Redes Neuronales Convolucionales

Luis Antonio Ortega Andrés

29 de noviembre de 2019

1. Introducción

El objetivo de esta práctica es practicar el diseño y entrenamiento de redes neuronales convolucionales usando Keras. Para ello, se ha hecho uso de diversas funciones auxiliares, que ayudarán simplificando el código a la hora de realizar los entrenamientos y evaluaciones.

1.1. Compilar modelos

En esta práctica compilamos los modelos utilizando el optimizador `SGD`. Por ello declaramos una función que nos sirva para compilarlos de forma mas cómoda.

```
def model_compile(model):
    model.compile(
        loss = keras.losses.categorical_crossentropy,
        optimizer = SGD(lr = 0.01, decay = 1e-6, momentum = 0.9, nesterov = True),
        metrics = ['accuracy']
    )
```

1.2. Ajustar datagen

En la práctica utilizaremos objetos `datagen` de la clase `ImageDataGenerator` para realizar el entrenamiento y la evaluación de los modelos. Estos `datagen` deberán ajustarse a los datos cuando se realice normalización. Para simplificar este ajuste, definimos una función que acepta dos parámetros:

- `data_generator`. Función que devuelve un `datagen`.
- `x_train`. Conjunto sobre el que ajustar el `datagen`.

```
def fitted_datagen(data_generator, x_train):
    datagen = data_generator()
    datagen.fit(x_train)

    return datagen
```

Según la documentación de «Keras», es necesario ajustar un `datagen` cuando se va a realizar `feature_center`, `featurewise_std_normalization` o `zca_whitening`. De forma que solo utilizaremos esta función cuando queramos utilizar un `datagen` que haga uso de alguna de estas funciones.

1.3. Entrenamiento

Para el entrenamiento de cada uno de los modelos, se utilizará la función `train`. Esta función tiene los siguientes parámetros:

- `model`. Modelo de la red a entrenar.
- `x_train`. Conjunto de entrenamiento.
- `y_train`. Etiquetas del conjunto de entremiento.
- `batch_size`. Tamaño del `batch`.
- `epochs`. Número de epocas del entrenamiento.
- `train_datagen`. Objeto de la clase `ImageDataGenerator` a utilizar. Puede ser `None`.
- `verbose`. El mismo parámetro se utilizará para la salida del entrenamiento.
- `val_split`. Proporción de validación.
- `shuffle`. El mismo parámetro de utilizará para las funciones de «Keras».

Si el valor `train_datagen` es nulo, entonces el modelo se entrena usando la función `fit` de «Keras» sobre el modelo. En caso de no ser nulo, se utilizará como parámetro de la función `fit_generator` de «Keras».
En caso de utilizar un `datagen` implementamos también «EarlyStopping», con una paciencia de 10 épocas y de forma que restaure los pesos que mejor resultado dieron en el conjunto de validación.
La estructura principal de la función es la siguiente:

```
if train_datagen is None:
   return model.fit(x_train, y_train, batch_size, epochs,
                    verbose = verbose, validation_split = val_split,
                    shuffle = shuffle)

else:
   return model.fit_generator(
            generator = train_datagen.flow(x_train, y_train,
                                           batch_size, subset='training'),
            steps_per_epoch = len(x_train)*0.9/batch_size,
            epochs = epochs,
            validation_data = train_datagen.flow(x_train, y_train,
                                                 batch_size, subset='validation'),
            validation_steps = len(x_train)*0.1/batch_size,
            verbose = verbose,
            callbacks = [EarlyStopping(monitor = 'val_acc', patience = 10,
                                       restore_best_weights = True)],
            shuffle = shuffle
            )
```

1.4. Evaluación

Esta función nos permitirá evaluar nuestro modelo sobre un conjunto de datos. Los parámetros que tiene son los siguientes:

- `model`. Modelo de la red a entrenar.
- `x_test`. Conjunto de entrenamiento.
- `y_test`. Etiquetas del conjunto de entremiento.
- `test_datagen`. Objeto de la clase `ImageDataGenerator` a utilizar. Puede ser `None`.
- `verbose`. El mismo parámetro se utilizará para la salida del entrenamiento.

  Al igual que la función de entrenamiento, esta función utilizará `evaluate` en caso de ser el `datagen` nulo y `evaluate_generator` cuando no lo sea.

1.5. Extractor de características

Esta función nos permitirá predecir las características de un conjunto de datos utilizando un modelo. Los parámetros son los siguientes.

- `model`. Modelo de la red a entrenar.
- `data`. Conjunto de datos de los que extraer las características.
- `datagen`. Objeto de la clase `ImageDataGenerator` a utilizar. Puede ser `None`.
- `verbose`. El mismo parámetro se utilizará para la salida de la predicción.

  Al igual que las dos funciones anteriores, esta función utilizará `predict` en caso de ser el `datagen` nulo y `predict_generator` cuando no lo sea.

2. Apartado 1

En este apartado se nos pide crear un modelo `BaseNet`, y con él trabajar sobre el conjunto de datos `Cifar100` reducido para tratar solo con 25 clases.

Utilizaremos un conjunto de entrenamiento de 12500 imágenes y uno de prueba de 2500. Utilizando una partición de validación del 10 %.

Para ello declaramos el modelo según se indica en el guión de prácticas.

```
def base_net_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())
```

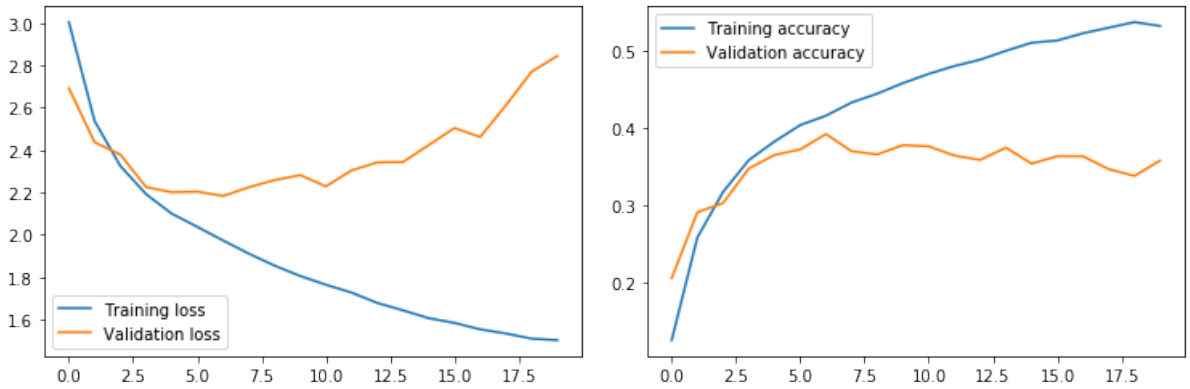
```
    model.add(Dense(50, activation='relu'))
    model.add(Dense(25, activation='softmax'))

    return model
```

Tras declarar y compilar el modelo, guardamos los pesos aleatorios con los que se ha inicializado, de forma que al hacer comparaciones con el mismo modelo, podamos cargarlos de nuevo. Para ello utilizamos las funciones `set_weights` y `get_weights`.

Si ejecutamos el modelo obtenemos los siguientes resultados en el conjunto de prueba.

```
PÉRDIDA:  2.89583932762146
PRECISIÓN:  0.3548
```

La evolución del modelo durante el entrenamiento ha sido la siguiente.



Como podemos observar, el modelo se sobreajusta rapidamente a los datos de entrenamiento, probablemente debido a la simplicidad del mismo.

3. Apartado 2

En este apartado se nos pide realizar una serie de mejoras al modelo y realizar «Data Augmentation» para conseguir acercarnos a un 50 % de precisión en el conjunto de prueba.

3.1. Data augmentation

Para realizar el «Data Augmentation», utilizaremos la clase `ImageDataGenerator` que nos facilita «Keras». Esta nos permite declarar unos generadores o `datagen`, que nos permitirán realizar transformaciones a los datos conforme se utilizan en el entrenamiento y la evaluación.

Para declarar los `datagen`, lo hacemos por parejas de forma que uno de ellos corresponde al conjunto de entrenamiento y otro al de prueba. El correspondiente al conjunto de prueba tendrá los parámetros de normalización que tenga el de entrenamiento. Esto es necesario ya que si entrenamos la red con datos normalizados, los datos de prueba deberán estar normalizados también.

Las funciones `train_data_generator_with_whitening` y `test_data_generator_with_whitening` crean 2 `datagen` donde el primero centra los datos, aplica `whitening` y volteo horizontal, por lo tanto el segundo realiza solo el centreo y el `whitening`.

A su vez, las funciones `train_data_generator_without_whitening` y `test_data_generator_without_whitening` crean otros 2 `datagen` donde la primera normaliza los datos y añade una variable de desplazamiento aleatorio de hasta el 10 % de las imagenes, además del volteo horizontal. De forma que el segundo realiza solo la normalización.

Estas 4 funciones se han elegido después de hacer varias combinaciones con los distintos parámetros de los que dispone la clase, muchos de ellos no han resultado en ningún cambio significativo en los resultados (por ejemplo la rotación). Sin embargo, son las que mejores resultados han dado de las que se han probado con y sin `whitening`.

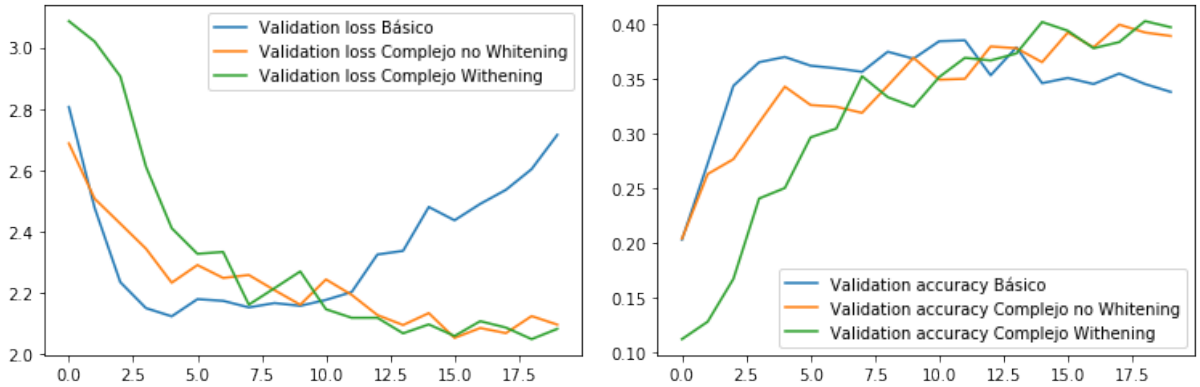
Para ajustar los `datagen` al conjunto de entrenamiento debemos utilizar nuestra función auxiliar `fitted_datagen` sobre dicho conjunto. De esta forma, los valores de normalización de ambos generadores serán los mismos.

Para hacer esto se podría tener un solo `datagen` y utilizar la función de `standardize` que nos facilita `Keras`, sin embargo, no permite hacer `whitening`. Por ello he optado por utilizar dos `datagen` aunque sea necesario ajustar los dos.

Veamos los resultados que nos dan ambas parejas de generadores

```
 - RESULTADOS SIN WHITENING -
PÉRDIDA:  1.9068185174722398
PRECISIÓN:  0.432
 - RESULTADOS CON WHITENING -
PÉRDIDA:  2.014278331503267
PRECISIÓN:  0.4304
```

En las siguientes gráficas podemos comparar como evoluciona el modelo que tenemos con ambas parejas de generadores en comparación a como evolucionaba sin ellos.



Como vemos, utilizar o no `whitening` no parece tener una gran repercusión en el modelo que tenemos actualmente, además, esta opción hace que el aprendizaje sea más lento. Por ello, el resto de pruebas las haremos sin `whitening`.

3.2. Modelos

Discutimos ahora el nuevo modelo que se va a utilizar. Como hemos visto en los resultados del modelo básico, este no tiene la potencia necesaria para aprender las características del conjunto de datos.

Para aumentar la potencia de aprendizaje de la red, vamos a crear dos bloques de dos convoluciones cada uno de ellos, separadas por una capa de activación que rompa la linealidad.

Tras cada uno de estos bloques añadimos una capa `MaxPooling2D` para quedarnos las características mas importantes de aquellas que ha abstraido el bloque. Debido a que esto disminuye considerablemente el tamaño de la imagen, optamos por añadir un «frag» a una convolución de cada bloque para que no disminuya la imagen.
Tras la capa `MaxPooling2D` añadimos una capa `Dropout`, de forma que cada bloque obtiene información parcial del bloque anterior y no se «acostumbre» a que este haga todo el trabajo de aprendizaje.

El último bloque de la red consiste en un par de capas totalmente conectadas separadas por otro `Dropout`.

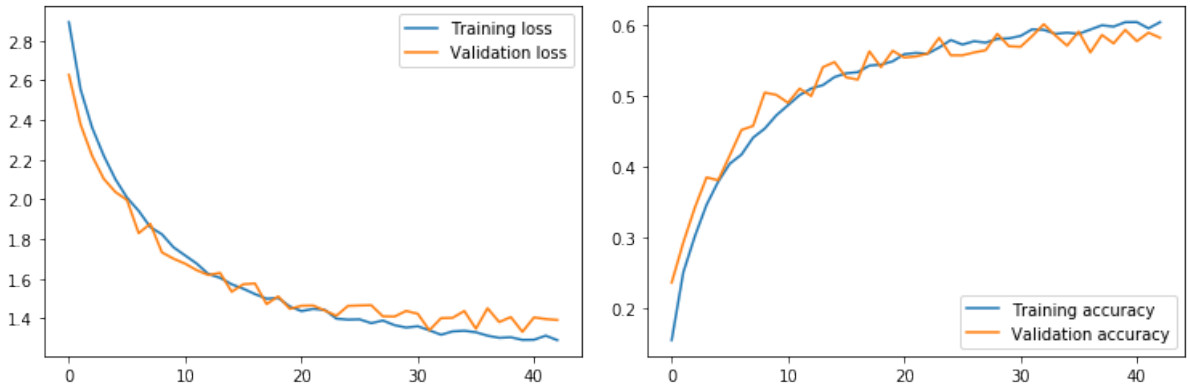
La estructura del modelo sería la siguiente:

Layer No.	Layer Type	Kernel size	Input-Output dim.	Input-Output channels
1	Conv2D	3	32-32	3-32
2	Relu	-	-	-
3	Conv2D	3	32-30	32-32
4	Relu	-	-	-
5	MaxPooling2D	2	30-15	-
6	Dropout (0.25)	-	-	-
7	Conv2D	3	15-15	32-64
8	Relu	-	-	-
9	Conv2D	3	15-13	64-64
10	Relu	-	-	-
11	MaxPooling2D	2	13-6	-
12	Dropout (0.25)	-	-	-
13	Flatten	-	6-2304	-
14	Dense	-	2304-512	-
15	Relu	-	-	-
16	Dropout (0.5)	-	-	-
17	Dense	-	512-25	-

Probamos el modelo utilizando el `datagen` que mejor resultado nos ha dado en las ejecuciones con el modelo más simple (sin `whitening`).

```
PÉRDIDA:  1.2674647610968077
PRECISIÓN:  0.6108
```

Veamos como evoluciona el aprendizaje del modelo.



Como vemos, no solo se obtienen mejores resultados finales que al utilizar el primer modelo, si no que además la curva de aprendizaje del modelo nos sugiere que es menos propenso al sobreajuste.

Para mejorar nuestro modelo actual, consideramos añadir unas capas de normalización. Nos preguntamos entonces si debemos ponerlas antes o después de las capas de activación de cada unas de las convoluciones.

Hacemos ambos experimentos sin cambiar el `datagen` obteniendo los siguientes resultados.

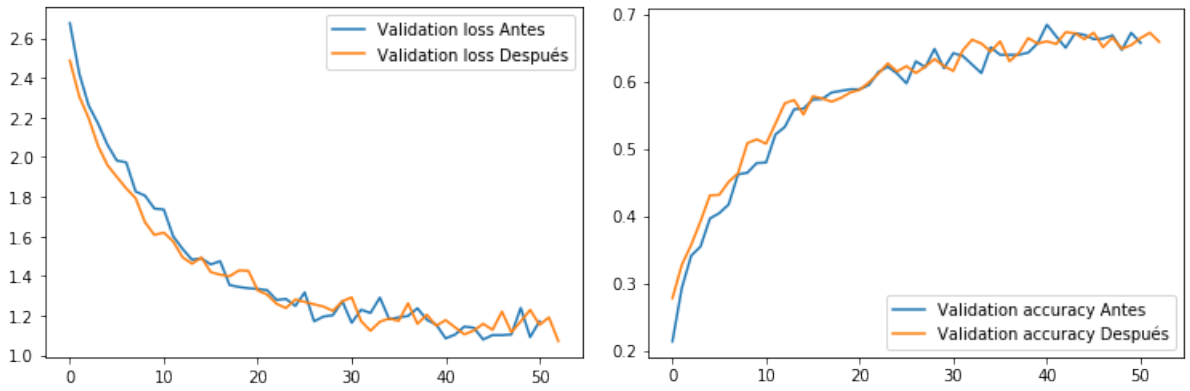
```
 -- BATCHNORMALIZATION ANTES --
PÉRDIDA:  0.9689475336031279
PRECISIÓN:  0.6968
```


```
 -- BATCHNORMALIZATION DESPUÉS --
PÉRDIDA:  1.0282180929724063
PRECISIÓN:  0.6844
```

Como podemos observar, añadir dichas capas ha permitido mejorar aún más los resultados del modelo.

Comparamos ahora el crecimiento de ambas en la misma gráfica, tal como se puede observar, se obtienen resultados parecidos y ambos modelos presentan un crecimiento similar.

Además, ambos han realizado aproximadamente el mismo número de épocas utilizando «EarlyStopping» siendo el máximo prefijado 100 (con idea de que no se alcanzara).



Ahora podemos experimentar si con un modelo mas complejo como el que tenemos ahora, un `datagen` que realice `whitening` obtendría mejores resultados.

```
 -- BATCHNORMALIZATION DESPUÉS Y WHITENING --
PÉRDIDA:  1.1020758721511346
PRECISIÓN:  0.6764
```

Como podemos ver, los resultados siguen siendo similares, sin embargo, en la evolución del modelo podemos observar como ahora se sobreajusta más a los datos.

Ya sea por el conjunto de datos o por los modelos que hemos utilizado, el uso de `whitening` no parece recomendable por la carga computacional que añade y los resultados que hemos obtenido.

4. Apartado 3

En este apartado se nos piden dos cosas sobre un nuevo conjunto de datos «Caltech».

- Usar «Resnet50» como un extractor de características.
- Hacer un ajuste fino de toda la red «Resnet50».

4.1. Extractor de características

Para realizar la extracción de características declaramos la red «Resnet» de forma que tenga los pesos de haber entrenado con el conjunto de datos «Imagenet», no incluya la última capa y realice un `GlobalAveragePooling` de forma que la salida sea un vector 1-dimensional de características.

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

La función `load_preprocessed_caltech_data` se encarga de cargar los datos de «Caltech» tal y como se nos ha indicado en el guión y aplicarles `preprocess_input` de forma que estos tengan el formato de los datos con los que se ha entrenado «Resnet». Esto lo podemos hacer ahora o indicarle la función a un `datagen` y que él la aplique a cada imagen.

Utilizamos nuestra función extractora de características sobre los conjuntos de datos.

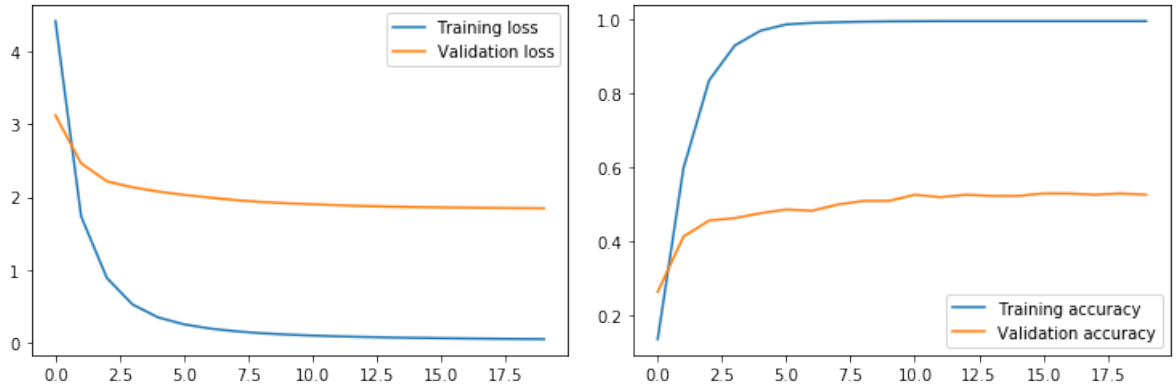
```
caltech_train_features = extract_features(resnet50, caltech_train)
caltech_test_features = extract_features(resnet50, caltech_test)
```

Ahora podemos utilizar estos conjuntos junto con sus etiquetas para clasificarlos. Para ello declaramos 3 modelos muy simples.

- `basic_classifier_model`. Consiste en una única capa densa, para realizar la clasificación.
- `two_layers_classifier_model`. Consiste en dos capas densas totalmente conectadas, separadas por una capa `Relu`.
- `two_layers_dropout_classifier_model`. Consiste en dos capas densas totalmente conectadas, separadas por una capa `Relu` y una capa `Dropout(0.75)`.

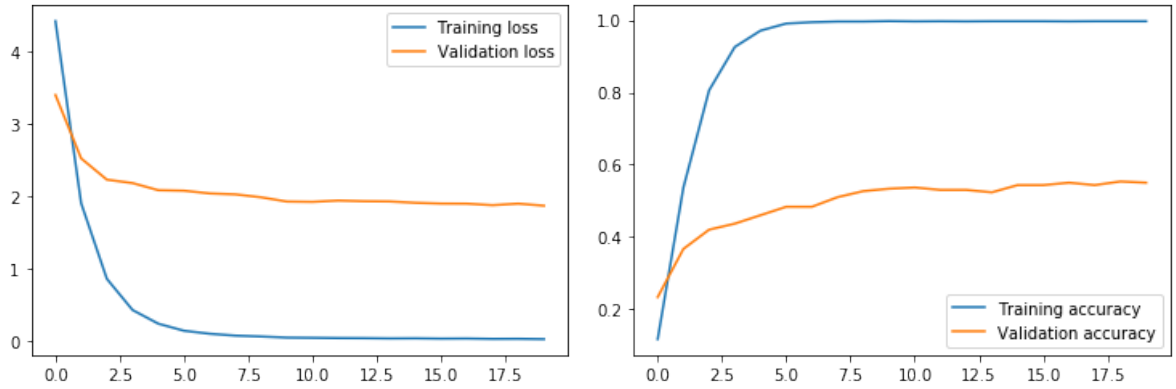
Veamos ahora que conseguimos con cada uno de estos modelos y por qué añadimos esa capa `dropout`. Ejecutamos el primero de ellos, que sería el correspondiente a ejecutar «Resnet» redimensionando la salida, de forma que podemos ver como de bien o mal lo hace la red que estamos utilizando por si sola.

```
PÉRDIDA:  2.4028188528261523
PRECISIÓN:  0.42927794258684127
```



Como vemos el modelo sobreajusta rapidamente al conjunto de datos que tenemos, que es lo mismo que nos pasa cuando utilizamos 2 capas totalmente conectadas en lugar de 1.

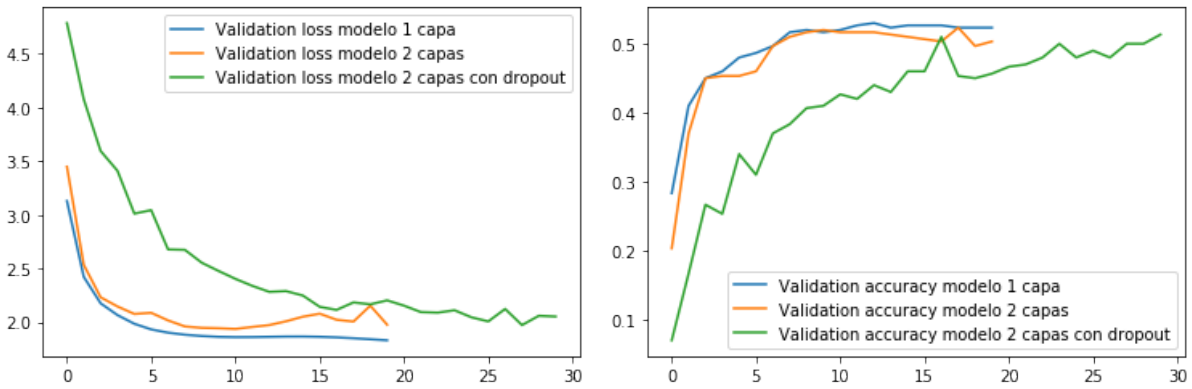
```
PÉRDIDA:  2.51853793688509
PRECISIÓN:  0.4329047147596075
```



Ante este sobreajuste, nos planteamos evitarlo utilizando una capa de regularización entre las capas totalemente conectadas.

```
PÉRDIDA:  2.5000689025763196
PRECISIÓN:  0.39235080769264064
```

Comparamos ahora todos los modelos en la misma gráfica.



Con ninguno de los modelos que se han probado se ha conseguido que la clasificación ascenda de ese 40 %. También se han probado modelos donde se añadían más capas `Dropout` o `BatchNormalization`, pero en esos casos el modelo no aprendía, probablemente debido a que no le estabamos permitiendo utilizar las características que habiamos extraido.

4.2. Ajuste fino

Para realizar el ajuste fino del modelo «Resnet», utilizamos API funcional de `Keras` que nos permite enlazar modelos. Información.

Para ello seguimos los siguientes pasos:

- Declaramos el modelo «Resnet» igual que haciamos en el apartado de extracción de características.
- Tomamos el `output` del modelo (`x = resnet50.output`).
- Sobre dicho output utilizamos la API para añadir capas (`x = Dense(200)(x)`).
- Definimos un modelo dado por el `input` de «Resnet» y el `output` que hemos construido (`model = Model(inputs=resnet50.input, output = x)`).

Esto nos permite construir de forma sencilla modelos que incluyan el de «Resnet». Con esta API podríamos crear modelos con multiples salidas, grafos acíclicos o modelos con capas compartidas, pero esto se escapa a lo que buscamos.

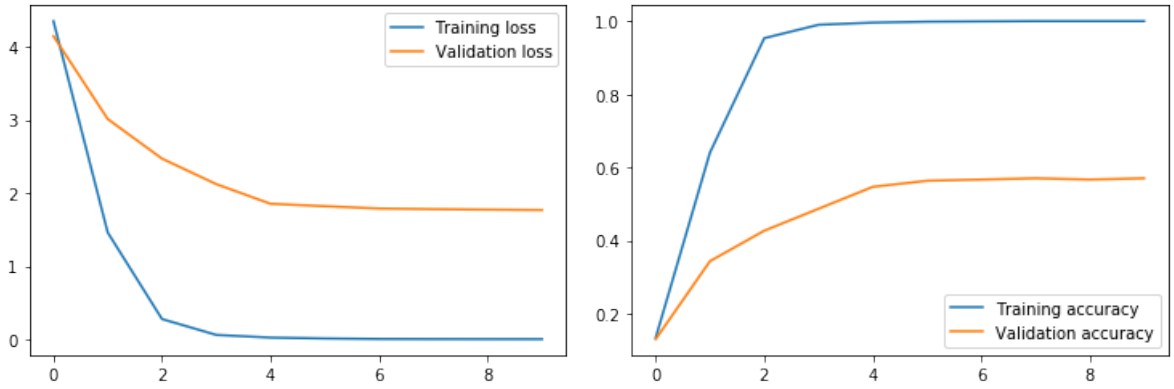
Para este apartado hemos definido 3 modelos distintos:

- `one_layer_resnet_model`. Una capa densa de 200 al final del modelo «Resnet». Lo utilizaremos de modelo de comparación ya que es consiste en dar los resultados que genera «Resnet» directamente.
- `two_layers_resnet_model`. Dos capas totalmente conectadas, la primera de 1024 y la segunda de 200. Separadas por una capa de activación.
- `two_layers_conv_resnet_model`. En este modelo, hemos tomado «Resnet» sin aplicar el GlovalAveragePooling, en este caso las imagenes que nos devuelve tienen tamaño 7x7. Aprovechamos esto para añadir las siguientes capas:
  - `Conv2D(64, (3, 3), activation='relu')`
  - `Dropout(0.2)`
  - `GlobalAveragePooling2D()`
  - `Dense(1024, activation='relu')`
  - `Dropout(0.4)`
  - `Dense(200, activation='softmax')`

Veamos los resultados de las ejecuciones en este mismo orden. Para el primer modelo obtenemos

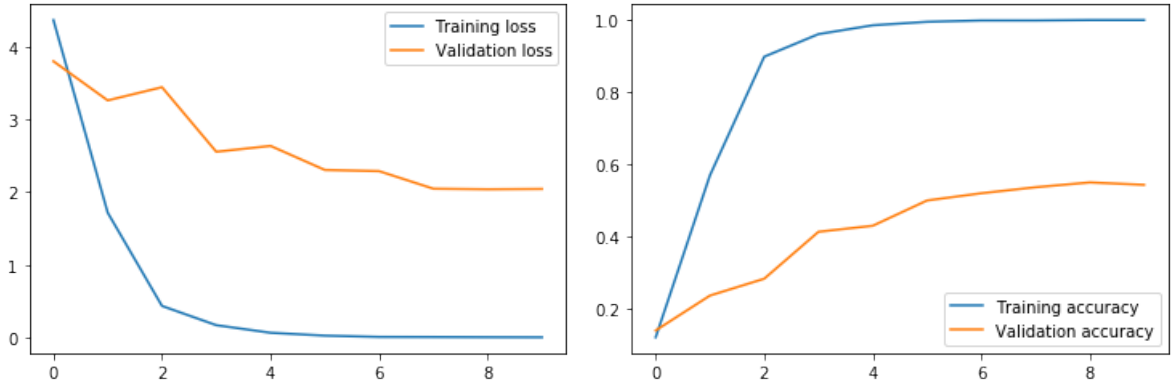
```
PÉRDIDA:  2.316258090304102
PRECISIÓN:  0.4467523903922205
```

Donde podemos ver que el modelo realiza un rápido sobreajuste a los datos de entrenamiento.



Esto mismo sucede cuando añadimos una segunda capa totalmente conectada.

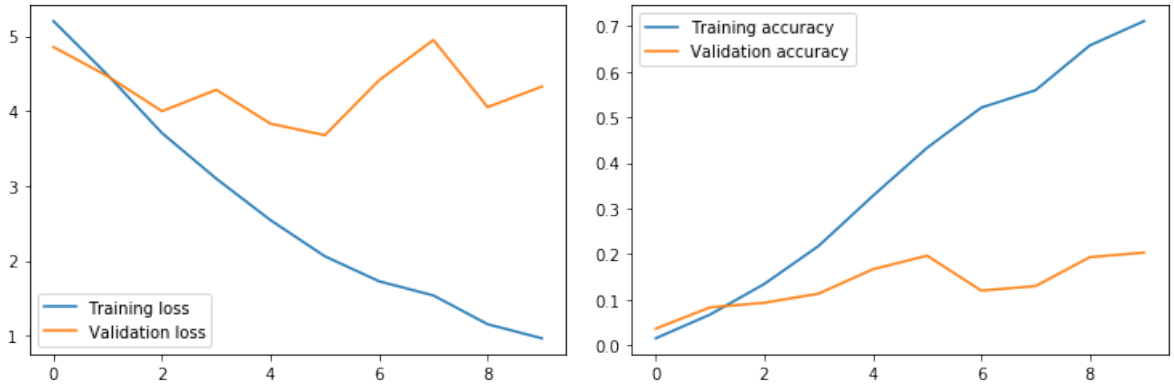
```
PÉRDIDA:  2.590243388039815
PRECISIÓN:  0.4256511705762044
```



Si intentamos solucionar este sobreajuste añadiendo capas de regularicación igual que en el apartado anterior, resulta que no es suficiente. Por esto, probamos ahora con un modelo algo mas complejo en sus capas finales.

Sin embargo, los resultados no son buenos y aunque el modelo no sobreajusta a tanta velocidad, lo sigue haciendo ademas de empeorar notablemente los resultados.

```
PÉRDIDA:  4.858583949007612
PRECISIÓN:  0.1678206395717725
```



La sensación con este ajuste fino es que cuantas mas capas se añadieran al final de «Resnet», peor era el modelo. Esto nos lleva a pensar que deberiamos quitar mas capas finales de «Resnet» si queremos obtener un rendimiento mejor.