado varias pruebas modificando los parámetros de *ImageDataGenerator* y añadiendo normalización y aumento de datos. Fijaremos el número de épocas a 20, como llevamos haciendo en el resto de la práctica.

Como vamos a entrenar con la red **ResNet** completa, el proceso tardará mucho más que el anterior, tanto el de compilación como el de entrenamiento.

Por lo general, uno de los problemas de esta red definida con fine tunning, es que sobreentrena demasiado los datos. Usando un aumento de datos, como

hemos explicado anteriormente, deberíamos reducir el overfitting, ya que al aumentar el conjunto de datos de entrenamiento mejoraremos el accuracy de validation al poder tratar con el entrenamiento de más muestras y reduciremos el de entrenamiento al haber más muestras que entrenar.

Fine Tunning ResNet V1	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
validation_split=0.1	0.1127	0.951	2.1577	0.5233	0.4566
validation_split=0.1 + normalización	0.0564	0.9507	2.2620	0.5200	0.4613
zoom_range=0.0 + horizontal_flip=True (*)	0.0673	0.9533	2.2172	0.5533	0.4942
zoom_range=0.2 + horizontal_flip=True (*)	0.1353	0.9583	2.1447	0.5967	0.4863

Tabla 12: Resultados V1 modificaciones realizando ajuste fino a ResNet

(*)También implementa *validation_split=0.1* y normalización.

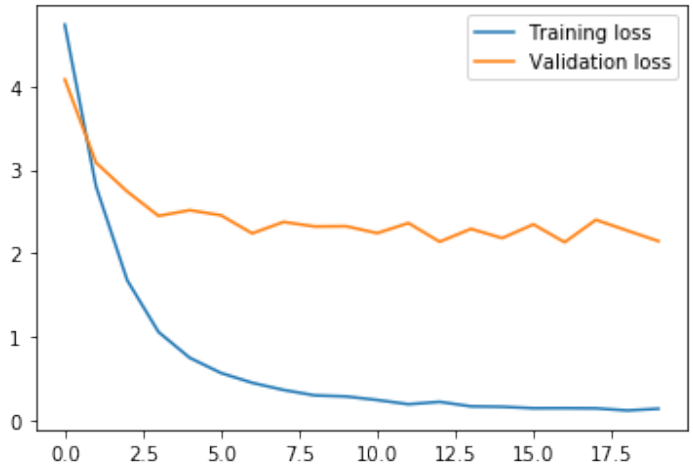

Figura 14: Evolución loss con ajuste fino de ResNet (V1).

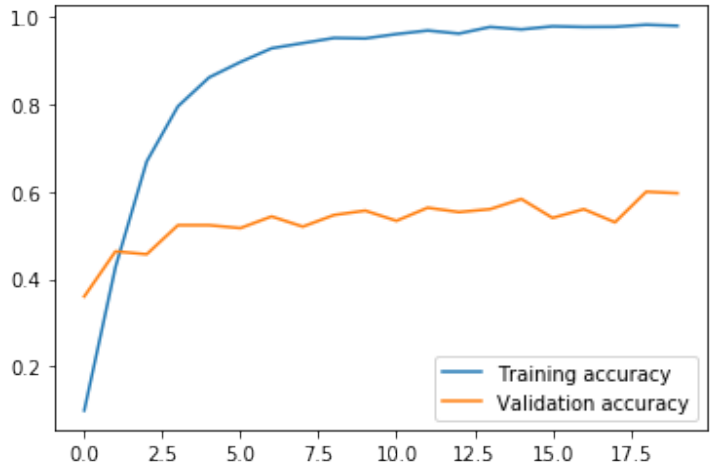

Figura 15: Evolución accuracy con ajuste fino de ResNet (V1).

A la vista de los resultados, las modificaciones consiguen buenos resultados en el conjunto de validación pero ni con aumento de datos conseguimos reducir el overfitting. Para ello, probaré a añadir dos capas de regularización con probabilidad 0.75 y 0.5 y una **Dense** con activación relu. Mostraremos los resultados obtenidos que deberían reducir el accuracy de train y mejorar o mínimo mantener los valores de validation.

```
# V2
x = resnet50.output
x = Dense(512, activation='relu')(x)
x = Dropout(0.75)(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
last = Dense(num_classes, activation='softmax')(x)
new_model = Model(inputs = resnet50.input, outputs = last)
```

(Había probado inicialmente añadiendo dos capas de 0.75 y la gráfica de validation se queda por encima de la de train, por lo que he optado por cambiarlo).

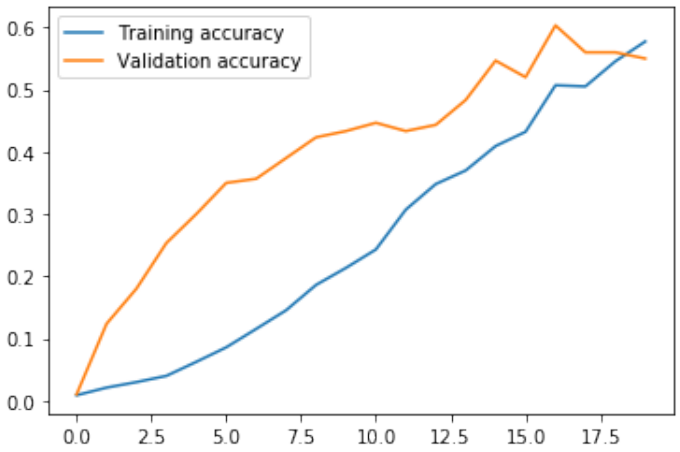

Figura 16: Evolución de Accuracy usando capas Dropout con prob. 0.75.

A continuación mostraré la tabla comparativa, en la que he añadido tres modificaciones y veremos si disminuye el overfitting como esperamos.

Fine Tunning ResNet V2	**Train**		**Validation**		**Test**
Modificación	loss	accuracy	loss	accuracy	accuracy
validation_split=0.1 + normalización	0.4030	0.9616	2.0793	0.6300	0.5417
zoom_range=0.0 + horizontal_flip=True (*)	0.4032	0.9606	2.0189	0.6400	0.5414
zoom_range=0.5 + horizontal_flip=True (*)	0.7669	0.911	2.9293	0.5300	0.45862

Tabla 13: Resultados V2 modificaciones realizando ajuste fino a ResNet

(*)También implementa *validation_split=0.1* y normalización.

Podemos ver que el modelo utilizado realiza sobreentrenamiento aún con todas las modificaciones que hemos hecho. Por tanto, probablemente la única forma para solucionar el problema de overfitting del modelo, sería modificar las capas de **ResNet** o preentrenar con otro conjunto de datos. Aún así, podemos ver como para la segunda modificación conseguimos bastante buena tasa de accuracy en validation a pesar de que la de train esté próxima a 1.

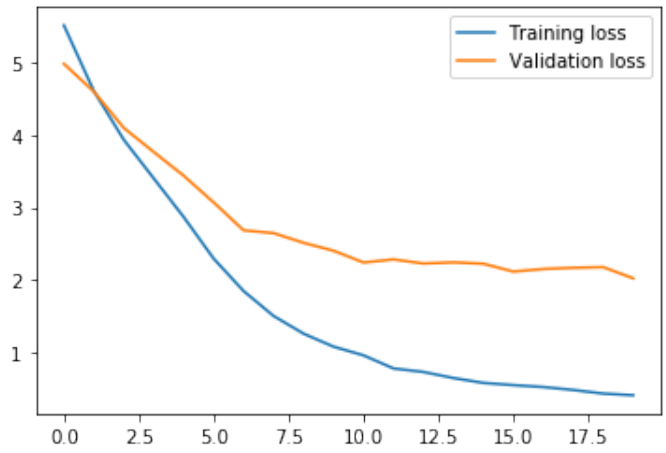

Figura 17: Evolución loss con ajuste fino de ResNet (V2-Mod 2).

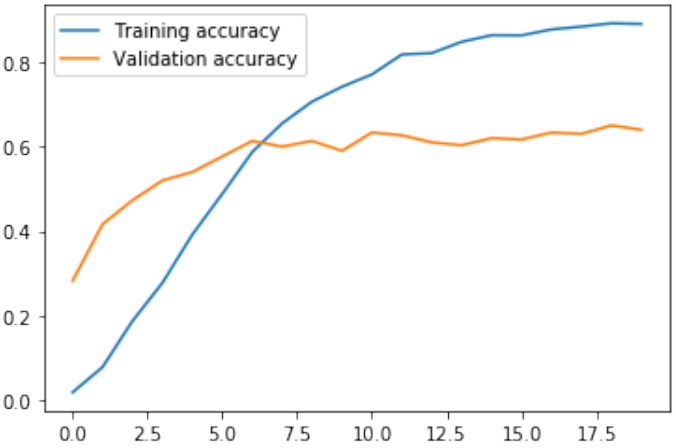

Figura 18: Evolución accuracy con ajuste fino de ResNet (V2-Mod 2).

Podemos observar como las gráficas de loss y accuracy se estabilizan. En el caso de la de accuracy se estanca en 0.6, por lo que nos da una idea de que este modelo funciona mejor como extractor de características que como red de entrenamiento para este conjunto de datos, ya que además de estabilizarse en 0.6, sobreentrena los datos como ya hemos comentado.

5. Bonus 1.

Consideraré parte del bonus la utilización de las capas de regularización **Dropout** utilizadas para mejorar los resultados obtenidos por los modelos implementados y reducir el overfitting de los mismos.

6. Bibliografía.

[1]. Documentación de Keras: https://keras.io/

- Optimizadores: https://keras.io/optimizers/
- **Dense**: https://keras.io/layers/core/
  Conv2D: https://keras.io/layers/convolutional/
  MaxPooling2D: https://keras.io/layers/pooling/
  BatchNormalization: https://keras.io/layers/normalization/
- Funciones Pérdida: https://keras.io/losses/
- Applications: https://keras.io/applications/

[2]. Diapositivas de clase.

[3]. Plantillas y diapositivas de la práctica en Keras.

[4]. Blog optimizers: https://ruder.io/optimizing-gradient-descent/

[5]. Early Stopping: https://elitedatascience.com/overfitting-in-machine-learning#how-to-prevent