UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 3: Redes Neuronales Convolucionales

Visión por Computador

Autor:
Lugli, Valentino Glauco · YB0819879

Índice

1. Notas sobre la implementación

- La práctica fue realizada en Python haciendo uso en principio del IDE "Spyder" con CUDA pero por necesidades de cómputo y tiempo, la práctica fue finalizada en Google Colab, de ahí el hecho de entregar una memoria aparte.
- Se ha divdido la práctica en tres ficheros: `p3_ej1_2.ipynb`. `p3_ej3-1.ipynb` y `p3_ej3-2.ipynb` debido a que el ejercicio 3 consume una alta cantidad de memoria, haciendo que se reinicie la máquina virtual de Colab.
- Para el ejercicio 3 se requiere que exista el fichero `imagenes.`**`zip`** en el Drive asociado a la cuenta y que dentro del comprimido se encuentren los archivos de texto `train.txt` y `test.txt` junto con las carpetas que contienen las imágenes de los pájaros, que serán descomprimidos al espacio de la máquina virtual de Colab para que se carguen más rápido.

2. BaseNet en CIFAR100

Para este ejercicio se ha implementado la red neuronal "BaseNet", la cual ha sido entrenada con los datos de CIFAR100 y posteriormente ha sido evaluada.

2.1. Definición del modelo

Para realizar esto se define en primer lugar la función auxiliar `getBaseNet()` la cual no recibe parámetros, interamente se define una red neuronal de forma secuencial utilizando la función `Sequential()` de Keras, a este objeto se le permiten añadir secuencialmente capas para ir construyendo la red neuronal especificada.

La primera capa a añadirse es una capa de convolución 2D, `Conv2D(filters, kernel_size, activation, [padding, input_shape]...)` donde `filters` se refiere a los canales de salida de la capa, `kernel_size` al tamaño del kernel de convolución, `activation` se refiere a la función de activación, el parámetro opcional `padding` indica el tipo de padding que se desea añadir e `input_shape` indica –si la capa es la primera del modelo– la forma que tienen los datos de entrada.

Notar que, si bien en la definición de la arquitectura aparecen las capas ReLU diferenciadas, a efectos prácticos esto es lo mismo que utilizar el parámetro `activation` anteriormente mencionado pues, para lo que concierne esta práctica, siempre se tendrá una función de activación luego de un bloque convolucional o totalmente conectado.

Por lo tanto, la primera capa que se añade al modelo `model` por medio del método `model.add()` se define como `Conv2D(6, 5, activation='relu', input_shape = (32, 32, 3))` se puede interpretar como una capa de convolución de un kernel 5×5, que tiene como entrada un bloque de 32×32 dimensiones espaciales y 3 canales puesto que por defecto Keras tiene el parámetro `data_format="channels_last"`; la función de activación relacionada con esta capa es la reconocida ReLU.

A continuación se incluye una capa `MaxPooling2D(pool_size)`, dónde `pool_size` indica las dimensiones que tendrá la `ventana` que se utilizará para reducir los valores de la capa anterior a uno solo, en el caso del ejercicio se pide `MaxPooling2d(pool_size=(2,2))`, lo que indica que en una ventana de 2×2, se tomará el máximo de esos 4 valores para que sea el siguiente "píxel" en la capa posterior, que en este caso es otra capa de Convolución 2D, esta vez con 16 canales de salida, manteniendo el kernel de convolución 5×5 y la función de activación que nunca ha de faltar para que se realice correctamente el entrenamiento.

Luego de otro `MaxPooling2D`, se aplana la salida con `Flatten()` y se incluyen capas totalmente conectadas o como las define Keras, `Dense(units, activation)` donde el parámetro `units` indica el número de neuronas y nuevamente `activation` indica el tipo de función no lineal a utilizar.

Se añade una capa `Dense(50, activation='relu')`, con 50 neuronas y luego se añade otra con 25 neuronas, puesto que esta es la última capa el número de neuronas debe de coincidir con el número de clases que se desean clasificar, en el caso de este ejercicio se están clasificando 25 clases, un subconjunto del total de clases que posee la base de datos CIFAR100 y también se utiliza en este caso `activation='softmax'` pues es la función de activación de uso para las redes convolucionales que se dedican a la clasificación multiclase.

Una vez creado el modelo, se retorna el objeto que lo contiene para ser utilizado en lo que resta del ejercicio.

Se realizó de esta manera pues es lo que se ha pedido para el ejercicio, y se puede confirmar que el modelo posee la estructura deseada al realizar `baseNet.summary()`, función que imprime las capas junto con sus detalles, las dimensiones con la que está trabajando cada capa y los parámetros que posee, se puede ver en el fragmento de código 1.

```
Model: "initialModel"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)           (None, 28, 28, 6)         456

 max_pooling2d_9 (MaxPooling  (None, 14, 14, 6)         0
 2D)

 conv2d_11 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_10 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten_4 (Flatten)          (None, 400)               0

 dense_16 (Dense)             (None, 50)                20050

 dense_17 (Dense)             (None, 25)                1275

=================================================================
Total params: 24,197
Trainable params: 24,197
Non-trainable params: 0
_________________________________________________________________
```

Listing 1: Visualizando las capas de BaseNet

En efecto puede verse que, esto coincide con los valores que se piden en la arquitectura de BaseNet, por lo que se puede verificar que el procedimiento realizado hasta el momento es el adecuado para el ejercicio.

2.2. Carga de datos y compilación

Una vez definido el modelo, se cargan las imágenes para realizar el entrenamiento: se utilizó la función provista por el profesor, `cargarImagenes()` que obtiene las imágenes de la base de datos CIFAR100, devolviendo 4 listas que fueron nombradas `train_x`, `train_y`, `test_x`, `test_y` indicando las imágenes que se utilizarán para entrenamiento, las etiquetas o clases de esas imágenes, las imágenes de test y sus etiquetas respectivamente.

A continuación, se realiza un pequeño procesado de estas imágenes con la función auxiliar denominada `getDataIterators(train_x, train_y, test_x, batchSize)`, que toma las imágenes de entrenamiento y test, junto a las etiquetas de entrenamiento para obtener un iterador con esos

datos para que luego se utilice para el entrenamiento del modelo como tal, para esto se hace uso de la clase `ImageDataGenerator()` que permite realizar aumento de datos, en este caso en particular no se está realizando aumento de datos para tener un "baseline" de la red sin esto, aún así, este objeto es luego ajustado a las imágenes con el método `fit()` de la misma clase y realizar un conjunto de validación más aleatorio haciendo uso de las función `train_test_split()` para realizar un shuffle de las imágenes en los conjuntos de entrenamiento y validación puesto que por defecto esta clase toma el x % final siempre para validar, haciendo que los valores obtenidos en el conjunto de validación no sean totalmente representativos del grupo total de datos.

Una vez realizado esto, y además normalizando también el conjunto de test por medio del método `standardize()` a los parámetros que poseen los datos de entrenamiento se retornan los iteradores obtenidos de realizar la llamada al procedimiento `flow()` de la clase, la cual genera iteradores de los datos para el entrenamiento.

Se ha realizado esta función para tener más control sobre el grupo de validación y así obtener valores más acorde a la realidad de los datos completos del problema, además de tener una manera estándar en todos los ejercicios para procesar los datos y pasar los mismos parámetros a las funciones pertinentes: reutilizar código.

Una vez se obtienen los iteradores, se procede a compilar el modelo, se utiliza la función auxiliar `compileModel(model)` que realiza la compilación con el método del objeto `model.`**`compile`**`( optimizer, loss, metrics)` dónde `optimizer` indica qué optimizador se utilizará para el descenso de gradiente, como no se especifica alguno en particular, se utiliza el popular algoritmo de optimización Adam pues es adaptativo y funciona por lo general de una buena manera con los parámetros por defecto [1], como función de pérdida `loss` se utiliza entropía cruzada categórica pues es la función de pérdida utilizada en problemas de clasificación multiclase, se utiliza la sentencia `k.losses.categorical_crossentropy` para definirlo, también se indica que la métrica que se desea evaluar mientras se está entrenando y evaluando el modelo: en este caso es la precisión, `accuracy`.

La función no devuelve nada pues los objetos en Python son pasados por referencia a las funciones.

La función se ha realizado para facilitar la compilación pues para el contexto de esta práctica no se vio la necesidad de tener que cambiar los parámetros de la compilación de los modelos, con una excepción que se verá más adelante.

2.3. Entrenamiento y Evaluación

Una vez se tiene el modelo compilado y los iteradores para los datos de entrenamiento y validación se procede a el entrenamiento del modelo, para realizar esto se realizó una función auxiliar denominada `trainModel(model, train_x, val_x, batchSize, trainEpochs)` que recibe un modelo, los iteradores de entrenamiento y validación así como el tamaño del batch y las épocas de entrenamiento, hace uso del método de la clase `Model` denominado `fit()` que puede tomar diferentes formatos de entrada, en este ejercicio se hacen uso de iteradores y del conjunto de validación, por lo que una llamada a dicha función quedaría de la forma `model.fit(train_x, batch_size=batchSize, epochs=trainEpochs, validation_data = val_x, verbose=1)`, se indica `verbose` para poder ver la evolución de la red a medida que se está entrenando.

Una vez entrenada la red, se obtiene de la función el objeto `hist` que contiene internamente una

Entrenamiento	Pérdida	Precisión
1	3.8687	42.6 %
2	4.1393	39.72 %
3	3.6088	41.76 %
Promedio	3.8723	41.36 %

Cuadro 1: Resultados de la evaluación en test de 3 entrenamientos independientes de BaseNet

lista de la precisión y la pérdida del modelo en cada época para los datos de entrenamiento y los de validación, este objeto junto con el objeto que contiene la red, los datos de test y sus etiquetas se pasan a otra función auxiliar denominada `testModel(model, text_x, test_y, hist)` que se utiliza para evaluar la red.

Dentro de la función, para obtener los gráficos con la evolución de la red a través de las épocas, se utiliza la función provista llamada `mostrarEvolucion()` además se utiliza un método de la clase `model` llamado `evaluate()` el cual devuelve una lista de dos entradas que contiene el error o pérdida para los datos de test así como la precisión en los mismos.

Para este ejercicio se ha utilizado un tamaño de batch de 64 y 50 épocas, se eligió el tamaño en 64 para acelerar ligeramente el entrenamiento del modelo, si bien la literatura recomienda 32 como valor por defecto, también indica que 64 es otro valor válido [2] [3] al menos para iniciar, luego estos valores pueden ajustarse más al problema en específico; las épocas se eligieron en función del tiempo, para que fuera una cantidad suficiente grande para lograr ver cómo se comporta la red pero también no muy grande para que el tiempo de entrenamiento no sea excesivo.

Se pueden observar los resultados del entrenamiento en la Figura 1, puede observarse que la curva de entrenamiento sigue creciendo conforme aumentan las décadas en la precisión con los datos de entrenamiento y equivalentemente el error se reduce, mientras que la curva de validación al principio replica este comportamiento, pero llega un momento al rededor de la época 30 que la precisión se estanca y de hecho empieza a bajar, también el error empieza a subir: es un clásico ejemplo de una red que está sobreentrenando los datos que posee, pues se está ajustando muy bien a los datos de entrenamiento y por lo tanto no está generalizando lo que ha aprendido a datos que no ha visto antes, y en efecto, al realizar la función de `evaluate()`, se obtiene que la precisión del modelo en el conjunto de test es de promedio 41.36 % y el error se encuentra promediado en 3.3723, ver Cuadro 1, mientras que en entrenamiento estaba ya sobre el +80 % de precisión.

Notar que para estos ejercicios se realizó la técnica de Hold-Out por ser más rápida aunque se está consciente que para un estudio más profundo sería necesario de una técnica como k-Fold Cross Validation para obtener unos valores más realistas, puesto que Hold-Out al evaluar solo una misma parte de los datos puede suceder que las predicciones sean más optimistas o pesimistas de lo que la realidad podrían ser.

En conclusión, en primer lugar se puede observar que la definición del modelo en Keras es válida pues no ha fallado, se ha podido entrenar normalmente, aunque como se ha mencionado produce sobreentrenamiento, pero esto es de esperarse siendo una red base, ya sea por falta de datos o porque hay muchos parámetros libres y por lo tanto la red está aprendiendo hasta el ruido de los datos de entrenamiento y ese aprendizaje no es generalizable.

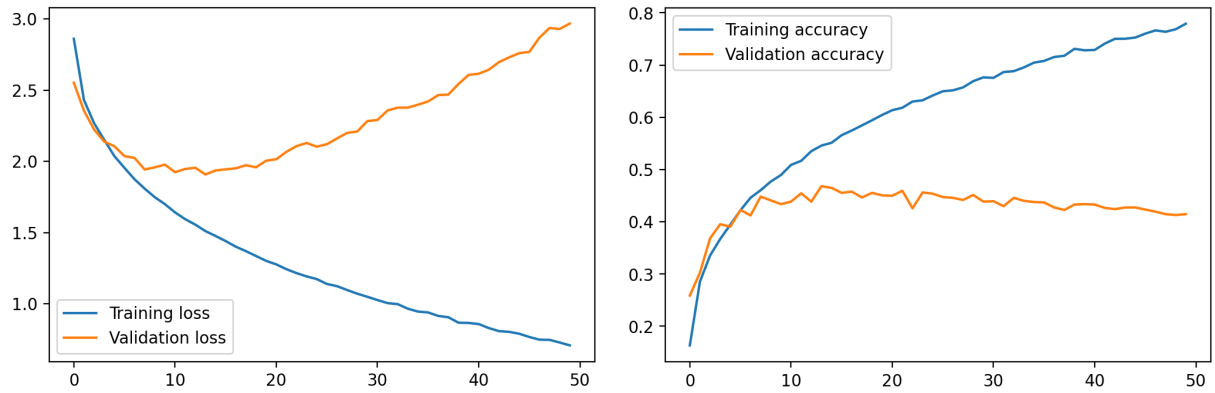

Figura 1: Gráficas de pérdida y precisión en el conjunto de entrenamiento y validación de BaseNet para 50 épocas

3. Mejora del Modelo BaseNet

Para este ejercicio se realizó una mejora de la estructura de la red BaseNet anterior, se define ahora como BaseNet++.

Para realizar esto, se definió una nueva función auxiliar llamada `getBaseNetPlusPlus(fix2, fix3, fix4)` que tiene como parámetros booleanos que controlan los diferentes cambios que se han realidado a la red, esta función devuelve un objeto `Model` con la red definida internamente; específicamente se realizaron los siguientes cambios:

1. **Normalización y Aumento de Datos**: Se hace uso de una función denominada `getNormalizedData(trainImg, trainTag, batchSize, testImg)` que toma las imágenes de entrenamiento, su etiqueta, las imágenes de test y el tamaño del batch para generar iteradores similares a los utilizados en `getDataIterators()` pero ahora se realiza un primer cambio: añade aumento de datos junto con la normalización haciendo uso de `featurewise_center = True` y `featurewise_std_normalization = True`, adicionalmente se ha añadido `rotation_range` con 10º máximo de rotación, `horizontal_flip = True` para darle la vuelta a la imagen en su eje horizontal, `width_shift_range = 0.2` y `height_shift_range = 0.2` para mover ligeramente la imagen de su centro, el resto de la función es idéntica a la anteriormente mencionada.

2. **Aumento de Profundidad**: Se añaden capas convolucionales y totalmente conectadas a la red, en concreto se añadió una nueva capa convolucional pero también se han aumentado la red en anchura: se han expandido la cantidad de canales que utilizan las capas convolucionales, en referencia a las capas totalmente conectadas, se añadieron 2 capas nuevas, con más neuronas que las anteriores capas.

3. **Batch Normalization**: Detrás de cada capa convolucional del nuevo modelo se ha añadido una capa `BatchNormalization()` para que la salida de dicha capa sea normalizada con los valores que posee ese batch de imágenes.

4. **DropOut**: Luego de cada capa totalmente conectada se ha añadido una capa de regularización o DropOut, fijada a un valor de 25 %, este valor se discutirá en breve.

Una vez definida la red con la función, se procede a realizar el procedimiento ya explicado en más detalle en el ejercicio anterior: compilar por medio de `compileModel()`, entrenamiento con `trainModel()` con sus respectivos parámetros y su evaluación con `testModel()`.

La arquitectura final de la nueva red, BaseNet++ queda entonces plasmada en el cuadro 2 y se puede confirmar ello haciendo uso del método `summary()` al igual que con BaseNet, se puede observar la salida de dicho comando se puede apreciar en el fragmento de código 2.

Capa Nº	Tipo de Capa	Tamaño de Kernel	Dimensiones		Canales	
			Input	Output	Input	Output
1	Conv2D + ReLU	(3x3), con `padding='same'`	32	32	3	128
2	BatchNormalization		32	32	128	128
3	Conv2D + ReLU	(3x3)	32	30	128	128
4	BatchNormalization		30	30	128	128
5	MaxPooling2D	(2,2)	30	15	128	128
6	Conv2D + ReLU	(5x5)	15	11	128	64
7	BatchNormalization		11	11	64	64
8	MaxPooling2D + Flatten	(2,2)	11	5	64	64
9	Dense		1600	256		
10	DropOut 0.25					
11	Dense		256	128		
12	DropOut 0.25					
13	Dense		128	64		
14	DropOut 0.25					
16	Dense + SoftMax		64	25		

Cuadro 2: Arquitectura final de BaseNet++

```
Model: "augmentedModel"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 32, 32, 128)       3584
_________________________________________________________________
batch_normalization_3 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 30, 30, 128)       147584
_________________________________________________________________
batch_normalization_4 (Batch (None, 30, 30, 128)       512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 11, 11, 64)        204864
_________________________________________________________________
batch_normalization_5 (Batch (None, 11, 11, 64)        256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
```

```
dense_4 (Dense)              (None, 256)               409856
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_5 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_7 (Dense)              (None, 25)                1625
=================================================================
Total params: 809,945
Trainable params: 809,305
Non-trainable params: 640
_________________________________________________________________
```

Listing 2: Visualizando las capas de BaseNet++

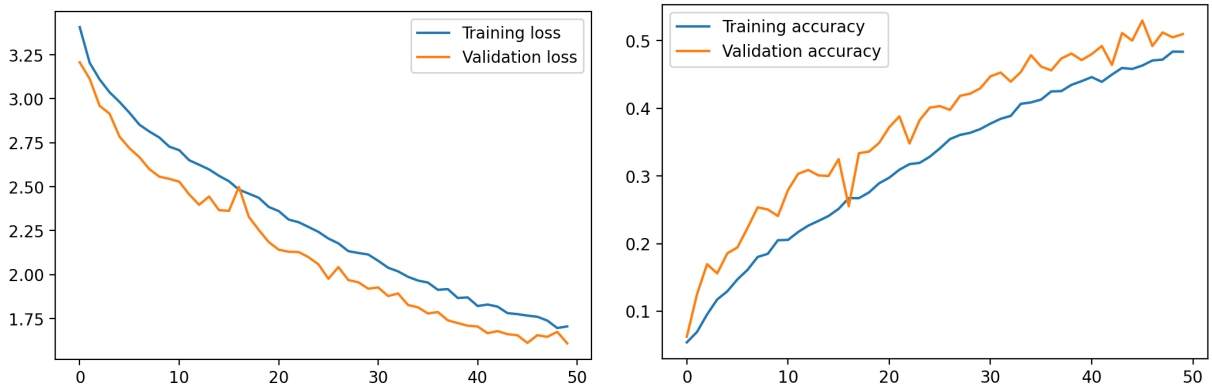

Figura 2: BaseNet++ luego de la primera mejora.

Estas adiciones se han pensado en solventar los problemas que posee la primera red, principalmente que tiene overfitting, una primera aproximación y una que es recomendada es hacer uso del aumento de datos [4] –no solamente la normalización– cuando se tratan de modelos complejos ya que de esta manera la red puede ser entrenada en básicamente más datos de los que se tienen originalmente y no solamente eso: el aumento de datos hace que la red se vuelva invariante a diversas transformaciones, se sabe que de por sí una red convolucional es ligeramente invariante solo a traslaciones por la convolución y el MaxPooling, por lo tanto en el aumento de datos se han añadido cambios de posición más bruscos, además de rotaciones y reflejar la imagen horizontalmente, se desea que la red aprenda características que sean invariantes a esas transformaciones y por lo tanto que generalice mejor la detección, se incluyeron esas transformaciones en particular ya que, por ejemplo, el objeto de la imagen no tiene porque estar siempre con la misma orientación o bien siempre centrado en el medio de la imagen, o con la misma orientación horizontal, también es posible añadir más transformaciones como el cambio de orientaciones verticales o deformaciones más fuertes, aunque ya con lo añadido se ha conseguido un buen resultado.

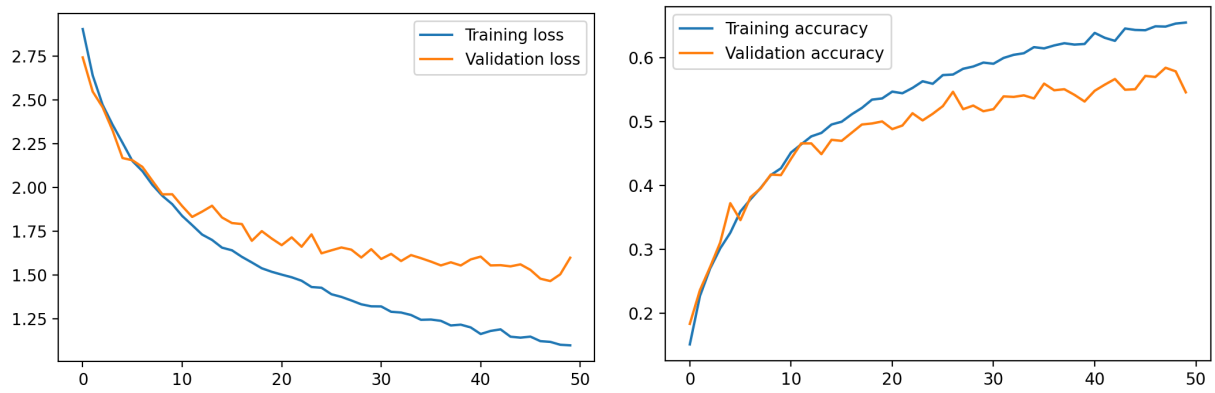

Figura 3: BaseNet++ luego de la segunda mejora.

Aplicando esta mejora, la red pasa de tener aproximadamente 40 % de precisión a tener 48.60 %, la pérdida baja de 3.8723 a 1.7321y además se ha solventado el problema del overfitting de momento, se puede observar en la Figura 2 que ahora la curva del grupo de datos entrenamiento y de validación están más juntas una de la otra en cada época del entrenamiento, indicando que lo que está aprendiendo con el aumento de datos hace que la red pueda generalizar los datos mejor que antes, aún así la precisión está por debajo del 50 % para la cantidad de épocas definida.

Ahora entra en cuestión otro tema; se desea mejorar la calidad de la detección: se sabe por teoría de manera muy clara que –con ciertos límites y circunstancias– tener una red más profunda o ancha, permite clasificar mucho mejor, por lo tanto el siguiente paso a seguir es el de añadir más capas. Como se describió anteriormente, se aumentó en profundidad la red en cuanto a capas y también en anchura, se pasaron de tener de 6 canales en la primera capa de convolución a tener 128 (también se añadió padding para mantener la dimensionalidad espacial de manera que no se redujese muy rápido), la capa convolucional adicional posee también 128 canales y de las capas totalmente conectadas, se pasan a tener 256, 128 y 64 neuronas en cada capa respectivamente.

Aplicando esta mejora sobre la anterior, se obtiene un salto de precisión de 48.60 % y pérdida de 1.7321 a una precisión de 55.32 % y pérdida 1.645, ahora la red logra clasificar las imágenes la mitad del tiempo, que dado el tamaño que posee la red y las relativas pocas épocas que se ha entrenado es una mejora notable, se puede observar el gráfico de su evolución en el entrenamiento en la Figura 3.

Ahora se puede notar que vuelve a incurrir el sobreentrenamiento, en las últimas épocas empiezan a alejarse la curva del grupo de datos de entrenamiento y de validación, esto también es de esperarse debido a que se han añadido muchos más parámetros y por lo tanto, la red es propensa a ajustar datos que no son las más relevantes y por lo tanto generaliza.

Para esto, se añade Batch Normalization que se sabe además que permite reducir el número de épocas en el entrenamiento, posee un efecto regularizador que aún no está totalmente entendido, pero se sabe que funciona: se añade una capa de Batch Normalization luego de cada convolución en el modelo, ahora se pasa de 55.32 % a 64.76 % de precisión y una pérdida que pasa de 1.645 a 1.1583, lo que indica que en efecto la capa de Batch Normalization está mejorando substancialmente la calidad del entrenamiento para la misma cantidad de épocas, se puede observar la curva de aprendizaje en la Figura 4 donde se puede observar que las curvas vuelven a estar muy cerca entre

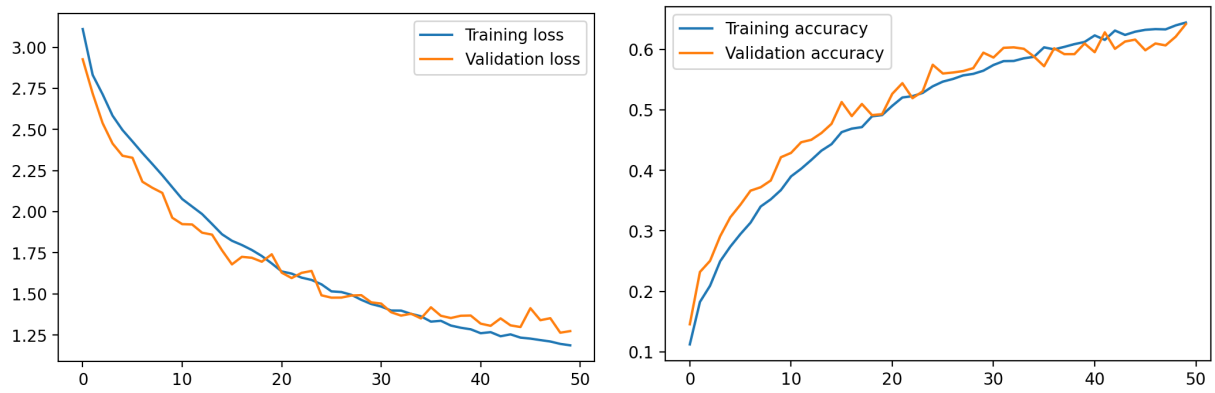

Figura 4: BaseNet++ luego de la tercera mejora.

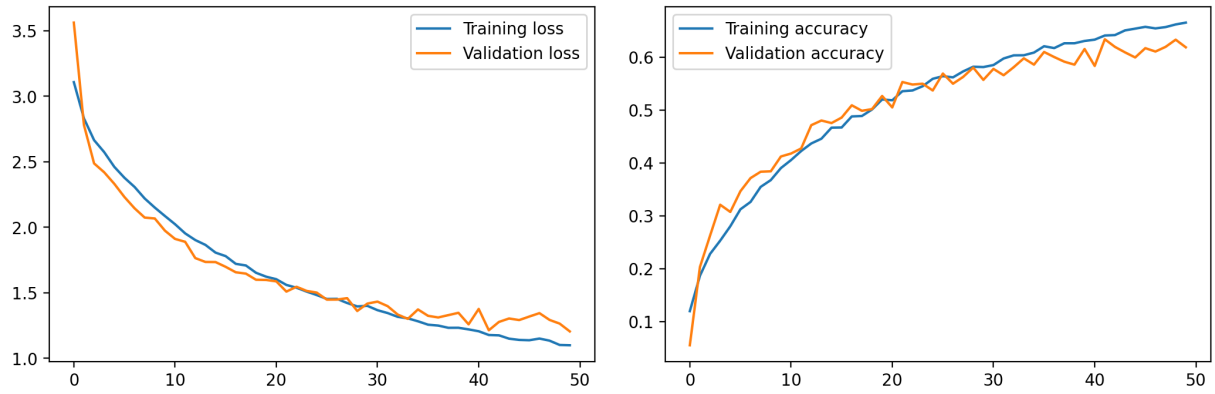

Figura 5: BaseNet++ totalmente mejorada

sí, indicando que lo que lo que se ha aprendido en entrenamiento, también da buenos resultados en la validación.

Como última modificación, se decide añadir también capas de DropOut en las capas totalmente conectadas, con un valor de 25 %, se probó inicialmente con valores de 50 % pero la red tardaba mucho más en entrenar, algo que concuerda con la teoría, puesto que cuando se apagan neuronas en la red, y las otras se ven forzadas a especializarse, esto ralentiza el aprendizaje en general de la red. En todo caso luego del aprendizaje, se obtiene un cambio marginal en precisión y una ligera degradación de la pérdida (64.76 % a 65.28 % y 1.1583 a 1.2123), aunque estos valores están tan cerca que podría concluirse que en esta red con la configuración actual no se ha producido una mejora significativa pues una fluctuación de ese tamaño puede deberse a la propia estocasticidad del método de aprendizaje, aunque no quiere decir que no se pueda aumentar más la calidad: la gráfica aún posee una tendencia creciente, eso quiere decir que si se siguiera entrenando en más épocas sería posible obtener mejores resultados que los actuales, la curva de aprendizaje para la versión definitiva de BaseNet++ con todas las mejoras se puede visualizar en la Figura 5, también, se han resumido en el Cuadro 3 los valores obtenidos de todas las mejoras y la referencia de lo obtenido por la red BaseNet original.

Adicionalmente, para tener un valor más certero respecto a la calidad de la red dentro de lo que cabe haciendo uso de Hold Out, se han realizado tres ejecuciones diferentes del entrenamiento de

Mejora	Pérdida	Precisión
1	1.7321	48.60 %
2	1.645	55.32 %
3	1.1583	64.76 %
4	1.2123	65.28 %
BaseNet	3.8723	41.36 %

Cuadro 3: Mejora en test de BaseNet++ con cada mejora incremental

la red y sus evaluaciones en el conjunto de test, que se pueden visualizar en el Cuadro 4 al igual que se hizo en el ejercicio 1.

Entrenamiento	Pérdida	Precisión
1	1.2123	65.28 %
2	1.0908	68.16 %
3	1.2021	65.32 %
Promedio	1.1684	66.2533 %

Cuadro 4: Resultados de la evaluación en test de 3 entrenamientos independientes de BaseNet++

Se puede confirmar lo que se ha visto en la teoría y en las investigaciones realizadas: que los diversos métodos que existen para la mejora de la calidad de una red funcionan, pero hay que saber cómo aplicarlos y en qué medida, por ejemplo el aumento de datos se debe de realizar pensando que las modificaciones que se están realizando a la imagen deben de no ser muy extremas pero tampoco muy sutiles, porque entonces estaría reduciendo la calidad de la propia red pues o no podría aprender nada ya que las transformaciones son muy extremas o aprendería datos muy similares si son muy sutiles, también la cantidad de canales y la profundidad deben de medirse pues muchos o muy pocos resultan en peores resultados y también en mucho mayor uso de recursos computacionales, también se pudo observar que la inclusión de las capas específicamente para mejoras de la red como Batch Normalization o DropOut realmente mejoran la red en cierta medida aunque no se sepa concretamente porqué lo hacen; esto confirma la idea que la inclusión con conocimiento de más capas o de capas especializadas da como resultado redes de mejor calidad, ejemplos más sofisticados como ResNet o GoogLeNet también dan apoyo a esta idea con sus módulos residuales e Inception respectivamente, que les permiten obtener una precisión muy alta y con un número reducido de parámetros.

4. Transferencia de Aprendizaje y Ajuste Fino con ResNet50 para la BBDD Caltech-UCSD

Para estos ejercicios se pide transferir el aprendizaje de la red neuronal ResNet50 que originalmente ha sido entrenada con ImageNet para que ahora clasifique la base de datos de pájaros de Caltech-UCSD.

Para realizarlo, se han definido diferentes funciones auxiliares para mantener el código lo más

organizado posible; esta nueva red se titula apropiadamente "BirdNet" , y tendrá diversas iteraciones dependiendo de lo que se añada a la red ResNet50 progenitora.

Para poder utilizar el conjunto de datos, se hace uso de la función provista `cargarDatos()` a la que se le indica la ruta donde se encuentra la base de datos junto con dos ficheros de texto indicando que imagen pertenece a que grupo.

Esta función fue modificada para convertir los datos a `float32` para poder hacer mejor uso de la memoria RAM disponible; la función devuelve 4 listas que se denominaron `train_x, train_y, test_x , test_y`, para poder utilizar estos datos con ResNet50 se utiliza la función `preprocess_input( img)` la cual preprocesa la imagen y la ajusta a los valores del dataset de ImageNet, luego estas imágenes son pasadas a una función similar a `getDataIterators()` para obtener los iteradores requeridos por el entrenamiento, para que se puedan reutilizar funciones definidas previamente aunque no se realice aumento de datos.

Lo primero que se realizará es que se tendrá a ResNet50 únicamente como extractora de características, esto es, se tendrá la red con sus pesos congelados y entrenados con ImageNet y utilizando las últimas capas se obtendrán características que serán luego la entrada de otra red neuronal que aprenderá de esas características y clasificará ahora la nueva base de datos, luego se realizará el ajuste fino de ResNet50: se entrenará la red entera sobre la nueva base de datos.

Para realizar esto se definió un función auxiliar denominada `getResNet50(p_pooling, inputShape )` con parámetros que permiten indicar si se desea incluir el pooling o especificar que tipo se desea y cual es la forma de la entrada, internamente la función llama al método `ResNet50(weights = 'imagenet', include_top = False, input_shape = inputShape, pooling = p_pooling)` para obtener un objeto `Model` que contiene la red ResNet50 entera salvo la última capa Densa clasificadora, esta red se devuelve de la función.

Este proceso es similar para todos los ejercicios que se presentarán a continuación.

4.1. Utilizando ResNet50 como Extractor de Características

4.1.1. Removiendo última capa y añadiendo capas densas

Se ha creado una red nueva denominada BirdNetV1, la cual contiene a ResNet50 como extractora de características para una red neuronal totalmente conectada que toma como entrada la última capa de ResNet50 mientras que la propia ResNet tiene sus pesos congelados, es decir, no es entrenable pues no interesa actualizar sus pesos.

Haciendo uso de la función previamente mencionada, `getResNet50()`, se obtiene el objeto con ResNet y ahora este se pasa por parámetro a la función auxiliar denominada `getBirdNetV1(model , onlyFinal)` que acepta tanto un modelo como un booleano que indica si solo se desea la capa clasificadora SoftMax final o una red neuronal densa junto a la capa clasificadora; en este caso se obtiene la red entera pues el parámetro está a `False`.

Para utilizar ResNet50 como extractora, se hace un bucle dentro de la función que va asignando el valor de `trainable` a `False` de cada capa de ResNet, de esta manera los pesos no serán actualizados en el entrenamiento, luego, para conectar la red nueva con esta se obtiene la última capa de la base –ResNet50 en este caso– por medio de `model.output` que se asigna a una variable auxiliar,

en este caso el `output` contiene la capa de `GlobalAveragePooling2D`, a partir de esta capa se irán conectando las capas nuevas.

Para realizar esto se hace `x = (CapaNueva)(x)` donde `x` contiene una capa anterior del modelo, que inicialmente contiene el output de ResNet50: de esta manera se puede construir una red acoplada a ResNet50 pero que la está utilizando únicamente como extractor de características.

Para finalizar la configuración, se define un nuevo objeto `Model` que tiene como entrada la entrada a ResNet50 y la salida la última capa que se ha definido, en definitiva está juntando ResNet50 con las capas adicionales, este objeto se devuelve de la función.

En concreto, las capas que se han acoplado al detector de características se encuentran en el Cuadro 5.

Capa Nº	Tipo de Capa	Tamaño de Kernel	Dimensiones		Canales	
			Input	Output	Input	Output
1	Dense + ReLU		2048	768		
2	Batch Normalization		768	768		
3	Dropout 0.25		768	768		
4	Dense + ReLU		768	512		
5	Dropout 0.25		512	512		
6	Dense + SoftMax		512	200		

Cuadro 5: Capas añadidas a ResNet50 para formar BirdNetV1

La adición particular de estas capas fue elegida de manera heurística, se piensa cómo poder sacarle provecho a ResNet50 entera: teniendo varias capas para que se pueda realizar un aprendizaje sin exagerar en la profundidad y con una cantidad de neuronas que se van disminuyendo gradualmente hasta llegar a las 200 necesarias para clasificar los pájaros para ir forzando a la red a seleccionar la información más importante para realizar la clasificación.

Una vez obtenido el modelo, se compila con la función `compileModel()` y luego se entrena con la función `trainModel()` que ha sido modificada para estos ejercicios, se está utilizando ahora un `callback`: un objeto de Keras que realiza acciones en lo que se está entrenando la red, en este caso se definió un callback que llama al recolector de basura de Python, `gb.collect()` al final de cada época de entrenamiento por medio de una clase denominada `GarbageCollectorCallback()`.

Se puede observar que efectivamente la se están añadiendo las capas adicionales si se realiza la llamada a la función `summary()`, en el fragmento de código 3 se puede ver que en efecto se han añadido las capas justo después del `GlobalAveragePooling2D` y que ResNet50 como tal no es entrenable, evidenciado por los 23 millones de parámetros que no lo son, solamente se entrenarán las adiciones hechas por el autor.

```
avg_pool (GlobalAveragePooling  (None, 2048)        0       ['conv5_block3_out
   [0][0]']
2D)

dense (Dense)                   (None, 768)         1573632 ['avg_pool[0][0]']

batch_normalization (BatchNorm  (None, 768)         3072    ['dense[0][0]']
```

```
 alization)

 dropout (Dropout)              (None, 768)          0       ['batch_normalization
    [0][0]']

 dense_1 (Dense)                (None, 512)          393728  ['dropout[0][0]']

 dropout_1 (Dropout)            (None, 512)          0       ['dense_1[0][0]']

 dense_2 (Dense)                (None, 200)          102600  ['dropout_1[0][0]']

==========================================
Total params: 25,660,744
Trainable params: 2,071,496
Non-trainable params: 23,589,248
__________________________________________
```

Listing 3: Visualizando las capas de BirdNetV1

Una vez realizado esto, se realiza el procedimiento conocido: se compila el modelo con la función `compileModel()` y luego se entrena utilizando las funciones previamente definidas.

Para este entrenamiento se utilizaron batches de 64 y 15 épocas de entrenamiento puesto que es una red particularmente grande, si bien ResNet50 no está siendo entrenada, las transformaciones que le suceden a los datos tiene un overhead notable por la cantidad que se tienen de los mismos, además, realizando varias pruebas se pudo observar que la red tal y como está empieza a sobreentrenar muy rápido y por lo tanto, tampoco son necesarias tantas épocas.

Las gráficas de aprendizaje se pueden observar en la Figura 6, también se obtuvo una precisión de 40.488 % y una pérdida de 2.5046: es fácil notar que esta red está sufriendo de overfitting, lo cual se debe probablemente a la cantidad tan grande de parámetros a entrenar en principio; incluso utilizando DropOut no fue posible taclear el problema y como se trata de una primera aproximación, la falta de aumento de datos también afecta la calidad del aprendizaje, no se desea extender más la red porque es posible que se degraden más los valores; aún así queda demostrado que es posible utilizar una red como extractor de características de otra red, si bien en este caso debería de realizarse un estudio más en profundidad para poder resolver la causa del overfitting demostrado.

4.1.2. Removiendo última capa y añadiendo solo el clasificador

Para la realización de este apartado, se hace uso de la misma función `getBirdNetV1()` solo que en esta ocasión se pasa el segundo parámetro a `True` para que solamente se incluya la última capa en RestNet50, a este modelo se le denomina BirdNetV0 puesto es más simple que el modelo anterior al solamente tener una única capa que realiza la clasificación que le da como salida ResNet50, el procedimiento es idéntico al anterior, utilizando los mismos conjuntos de datos y los mismos parámetros para el número de batches y épocas para poder compararlos entre si, la arquitectura resultante, tomando en cuenta como entrada a la red la salida de la última capa de ResNet50 se puede observar en el Cuadro 6.

Una vez que se ha entrenado, se obtienen las curvas de aprendizaje que se pueden ver en la Figura 7, se obtuvo 2.321 de pérdida y un 42.5651 % de precisión.

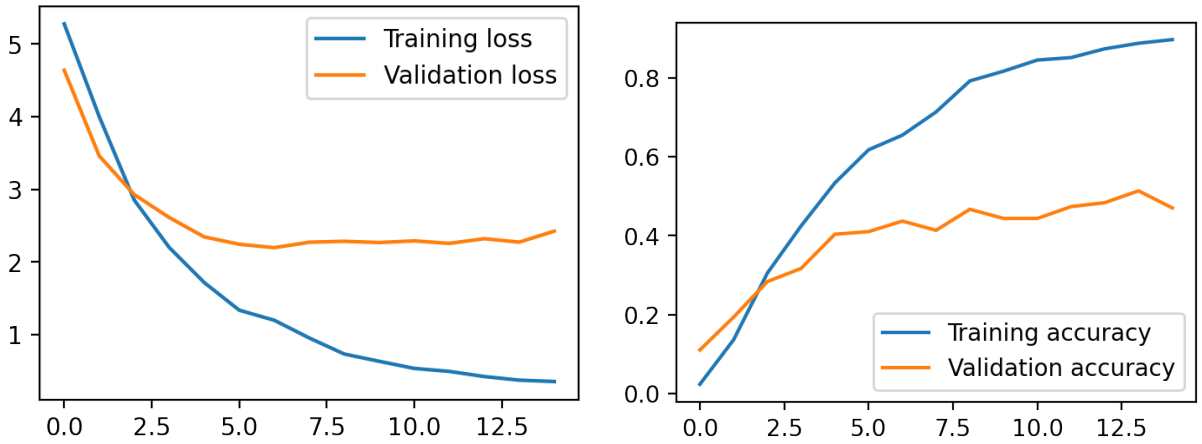

Figura 6: Aprendizaje de la red BirdNetV1

Capa Nº	Tipo de Capa	Tamaño de Kernel	Dimensiones		Canales	
			Input	Output	Input	Output
1	Dense + SoftMax		2048	200		

Cuadro 6: Capa añadida a ResNet50 para formar BirdNetV0

Si bien es cierto que tanto la precisión como la pérdida se encuentran en valores similares, si se observan las curvas de entrenamiento puede verse que la curva de entrenamiento de BirdNetV0 (Figura 7) crece mucho más rápido hacía valores altos que en BirdNetV1 (Figura 6) indicando que, si bien ambos modelos están sufriendo de overfitting, las capas adicionales que se añadieron en BirdNetV1 están haciendo un efecto en el entrenamiento, aunque sea ligero, esto da como conclusión que este modelo tiene más posibilidades de poder ajustarse mejor a los datos realizando un estudio del mismo, pues se tiene la libertad de experimentar con diferentes capas, mientras que BirdNetV0 como solamente puede entrenarse una única capa final, no se tiene la misma libertad para ajustar el aprendizaje.

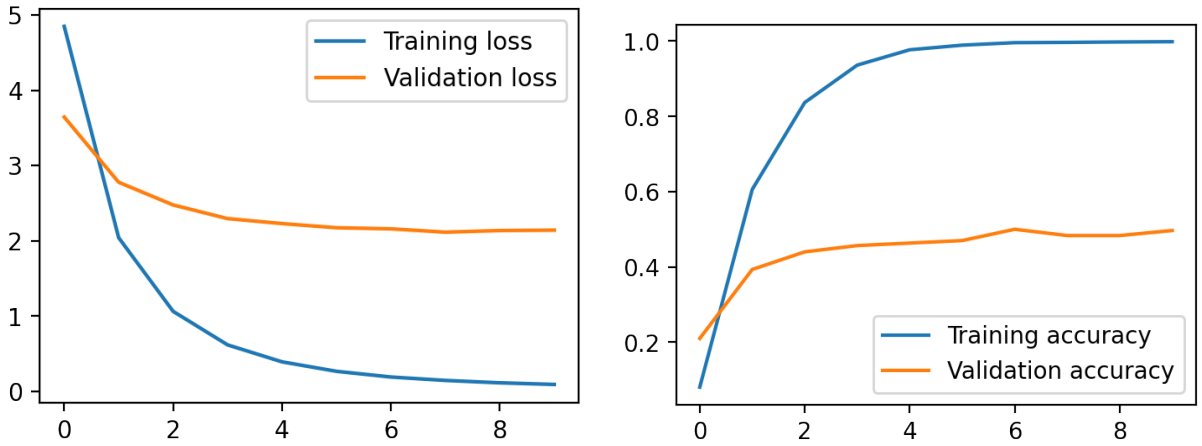

Figura 7: Aprendizaje de la red BirdNetV0

Capa Nº	Tipo de Capa	Tamaño de Kernel	Dimensiones		Canales	
			Input	Output	Input	Output
1	Conv2D + ReLU	(3x3), con padding='same'	7	7	2048	64
2	BatchNormalization		7	7	64	64
3	Dense				3136	256
4	Dense				256	200

Cuadro 7: Aprendizaje de la red BirdNetV2

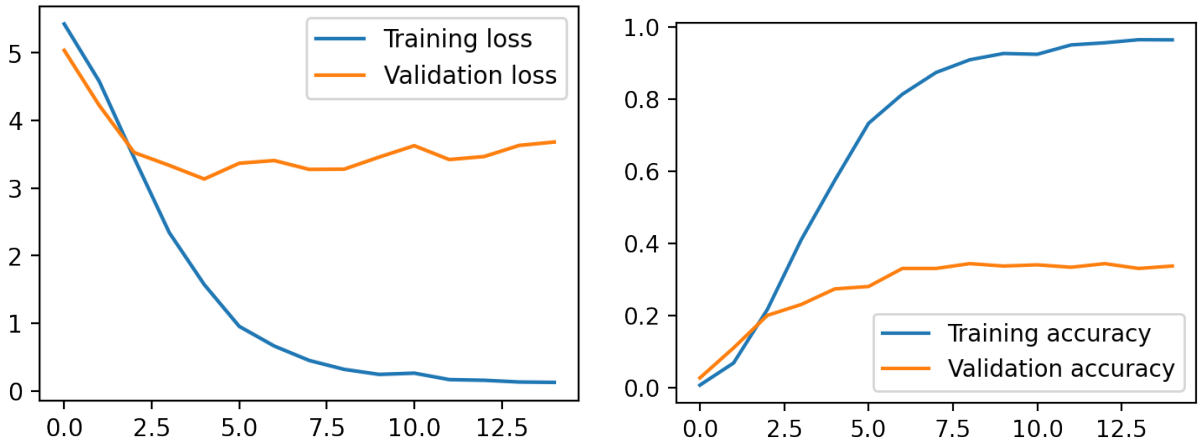

Figura 8: Aprendizaje de la red BirdNetV2

4.1.3. Removiendo últimas capa, añadiendo capas convolucionales y densas

Para este se ejercicio se repite el proceso anterior, se tiene una función auxiliar denominada `getBirdNetV2()` la cual se le pasa el objeto que contiene a ResNet50 obtenido de `getResNet50()` pero esta vez sin el `GlobalAveragePooling2d` para que se tenga contacto directo con las capas convolucionales de la red, nuevamente se marcan las capas originales de ResNet50 como no entrenables y se añaden las capas descritas en el Cuadro 7.

Una vez realizado esto, se realiza el mismo procedimiento de compilación, entrenamiento y test; la gráfica de resultados puede verse en la Figura 8, esta vez se optó por no añadir muchas capas pues se tiene el problema del sobreentrenamiento, aún así se puede notar que se obtuvo un valor de 3.2539 de pérdida y de 29.6076 % de precisión, valores que son menores a los obtenidos en los apartados anteriores, el hecho de añadir una capa convolucional no parece haber hecho un gran cambio positivo, nuevamente se puede pensar que la cantidad de parámetros para ajustar es muy alto para la red y no se están considerando otros parámetros que serían posibles estudiar para poder mejorar la calidad de la red.

Se puede observar que, tal y como están estas aproximaciones para la transferencia de conocimiento no están funcionando del todo bien, de hecho ninguna logra superar el nivel "base" puesto por la red que tiene una única capa clasificadora (esto puede verse en el Cuadro 8 que resume los datos obtenidos), por una parte esto sucede por la falta de data augmentation y la falta de un estudio más profundo de las redes: por ejemplo, además de modificar la red en sí, se podría estudiar realizar cambios en los hiperparámetros como el learning rate del optimizador ya que en este caso se ha

Modelo	Pérdida	Precisión
BirdNetV0	2.321	42.5651 %
BirdNetV1	2.5046	40.488 %
BirdNetV2	3.2539	29.6076 %

Cuadro 8: Resultados de las redes BirdNetV0 a V2

estado utilizando Adam con los parámetros por defecto.

Además, las limitaciones sobre el hardware impidieron más pruebas para poder mejorar los resultados, aún así, se sabe por las investigaciones[5] y la teoría que la transferencia de aprendizaje sobretodo de redes que se han entrenado en dominios más generales a dominios más específicos funciona siempre y cuando estos dominios compartan similitudes, no tendría sentido transferir el conocimiento de ResNet sobre ImageNet a una red que pretende clasificar firmas o células cancerosas, aunque esto no parece ser el caso en el contexto actual, también otra posibilidad es que lo ideal sería utilizar otra técnica de aprendizaje para tratar con los datos de salida, por ejemplo, haciendo uso de Support Vector Machines o K-Means.

4.2. Ajuste fino de ResNet50

Para realizar el ajuste fino de ResNet50 sobre los datos de Caltech-UCSD se define una última función auxiliar denominada `getBirdNetV3(model)` que retorna una red ResNet50 llamada BirdNetV3 pero a diferencia de los anteriores ejercicios, ahora todos sus parámetros son entrenables, se añade únicamente el clasificador nuevo pues lo pedido es realizar un ajuste entero de la red, no de una parte de la red como tal.

Se hace uso de las funciones previamente mencionadas para compilar y entrenar la red, al igual que antes con 15 épocas y batches de 32 imágenes.

Las curvas de aprendizaje se pueden visualizar en la Figura 9, se obtuvo como resutados de evaluación en test una pérdida de 4.7275 y una precisión del 21.299 %.

De esta figura también se puede ver como la pérdida de la red comienza extremadamente alta y luego se desploma, esto evidencia que el learning rate por defecto está haciendo saltos demasiado grandes y se está estancando en mínimos locales.

Se puede observar que, las conclusiones descritas anteriormente siguen siendo verdad, el modelo con todos los pesos sigue sobreentrenando y de hecho, se degrada la calidad del mismo, al realizar investigaciones se confirma que no es recomendado reentrenar la red entera puesto que entonces los pesos que poseía esta red, que en este caso se trata pesos entrenados sobre ImageNet, están siendo ahora sobreescritos con nuevos datos y por lo tanto se está básicamente entrenando la red desde cero, además que se pierde la esencia de la transferencia de conocimiento: también se modifican los filtros más básicos al inicio de la red que están entrenados para detectar las características a bajo nivel que se supone son los que se quieren aprovechar que ya están entrenados para construir a partir de ellos filtros especializados en el problema a tratar.

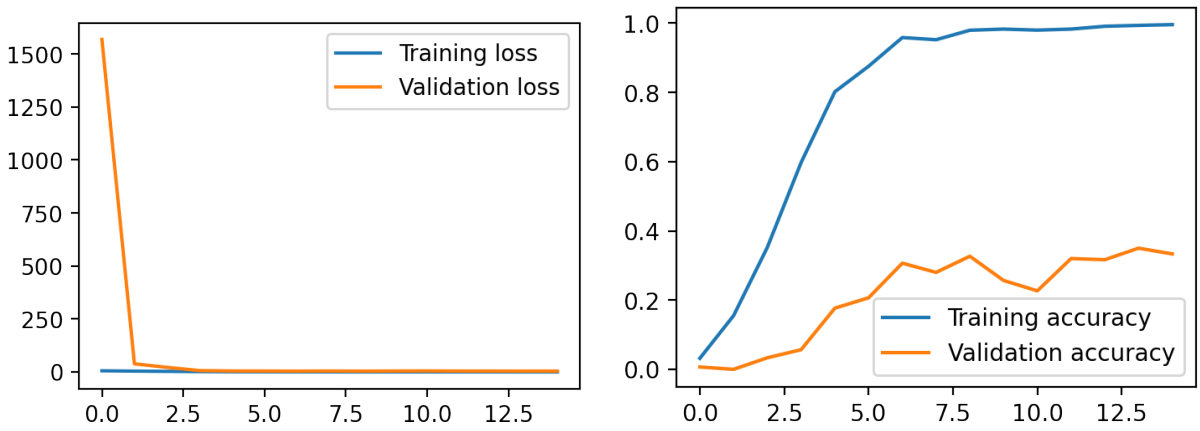

Figura 9: Aprendizaje de la red BirdNetV3

Capa Nº	Tipo de Capa	Tamaño de Kernel	Dimensiones		Canales	
			Input	Output	Input	Output
1	Dense				2048	1024
2	DropOut 0.5				1024	1024
3	Dense				3136	512
4	Dense				512	200

Cuadro 9: Arquitectura de BirdNetV5

4.3. Mejorando el Ejercicio

Luego de una investigación a fondo, se determinó que para realizar un ajuste fino se debe de, efectivamente, no entrenar la red entera, más bien una parte junto con adiciones hechas a la red.

Se parte igualmente obteniendo el modelo de ResNet50 sin la última capa pero con el Pooling, se tiene la función `getBirdNetV4()` que añade a la red un par de capas totalmente conectadas, se puede obsevar su arquitectura en el Cuadro 9; se realizó de esta manera pues por la experiencia anterior, la red de por sí realiza mucho sobreentrenamiento por lo tanto una red muy grande no se pensó fuera lo recomendable, pero también se quiere tener un poco de control en las últimas capas, por lo tanto se decidió por una red intermedia que al menos posee una capa de DropOut para controlar el aprendizaje en cierta manera.

Una vez se obtiene la red, el proceso es similar a los anteriores, se congelan nuevamente los pesos originales de la red, y se procede a compilar con una diferencia: se modifica el *Learning Rate* del optimizador Adam, algo que hasta ahora no se había considerado pero luego de la investigación se ha observado que es recomendado ajustar este parámetro y ajustarlo a un valor bastante pequeño [6] y de hecho este valor puede ser ajustado para cada red dependiendo de los datos pero por defecto se mencionan valores entre $1 \times 10^{-}4$ hasta 1×10^{6}, la lógica detrás de esto es que se quiere que la gradiente avance lentamente pues no se quiere que los pesos sean actualizados con mucha fuerza pues si esto sucede los pesos podrían no generalizar bien y luego la red pierde calidad, por lo tanto, se realiza la compilación del modelo con el objeto `tf.keras.optimizers.Adam(learning_rate=1e-4)` para ajustar el parámetro del algoritmo Adam.

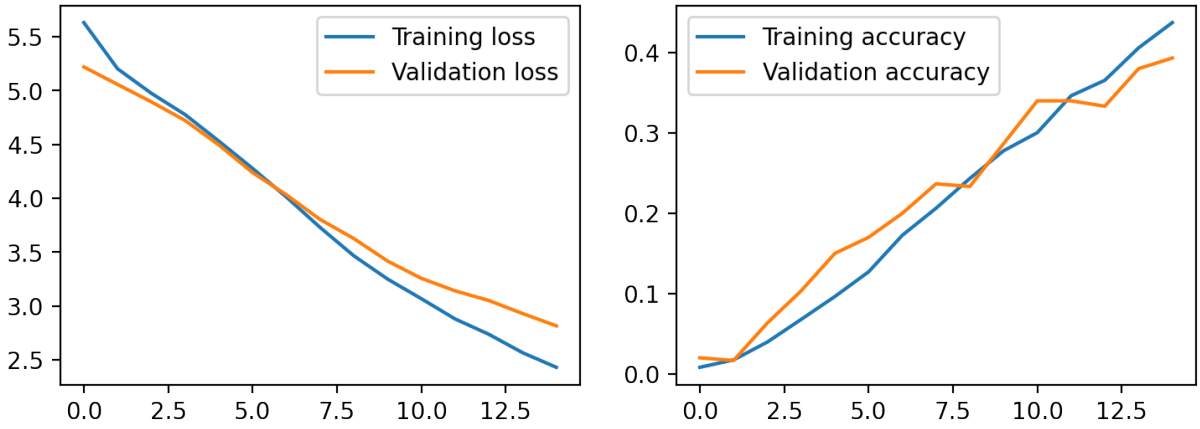

Figura 10: Aprendizaje de la red BirdNetV4 con capas de ResNet50 congeladas

Se realiza ahora un entrenamiento que se denomina "Warm-Up" que tiene la idea de darle valor a los pesos de las capas nuevas dejando las capas originales intactas, esto era en parte la idea que se utilizó en las partes anteriores de los ejercicios pero aquí no termina el entrenamiento de la red; solamente se desea que los pesos se entrenen hasta que no se empiece a generar overfitting, también, se ha utilizado una función similar a la utilizada en el ejercicio 2 para aumento de datos, se realizó en esta red pues se ha logrado una mejora notable respecto a las redes anteriores, en las cuales se probó pero no se obtuvo una mejora significativa de calidad, también surgió el problema del uso de memoria RAM al ejecutar los ejercicios por lo tanto se prefirió realizar un entrenamiento más básico, ahora se deseó realizar un entrenamiento más completo, pero estas conclusiones para las redes anteriores, pues la recomendación de utilizar un learning rate bajo también es recomendada para la transferencia de aprendizaje, de hecho, también se habla de tener diferentes learning rates para diferentes capas de la red pues en ciertos momentos es deseable ir más rápido y en otros ir más lento como se ha podido evidenciar en este ejercicio.

Una vez se realiza esto ahora si lo que cambia es que se ponen a que se puedan entrenar –se "descongelan"– los pesos de la red, y no *todos* sino *algunos*, específicamente los últimos pesos son los que se desean descongelar pues son los que poseen características más complejas luego son los que se desean sean ligeramente alterados para adaptarse a los nuevos datos mientras que los pesos iniciales, aquellos detectores de nivel bajo no interesa tocarlos pues ya poseen valores útiles, además que ya se sabe que la idea de la transferencia de aprendizaje es que ciertas capas son tan bajo nivel, tan básicas, que para muchos problemas se pueden utilizar las mismas pues al final si lo que detectan son blobs o esquinas, esto es algo que necesitan muchos problemas y dominios diferentes, lo importante son las capas más avanzadas donde cada red empieza a diferenciarse una de otra.

En este caso, se decidió por descongelar las últimas capas del modelo: aquellas donde el bloque convolucional es de tamaño 14×14 pues se piensa que a ese nivel las características empiezan a ser más avanzadas y por lo tanto, se desean se adapten a los nuevos datos.

Se puede observar en la Figura 10 el aprendizaje realizado con los pesos congelados, se puede ver que ahora las gráficas de aprendizaje y entrenamiento están de la mano, pero también se sabe que esta red, que es muy similar a las del ejercicio pasado, empieza a sobrenetrenar a partir del 40 %

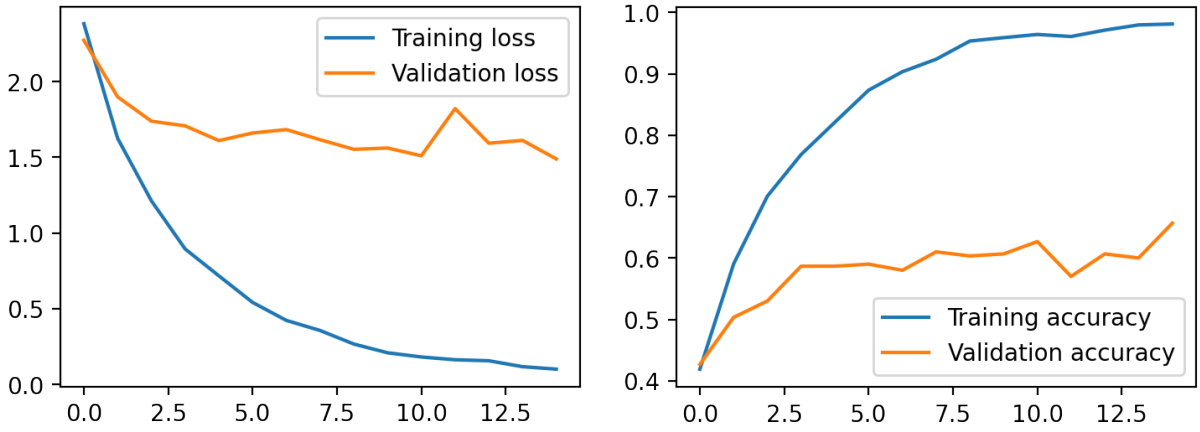

Figura 11: Aprendizaje de la red BirdNetV4 luego de descongelar capas de ResNet50

de precisión, por lo tanto, allí se detiene el entrenamiento, el Warm-Up.

También notar que en este ejercicio, a diferencia de los anteriores, se ha utilizado Data Augmentation, se ha preferido utilizarlo aquí para demostrar que esto también puede mejorar muchísimo la red, no se pudo incluir en los anteriores por el uso de memoria que esto produce.

Luego, se vuelve a compilar la red para que pueda volver a ser entrenada con el uso de nuevamente un *Learning Rate* bajo para no incurrir dentro de lo posible en overfitting.

Como puede observarse en las Figuras 11, la red ahora está clasificando aproximadamente un 20 % mejor los datos que cualquiera de las redes probadas anteriormente, confirmando que los consejos investigados afirmaban, si bien la red empieza a incurrir en sobreajuste al final, ya esto puede solventarse utilizando las técnicas vistas anteriormente, y también utilizando las técnicas nuevas: ajustar el optimizador, ajustar cuales capas se descongelan y cuales no, que permitirán entonces mejorar la calidad de la red; concretamente se obtiene al rededor de un 2.1333 de error de pérdida y 50.1813 % de precisión.

Esto en conclusión da soporte al ajuste fino: que mantener las capas iniciales e ir modificando ligeramente los pesos al final realmente logra que se puedan adaptar redes inicialmente entrenadas en una clase de imágenes ser utilizada para otra, confirma lo visto en teoría sobre las características que se ha aprenden en cada nivel de la red, las capas iniciales de una red aprenden las características más básicas –lo que serían detectores de esquinas y blobs– y las siguientes capas van construyendo sus filtros más especializados a partir de filtros anteriores, entonces tiene sentido que las primeras capas de las redes sean todas muy similares y luego las capas últimas sí sean específicas para cada problema a tratar, entonces se pueden tomar las capas iniciales de una red entrenada en una clase de problema y ser utilizada para otro problema que tenga un dominio similar, como se ha mencionado anteriormente; aquí se puede ver que adaptar de un dominio más general de ImageNet a un dominio más específico pero similar como serían imágenes de pájaros funciona, esta lógica también se puede aplicar a la transferencia de aprendizaje pues la idea básica se mantiene, una red entrenada para un problema más general puede generar características útiles en problemas más específicos que compartan similitudes, lo que las diferencia es que en ajuste fino se tocan los pesos de la red original para adaptarlos mientras que en transferencia de aprendizaje ya los pesos están congelados, que posiblemente para ciertos problemas sea suficiente, además que estas características también

pueden ser utilizadas por otros modelos de aprendizaje diferentes a perceptrones multicapa que no requieren de ajustar los filtros de la red original, sino que se desea obtener otra clase de información que no sea necesariamente la clasificación para la cual se entrenó la red originalmente, por ejemplo, para el cálculo de las bounding boxes en los problemas de detección, se utilizan las características de una red para realizar un problema de regresión mientras con esas mismas características también se realiza la clasificación del objeto dentro del bounding box.

También se han notado los retos y problemas que presenta esta técnica, pues se han incurrido en ellos en el desarrollo del ejercicio a mucha frustración del autor, aún así, los beneficios de esta técnica son también enormes, permite un aprendizaje más rápido pues una gran parte del esfuerzo ya ha sido realizado por otras personas que por ejemplo tienen acceso a más recursos y también porque "no es necesario reintentar la rueda siempre", si ya se tiene una red que funciona muy bien para clasificar, mejor empezar de eso que desde cero, aunque también si el problema es muy particular sigue siendo también una opción rentable construir una red desde cero si se poseen los recursos de cómputo necesarios y los datos.

Referencias

[1] S. Bera y V. K. Shrivastava, "Analysis of various optimizers on deep convolutional neural network model in the application of hyperspectral remote sensing image classification," *International Journal of Remote Sensing*, vol. 41, n.º 7, págs. 2664-2683, 2020. DOI: 10.1080/01431161.2019.1694725. eprint: https://doi.org/10.1080/01431161.2019.1694725. dirección: https://doi.org/10.1080/01431161.2019.1694725.

[2] I. Kandel y M. Castelli, "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset," *ICT Express*, vol. 6, n.º 4, págs. 312-315, 2020, ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2020.04.010. dirección: https://www.sciencedirect.com/science/article/pii/S2405959519303455.

[3] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures*, 2012. arXiv: 1206.5533 [cs.LG].

[4] X. Ying, "An Overview of Overfitting and its Solutions," *Journal of Physics: Conference Series*, vol. 1168, pág. 022 022, feb. de 2019. DOI: 10.1088/1742-6596/1168/2/022022. dirección: https://doi.org/10.1088/1742-6596/1168/2/022022.

[5] S. J. Pan y Q. Yang, "A Survey on Transfer Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, págs. 1345-1359, 2010.

[6] H. Li, P. Chaudhari, H. Yang y col., *Rethinking the Hyperparameters for Fine-tuning*, 2020. arXiv: 2002.11770 [cs.CV].