VC

Visión por computador

# Trabajo 2: Redes neuronales convolucionales

**Autor:** Jesus Medina Taboada

**Correo:** jemeta@correo.ugr.es

**Profesor:** Pablo Mesejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2020 - 2021

# Índice general

## 0.1. Introducción

El objetivo de este trabajo es obtener experiencia práctica en el diseño y entrenamiento de redes neuronales convolucionales profundas, usando Keras, una de las bibliotecas de redes neuronales de código abierto más relevantes y escrita en python.
Trabajaremos a partir de una arquitectura base de red que se proporciona(BaseNet), aprenderemos a experimentar con ella y mejorarla, añadiendo, modificando o suprimindo capas de dicha arquitectura y cuya tarea principal será clasificar imágenes en 25 categorías.

# Capítulo 1

# BaseNet en CIFAR100

En este primer ejercicio empezaremos trabajando y definiendo un modelo base llamado BaseNet familiarizandonos con su arquitectura, los hiperparámetros y capas que lo conforman. Finalmente entrenaremos el modelo, extraeremos los valores de accuracy y loss function y presentaremos los resultados usando unas gráficas comparativas.

El **conjunto de datos** que usaremmos es **CIFAR100**, trabjaremos con una parte de este conjunto el cual consta de 60K imágenes en color de dimensión 32x32x3 (RGB) de 100 clases distintas, con 600 imágenes por clase. Hay 50K imágenes para entrenamiento y 10K imágenes de prueba. Para el desarrollo de esta práctica solo consideraremos 25 clases de las 100, por tanto el conjunto de entrenamiento tiene 12500 imágenes y el de prueba 2500. Del conjunto de entrenamiento se usará un 10 % para validación.

## 1.1. Creación del modelo

Vamos a desarrollar una red convolucional básica para clasificar imágenes del conjunto de datos CIFAR100.
Utilizamos la función cargarImagenes 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.
Como mencionamos anteriormente solo usaremos las 25 primeras clases de las 100 disponibles en el conjunto de datos, reduciendo así el tamaño del problema.
Las imágenes de entrada las trataremos gracias al uso de la clase ImageDataGenerator, creando cuando sea necesario un datagen que permitirá el flujo

de imágenes.
Nos aseguramos de reservar un 10 % de datos de entrenamiento para validación, mediante el argumento validation_split = 0.1.
Por otra parte tenemos la función compile que se encarga de definir el optmizador y compilar el 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, que mide el desempeño de un modelo de clasificación en el que las predicciones son una probabilidad).

Antes de definir el modelo que se nos propone veremos brevemente las capas con las que trabajaremos:

- **Convolución (Conv2d):** capa que realiza convoluciones sobre la entrada dando como parámetros un kernel_size, y el número de filtros diferentes que harán estas capas. La salida dependerá del tamaño del kernel y del número de filtros que le indiquemos que aplique.
- **Ativación Relu:** capa que nos aporta la no-linealidad al modelo. Aplica a cada elemento xi (píxeles) de la entrada X la función:

$$g(xi) = max(0, xi) \tag{1.1}$$

- **Pooling:** dado un tamaño para obtener subvectores de nuestro vector (multi o unidimensional) de entrada tomará el máximo de ese vector como salida por cada subvector. Realmente hay muchas variantes de Pooling como MinPooling o AveragePooling, pero nosotros usaremos MaxPooling la mayoría de las veces.
- **Flatten:** hace que nuestra matriz pase a ser un vector, añadiendo cada fila de la misma al final de la anterior, empezando por la primera.
- **Dense:** aplica a la entrada un producto por un vector que tendrá pesos que se mejorarán durante el entrenamiento.
- **Activación Softmax:** obtiene un vector que tendrá tantas entradas como número de clases tengamos para clasificar, y tendrá como output en la posición i-ésima la probabilidad de la imagen de pertenecer a la clase i-ésima. Utilizará como distribución de probabilidad la exponencial.

Figura 1.1: Model a implementar

| Layer No. | Layer Type | Kernel size conv layers | Input \| Output dimension | Input \| Output channels 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 | - |

Figura 1.2: Código de la implementación

```
#Devuelve el modelo de referencia BaseNet
def basenet_model():
    model = Sequential()

    # Layer 1 y 2
    model.add(Conv2D(6,
                     kernel_size = (5, 5),
                     activation = 'relu',
                     input_shape = (32, 32, 3)))

    # Layer 3
    model.add(MaxPooling2D(pool_size = (2, 2)))

    # Layer 4 y 5
    model.add(Conv2D(16,
                     kernel_size = (5, 5),
                     activation='relu'))

    # Layer 6
    model.add(MaxPooling2D(pool_size = (2, 2)))

    # Flatten
    model.add(Flatten())

    # Layer 7 y 8
    model.add(Dense(50,
                    activation = 'relu'))

    # Layer 9 + softmax
    model.add(Dense(N,
                    activation = 'softmax'))

    return model, "basenet"
```

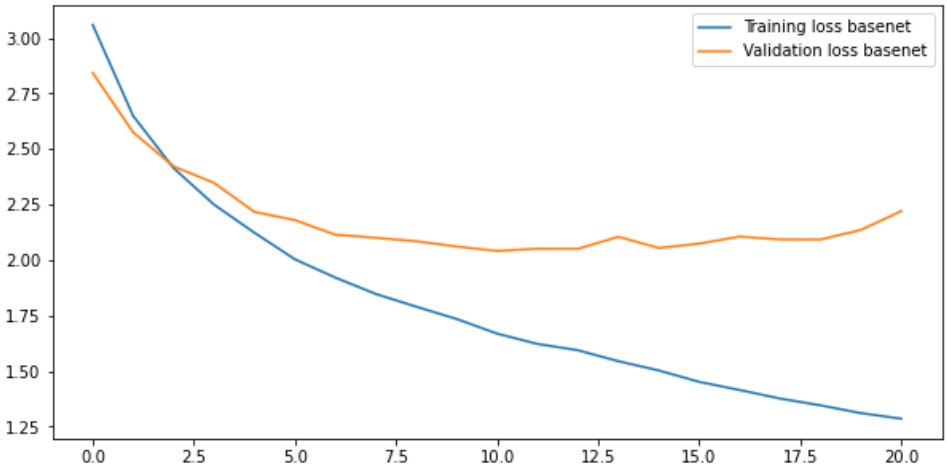
## 1.2. Entrenamiendo y resultados

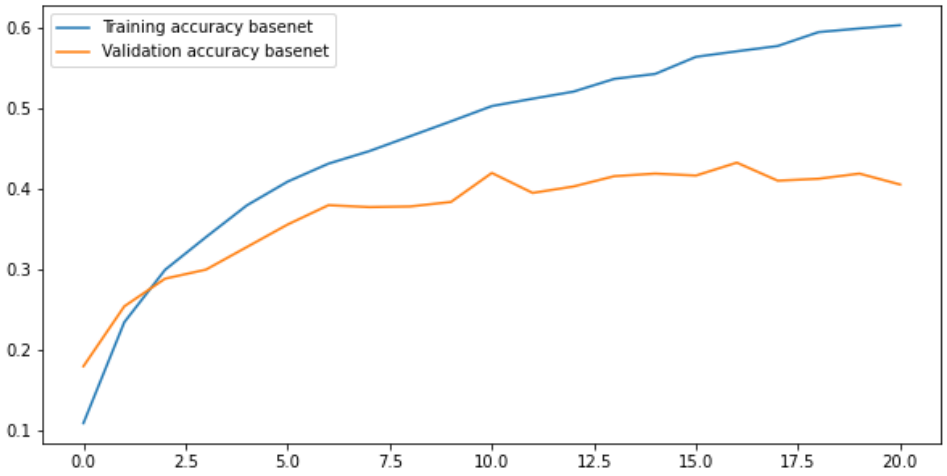
Una vez tenemos definido el modelo procedemos a entrenarlo. A continuación se muestra como entranamos el modelo, dividiendo los datos de entrenamiento y validación todo en la misma sentencia.

```
# Entrenamos el modelo
hist = model.fit_generator(datagen.flow(x_train,
                                        y_train,
                                        batch_size = BATCH_SIZE,
                                        subset = 'training'),
                           epochs = EPOCHS,
                           steps_per_epoch = len(x_train) * (1 - SPLIT) / BATCH_SIZE,
                           verbose = 1,
                           validation_data = datagen.flow(x_train,
                                                          y_train,
                                                          batch_size = BATCH_SIZE,
                                                          subset = 'validation'),
                           validation_steps = len(x_train) * SPLIT / BATCH_SIZE,
                           callbacks = callbacks_list)
```

El parámetro batch_size controla el número de imágenes que se procesan antes de actualizar el modelo(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. Para ser más eficientes en tiempo se ha añadido un callbacks de tipo EarlyStopping que permiten detener el entrenamiento cuando no se ha mejorado en error o la precision de validación en un número determinado de épocas(parámetro PATIENCE.

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 mostrarStats y mostrarEvolution nos permiten mostrar gráficas y estadísticas del entrenamiento y la evaluación del modelo.




El **resultado** es 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.Al ejecutar el modelo, obtenemos los siguientes resultados:

- **TEST LOSS=** 1.9816 **TEST ACCURACY=** 0.4273
- El modelo se ha entrenado durante 26 épocas, obteniéndose los mejores pesos en la época 16.
- A partir de unas 10 épocas el valor de la función de pérdida comienza a subir, mientras que el valor de acierto del modelo se mantiene constante o incluso disminuye.
- Esto nos indica que el modelo ha aprendido lo suficiente y se produce el fenómeno de overfitting, pues tanto la pérdida como la precisión llegan

a ser mucho mejores en el entrenamiento cuando hay un aumento de épocas.

- Recordar que en cada ejecución podemos obtener diferentes resultados, pues los valores de los pesos se inicializan aleatoriamente cada vez.

# Capítulo 2

# Mejora del modelo BaseNet

Nuestro objetivo ahora es modificar el modelo que acabamos de crear haciendo ajustes de arquitectura e implementación, construyendo una red profunda mejorada para obtener una mayor precisión en la clasificación de imágenes.
A continuación vamos a hacer y mostrar varias modificaciones de las muchas que se proponen. Con ello conseguiremos que el nuevo modelo incremente considerablemente sus capacidad de acierto en clasificación.

## 2.1. Normalización de datos

Lo primero que haremos sera normalizar nuestros datos de entrada para disminuir la complejidad del problema. Transformamos los datos con varianza = 1 y media = 0, para conseguirlo 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. Por último aplicamos esta misma modificación al conjunto de datos test.
Tras estas operaciones ejecutamos el modelo de nuevo, los resultados son muy similares, incluso podrían darse un poco inferiores, pero los datos no son significativos y siempre es mejor normalizar los datos.

**TEST LOSS=** 2.0426
**TEST ACCURACY=** 0.4291

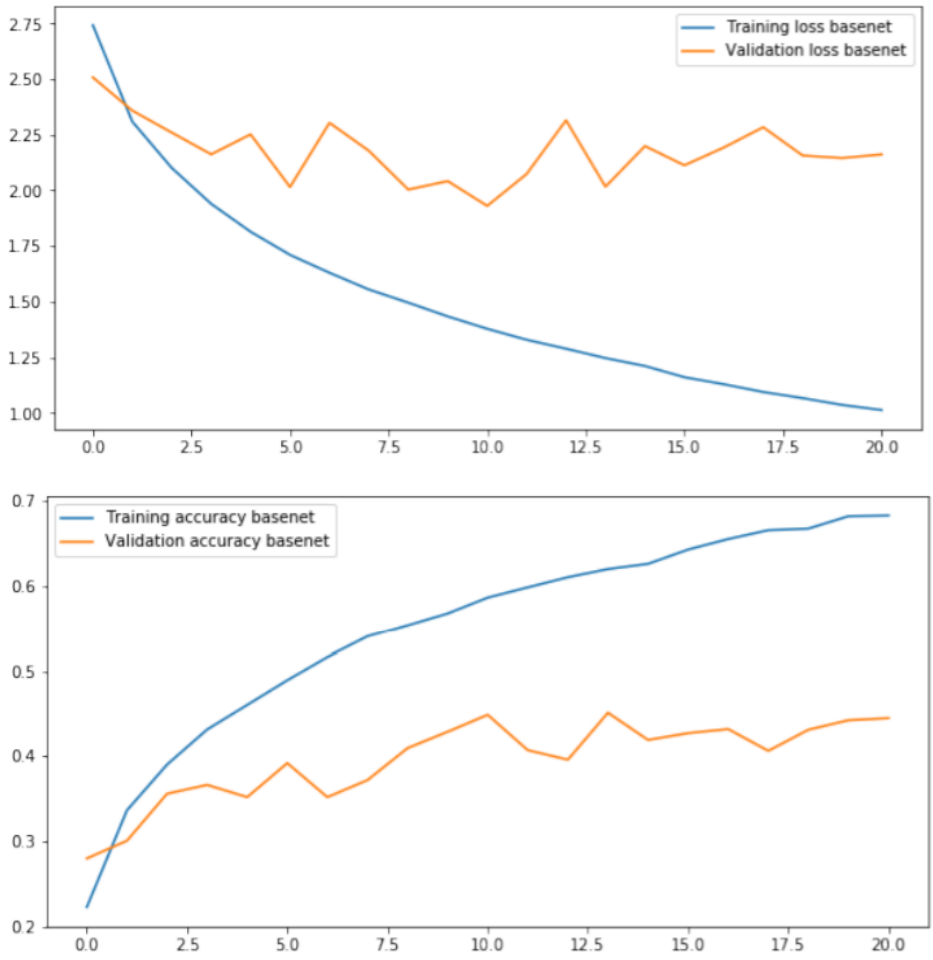
## 2.2. Batch Normalization

Batch Normalization es una técnica para mejorar el funcionamiento, velocidad y estabilidad de nuestra red neuronal convolucional. Consiste en normalizar según batches, es decir, tomar un batch de imágenes que se tomará del tamaño batch_size que hayamos utilizado, y hacer una normalización usando solo esas imágenes.
Nuestra intención es añadir alguna capa para reducir el overfitting que como vimos anteriormente, sufre nuestra red.

Por lo tanto esta nueva modificación añadirá 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.
Existen dos variantes de esta modificación, poner la capa antes o después de las capas convolucionales, es decir, antes o después de la activación. Después de algunas pruebas hemos decidido usar el criterio de ponerla despues de cada capa pero no se han notado diferencias significativas entre ambas metodologías.

Los resultados son los siguientes:
**TEST LOSS=** 1.8043
**TEST ACCURACY=** 0.4721

Un punto important a tener en cuenta es el uso 'use_bias = False' en las capas que preceden a la normalización. Con este parámetro indicamos a estas capas que no añadan el sesgo que es una constante aditiva, ya que en las capas de normalización ya se contempla el término.




## 2.3. Aumento de datos

Para seguir evitando que nuestra red se ajuste demasiado al conjunto de datos de entrenamiento usaremos el 'data augmentacion'.
Esta es una técnica que mejora el entrenamiento modificando de diversas maneras las imágenes de entrada, de forma que se puedan obtener diferentes características de ellas y puedan aportar diferentes informaciones a nuestro modelo. Estas modificaciones no se hacen a todas las imágenes, sino que se realizan solo a algunas aleatoriamente.
Hay muchos tipos de perturbaciones que podemos realizar en este proceso, como podrían ser rotaciones, zooms o traslaciones.
Recalcamos que esto solo se hace 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.

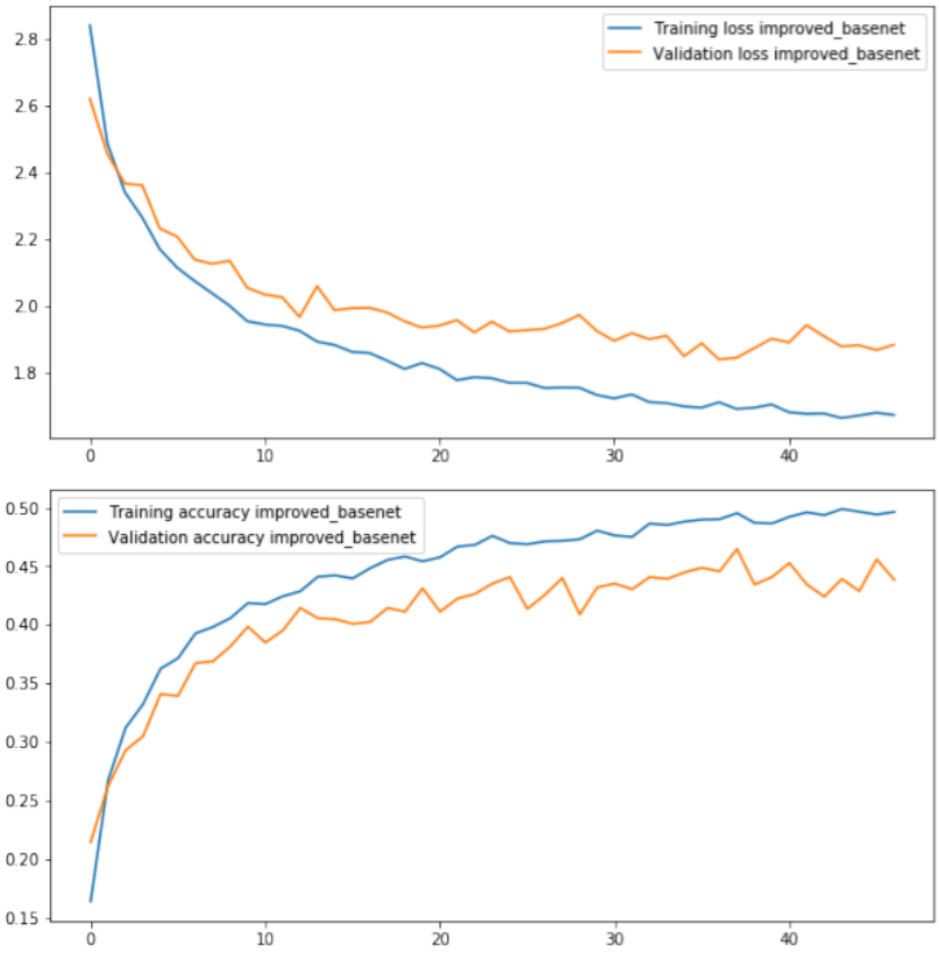
En la implementacion usaremos la clase ImageDataGenerator para crear un datagen de entrenamiento con los parámetros que queremos (contando la normalización inicial que ya teníamos):

```
# Generador para las imágenes con preprocesamiento y data augmentation
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)
```

De las muchas posibilidades hemos usado 4: translaciones horizontales y verticales, volteos horizontales y zoom.
El valor de los mismos indican el grado de intensidad de la transformación.Los parámetros numéricos controlan la "intensidad" de la transformación. Funciona de forma que si definimos el valor de las translaciones horizaontales (width_shift_range = 0.1) se llevará a cabo un aumento aleatorio de esta transformación en el intervalo [1 - 0.1, 1 + 0.1]

Los resultados que obtenemos son los siguientes:
**TEST LOSS=** 1.7209
**TEST ACCURACY=** 0.4813

Si nos fijamos hemos mejorado la precisión en la evaluación, pero lo más importante, hemos conseguido nuestro objetivo de evitar el overfitting.
Podemos verlo en la gráfica como ambas funciones ahora se pegan mucho más a las de entrenamiento.




## 2.4. Aumento profundidad

Para finalizar nuestas mejoras en el modelo aumentaremos la profundidad de nuestra red neuronal añadiendo 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. La distribución quedaria en dos bloques conv-conv-maxpool y dos capas totalmente conectadas antes de la última capa de activación softmax.
El tamaño de las imágenes se mantendrá en la primera convolución de cada bloque mediante el parámetro padding = 'same', para no perder rápidamente dimensión en las imágenes. Adicionalmente, reducimos el tamaño de los kernels de convolución a $3 \times 3$, ya que ahora tenemos dos convoluciones seguidas.
La última opción de mejora será añadir capas Dropout. Estas capas des-

activan 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.

La implementación final sería esta:

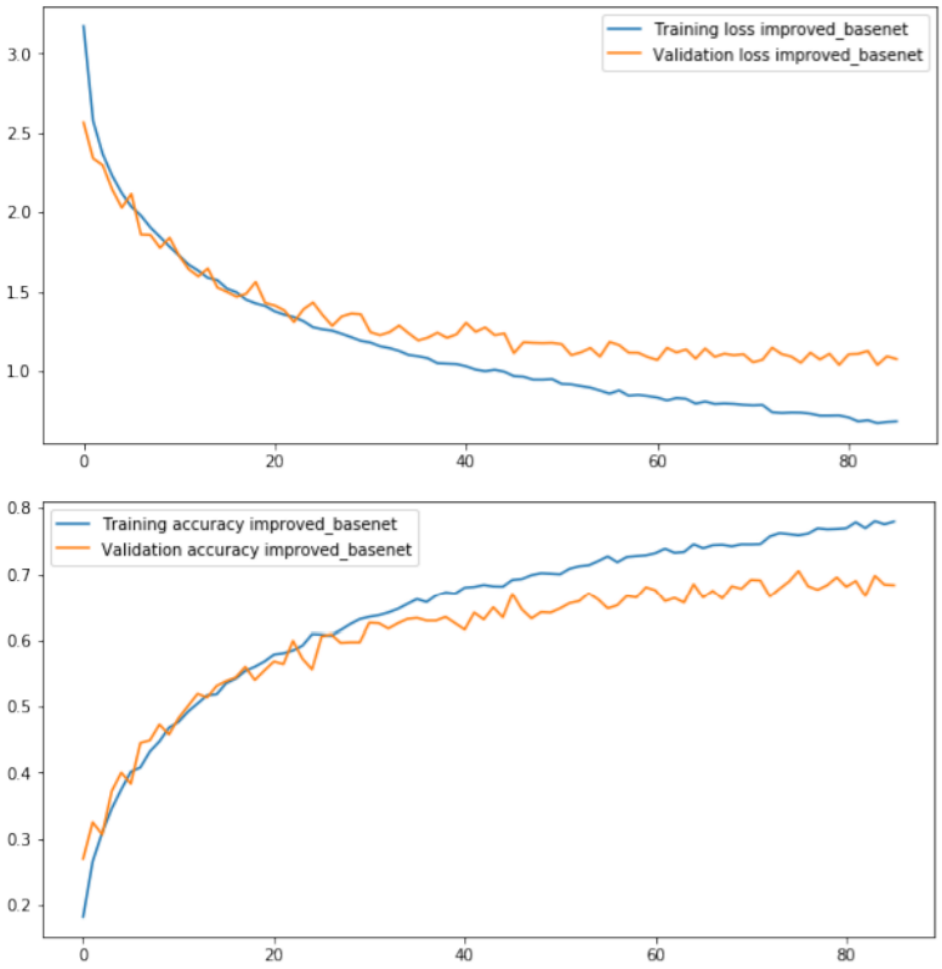
```python
def improved_basenet_model():
    """Devuelve el modelo BaseNet mejorado."""

    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"
```

Y la arquitectura del modelo final mejorado:

| 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 | - |

Tras implementar todas estas mejoras y entrenando de nuevo el modelo durante 84 épocas, hemos obtenido los mejores pesos en la época 74 cuyas estadíticas son: **TEST LOSS=** 0.9153 **TEST ACCURACY=** 0.7239

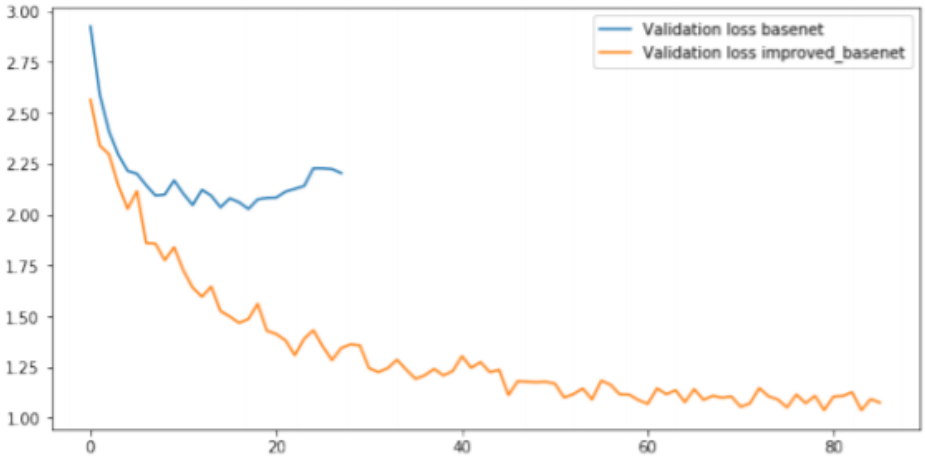
Si analizamos los resultados la función de pérdida la función de pérdida tiene valores menos que antes tanto en entrenamiento como en test.
De igual forma vemos que la precisión ha aumentado en todas las etápas, obteniendo una precisión del 72 % frente al 42 % del modelo inicial en la tarea de clasificación de imágenes de la base da datos.
No podemos negar que las mejoras hechas a nuestro primer modelo han sido correctas y tampoco ha aumentado mucho el tiempo de entrenamiento, unos 10 segundos más por época.

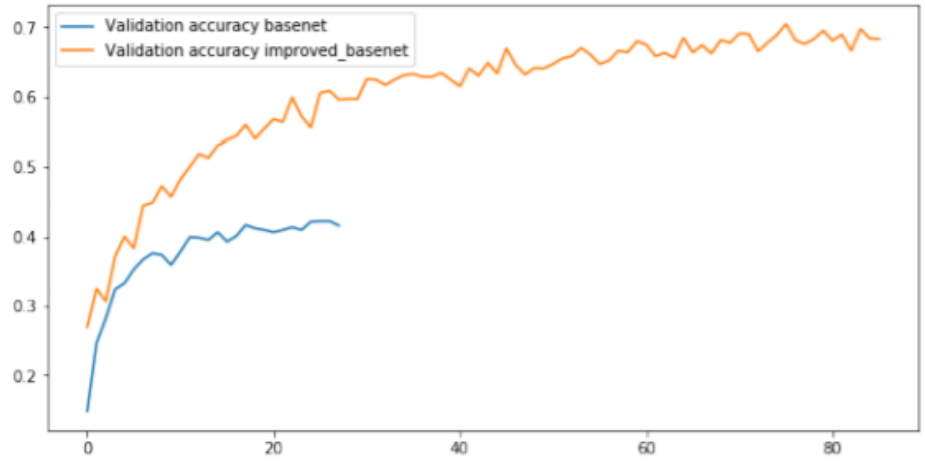



## 2.5. Comparación de modelos

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.

Independientemente de la práctica, ya que no se permite guardar datos en disco, he creado unas funciones auxiliares que añaden un callback de la clase ModelCheckpoint para ir guardando los mejores pesos obtenidos hasta el momento en un fichero. Estas funciones no se ejecutan en la versión final, pero si guardamos las estadísticas de entrenamiento y evaluación de nuestros dos modelos, obtenermos la siguiente gráfica comparativa. 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.


Validation loss basenet
Validation loss improved_basenet
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
20
40
60
80


Validation accuracy basenet
Validation accuracy improved_basenet
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80

# Capítulo 3

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

# Bibliografía


[1] *Convolutional neural networks (convnets).* http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture05.pdf.

[2] *Keras api reference.* https://keras.io/api/.

[3] Haibing Wu, Xiaodong Gu, *Towards dropout training for convolutional neural networks.* https://arxiv.org/ftp/arxiv/papers/1512/1512.00242.pdf.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015), *Deep residual learning for image recognition.* https://arxiv.org/abs/1512.03385.