Práctica 1

Visión por Computador

Grupo de Prácticas 2

Juan José Herrera Aranda - 26516011-R
juanjoha@correo.ugr.es

24 de diciembre de 2021

Índice

1. Aclaraciones y comentarios sobre la práctica. 3

2. Ejercicio 1 4

3. Ejercicio 2 + Bonus 7
3.1. Versión 1 7
3.2. Versión 2 11
3.3. Versión 3 - BONUS 14

4. Ejercicio 3 19
4.1. Apartado 1 19
4.2. Apartado 2 24
4.3. Comparación 25

5. Bibliografía 28

1. Aclaraciones y comentarios sobre la práctica.

Como aclaraciones a la implementación del código decir que:

- El ejercicio 1 y el 2 tienen cada uno una función **cross-validation** debido a que en el primer ejercicio la centramos exclusivamente para la experimentación de los distintos optimizadores mientras que en el ejercicio 2, al añadir las mejoras, usamos aumento de datos y calculamos además del accuracy, la función de pérdida. Aún así es cierto que se podría haber implementado todo en una función en vez de dos. Lo mismo ocurre con la función **fit_evaluate**. Con esto quiero decir, que aunque se llamen igual y realicen la misma función, el cómo la realizan es distinto, cada una orientada al ejercicio correspondiente.

- El vector que llamo **v_xy_train_test** es un vector de cuatro componentes, en las dos primeras posiciones van los datos de entrenamiento, mientras que en las dos últimas van los datos de test. En la función **cross-validatinon** del ejercicio 2, aunque la llamo con el vector anterior, únicamente trabajo con la dos primeras componentes para hacer el proceso de experimentación.

- Las limitaciones en RAM y Disco de Google Colab ha hecho que por comodidad se usen dos ficheros notebooks separados. El primero corresponde a los ejercicios 1, 2 y el bonus y el segundo al ejercicio 3.

- El ejercicio 1 se ha tratado de forma independiente al resto. Es decir, hemos entrenado con los datos correspondientes al entrenamiento y evaluado con los datos de test, puesto que no hemos hecho procedimiento experimental. En el ejercicio 2 partimos del modelo del ejercicio 1 y el objetivo es mejorarlo para que generalice mejor que el 1. Así que, para no cometer DataSnooping, establecemos un procedimiento experimental sin usar los datos de test y en donde incrementalmente vamos construyendo versiones sobre el modelo para finalmente escoger una que tras evaluarla, (ahora sí) con los datos de test, comprobar si mejora o no al ejercicio 1. Posteriormente en el Bonus, hacemos dos versiones y tomamos una de ellas para evaluarla con los datos de test y comprobar de nuevo si mejora o no. Observando los resultados experimentales se puede intuir de manera obvia si mejoran o no los resultados.

- Varias ejecuciones de un modelo no dan los mismos resultados, por tanto puede que difieran los resultados de la memoria con respecto a los resultados cuando usted ejecuta la práctica, debido a que la partición de los datos es aleatoria, entre otros factores.

- El enunciado del ejercicio 3 expone que usemos un 10 % de los datos de entrenamiento para validación, pero no queda claro si se refiere a validación en el entrenamiento (usando la función fit con validation_split=0.1) o partir el conjunto de entrenamiento en 90-10 y usar ese 10 % como si fuera test. En cualquier caso, hemos optado por otra alternativa detallada más adelante.

- **IMPORTANTE:** En el ejercicio 3, debido a la gran espera en tiempo que conlleva cargar los datos, se han guardaron las matrices en binario en formato **npy** para luego cargarlas con mayor facilidad.

- En el ejercicio 3 no se ha empleado cross-validation por problemas de cómputo, tiempos largos y falta de memoria RAM, por lo que los resultados son poco representativos ya que dependen de la aleatoriedad de la partición

2. Ejercicio 1

En este apartado implementaremos un modelo al que llamaremos BaseNet, cuya arquitectura viene definida por la figura 3. Lo entrenaremos con un famoso conjunto de datos llamado CIFAR100 disponible en la base de datos de Keras. No trabaremos con todos los datos y clases de ese dataset, si no con un subconjunto suyo. Dicho subconjunto tendrá 25 clases y 12500 imágenes para entrenamiento y 2500 para la prueba.

Para la obtención del subconjunto de datos de Cifar100 hemos usado la función **cargarImagenes** proporcionada por el profesorado, tras ellos hemos desordenado de manera pseudoaleatoria los datos obtenidos usando como semilla *123456*. También se ha establecido el formato de las imágenes, como Keras usa el criterio *Channel First* tenemos que el formato será (3,filas_imagen,columnas_imagen) donde el número tres representa los canales de la imagen.

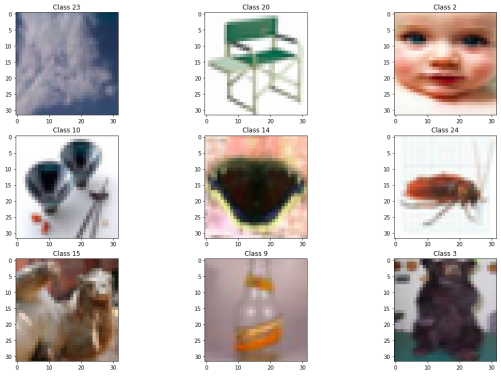


Figura 1: Algunas imágenes de CIFAR100

Hemos usado la **función de entropía cruzada categórica** como función de pérdida la cual se define mediante la expresión $loss = -\sum_{i=1}^{n} y_i log \hat{y_i}$ donde n es el tamaño de salida, y_i es la etiqueta i-ésima e $\hat{y_i}$ es la salida i-ésima del modelo. Esta función de pérdida es muy útil para tareas de clasificación multiclase donde una imágen tiene una sola etiqueta de un conjunto de varias, por eso la hemos usado. Comentar que hemos usado 30 épocas en el entrenamiento y un tamaño de batch de 64 para el conjunto de training, para los conjuntos de validación se ha usado un tamaño de batch menor.

Como métrica hemos usado lo que se conoce como **accuracy** que es la precisión que ha tenido el modelo, esto es, el porcentaje de predicciones correctas. Esta medida del rendimiento no es adecuada para problemas en las que haya un desbalanceamiento de clases, pero como no es el caso, la utilizaremos como única medida de rendimiento.

En cuanto al optimizador seleccionado nos hemos quedado con el *Nadam* que ha sido el que mejores resultados ha presentado tras un pequeño experimento usando validación cruzada (solo con los datos de entrenamiento) y donde se han probado otros optimizadores (Adadelta, SGD, Nadam, Adam, RMSprop, Adagrad, Adamax y Ftrl). En el experimento se ha obtenido un accuracy de *0.44232* en el modelo, el más alto. Aunque en términos generales, el modelo presenta un rendimiento bastante pobre.

Optimizador	accuracy
Adadelta	0.050448
SGD	0.333952
Nadam	0.44232
Adam	0.438416
TMSprop	0.426832
Adagrad	0.092032
Adamax	0.39584
Ftrl	0.04

Tabla 1: Comparativa de optimizadores: Podemos ver algo interesante, los optimizadores Adadelta y Ftrl presentan resultados nefastos, lo cual puede deberse a muchos factores, aunque tampoco entraremos en detalle.

Tras la elección del optimizador que mejores resultados aporta en la validación cruzada, entrenamos de nuevo el modelo con todos los datos de entrenamiento, con un tamaño de batch de 64 y durante 30 épocas, dejando un 10 % de ellos para validación y lo evaluamos en el conjunto de test . En general observamos, gracias a la gráfica 2 que a medida que aumentamos en época la diferencia entre el error de entrenamiento y el de validación es cada vez más grande, lo cual es un indicio de un claro sobreajuste en el entrenamiento. El error en el conjunto de entrenamiento va disminuyendo a medida que avanza el entrenamiento, mientras que el error de validación tiende a estabilizarse. Además la tabla 2 nos indica que la precisión es bastante pobre, acertamos menos de la mitad de las predicciones. Cabe notar también que, al tener más dtos para el entrenamiento de los que usamos la hora de hacer cross-validation para elegir el mejor optimizador, el accuracy sube ligeramente, de un 0.44 a un 0.6 .

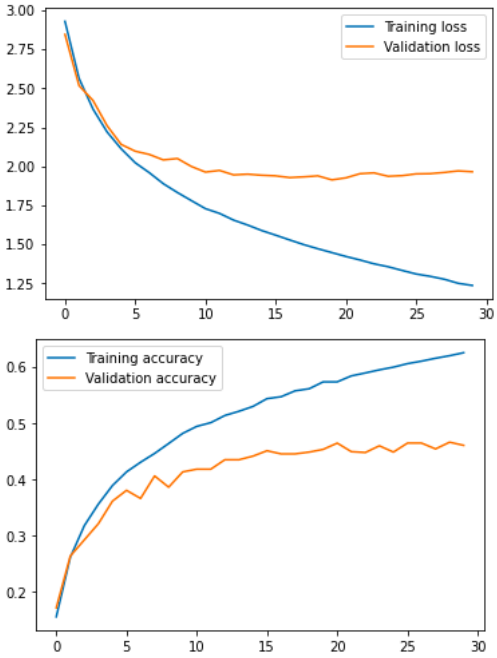


Figura 2

Tabla 2: Resultados

Accuracy BaseNet:	**0.46880000829696655**
Loss BaseNet:	**1.9520515203475952**

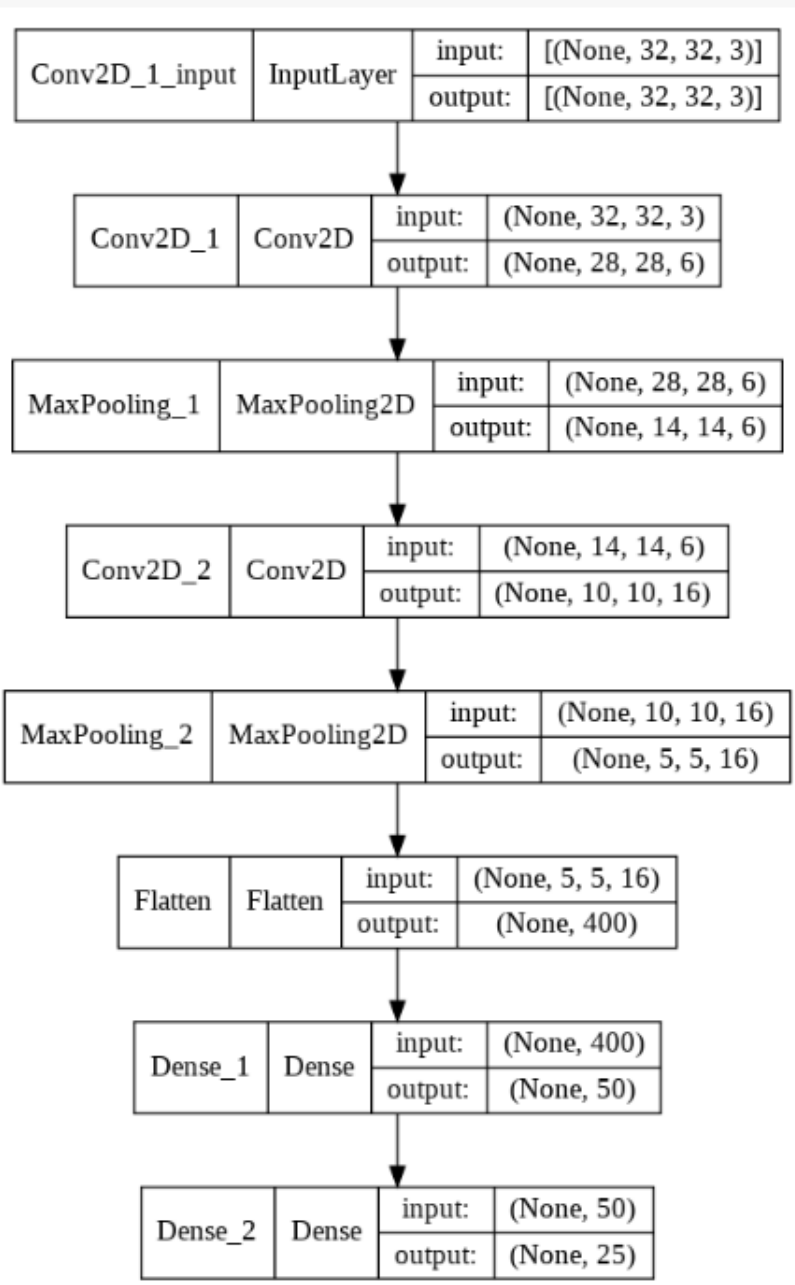


Figura 3: Arquitectura del modelo BaseNet.

3. Ejercicio 2 + Bonus

En este apartado presentaremos distintas versiones y subversiones incrementales con más o menos cambios realizados, ejecutados usando la técnica de cross-validation (5-folds) con los datos de entrenamiento sobre cada modelo para poder compararlos entre sí. Tras un breve análisis elegiremos uno de ellos para ejecutarlo con los datos de test para ver su capacidad de generalización.

El bonus corresponde con la versión 3, en la que se ha usado una técnica distinta de las ya propuestas en el enunciado de la práctica.

3.1. Versión 1

Presentamos ahora la versión 1, la cual consta de dos sub-versiones. Los cambios comunes a las dos sub-versiones con respecto al modelo BasetNet han sido sustituir el tamaño de los filtros convoluciones de 5x5, por dos filtros 3x3, ya que de esta forma conseguimos eficiencia en lo relativo al número de parámetros. Además, se sabe que aplicar un filtro 5x5 equivale a aplicar dos filtros 3x3.

También se han añadido capas de *BatchNormalization* después de aplicar una convolución pero antes de aplicar la función de activación. La añado ya que como comenta el paper de BatchNormalization detallado en 5 es un mecanismo para acelerar el entrenamiento de las redes profundas y se basa en la premisa del "shift covariance"que se sabe que complica el entrenamiento de los sistemas de ML y eliminarlo de las capas de activacion internas de la red puede ayudar al entrenamiento evitando también el "sobreajuste".

Se han añadido capas de *DropOut*, que es una técnica que de forma aleatoria inutiliza neuronas con una cierta probabilidad para evitar el sobreentrenamiento. En el paper de dropout indicado en la bibliografía (5) recomienda un valor mayor o igual a 0.4, en nuestro caso eligiremos 0.4.

Se han *normalizado* los datos, tanto de test como de entrenamiento usando **DataImageGenerator**, de Keras. Para ello, se han calculado los parámetros usando el conjunto de entrenamiento (habiéndole quitado previamente el de validación) con el propósito de usarlos para normalizar a media cero y desviación estándar 1 los datos de entrenamiento. Posteriormente, con dichos parámetros de normalización, normalizamos también el test y el conjunto de validación.

También se ha realizado un aumento de datos en el conjunto de entrenamiento, para ello se ha vuelto a usar la clase DataImageGenerator con el flag *horizontal_flip* activado y un *zoom_range* = 0.2 ya que en las imágenes aparecen pájaros de distinto tamaño y de distintas perspectivas, por lo que una reflexión horizontal y un aumento de tamaño pueden llegar a ser beneficiosos (Se han probado a usar distintos flags como el factor de rotación entre otros, pero los resultados empeoraban mucho, por ese motivo se han usado solo estas dos transformaciones).

Por último, se ha empleado la técnica de *EarlyStopping* que monitoriza el entrenamiento guardando los pesos junto con el rendimiento en cada época hasta que se aprecie que el error de validación aumenta de forma sostenida, momento en el cual se para el entrenamiendo y se reestablecen los pesos en la época que tenía justo antes de empeorar. Se ha establecido un valor 4 de paciencia, para lograr por un lado mayor rapidez en el entrenamiento y evitar sobreajuste. Hasta aquí los aspectos comunes a las dos sub-versiones, ahora se procede a comentar las diferencias entre ellas, que estriban en el modelo.

La regla más extendida sería la de entrenar el modelo monitorizando su rendimiento y guardando sus parámetros al finalizar cada epoch, hasta que apreciemos que el error de validación aumenta de forma sostenida (hay empeoramientos que son debidos a la componente estocástica del algoritmo). Nos quedaremos con el modelo que teníamos justo en el momento anterior.

La primera versión tiene filtros de profundidad 16 y 32 y tras aplicar la capa de *Flatten* tenemos las siguientes capas:

- DropOut con 0.4
- Fully Connected con 50 unidades
- BatchNormalization + ReLu
- DropOut con 0.4
- Fully Connected con 25 unidades ocultas + SoftMax

La versión 2 tiene filtros de profundidad 32 y 64 y tras la capa de *Flatten* añadimos una capa de *DropOut* con 0.4 y una capa *Fully Connected* con 25 capas ocultas + *SoftMax*. Las arquitecturas de estas versiones se puede apreciar en las Fig 4 y 5

Como puede observarse el modelo 1 tiene más capas que el modelo 2, pero este último consta de filtros con mayor profundidad. Por un lado las capas Dense de la versión 1 añaden más de complejidad al modelo, en cambio hemos querido evitar esa complejidad en el modelo 2, pero la hemos añadido por otro lado al aumentar la profundidad de los filtros. De hecho, esto provoca unos mejores resultados (tabla 3)

Tabla 3: Resultados

	Versión2	**Versión1**
Accuracy:	0.506080	0.486319
Loss:	1.6825984	1.760656

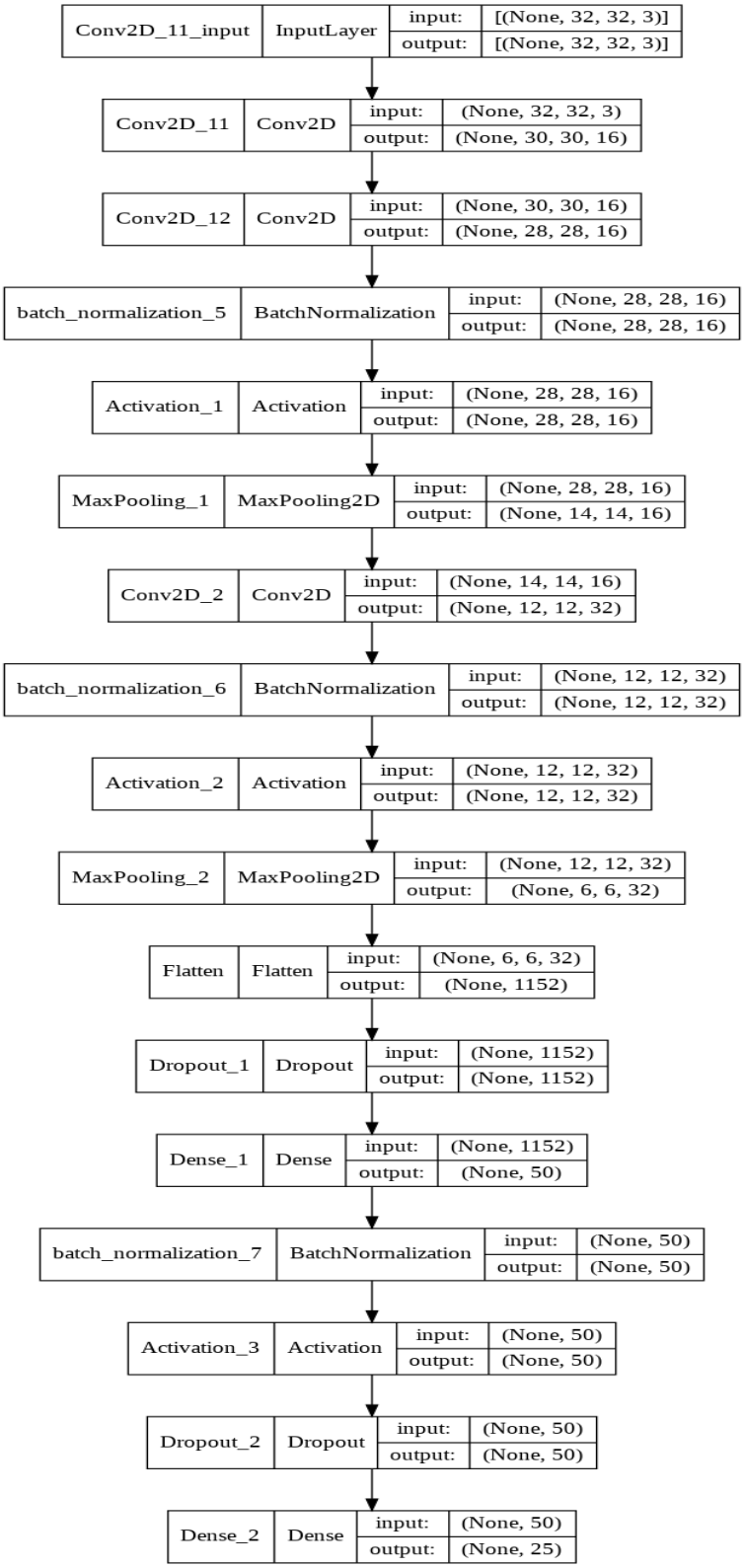


Figura 4: Arquitectura Versión 1

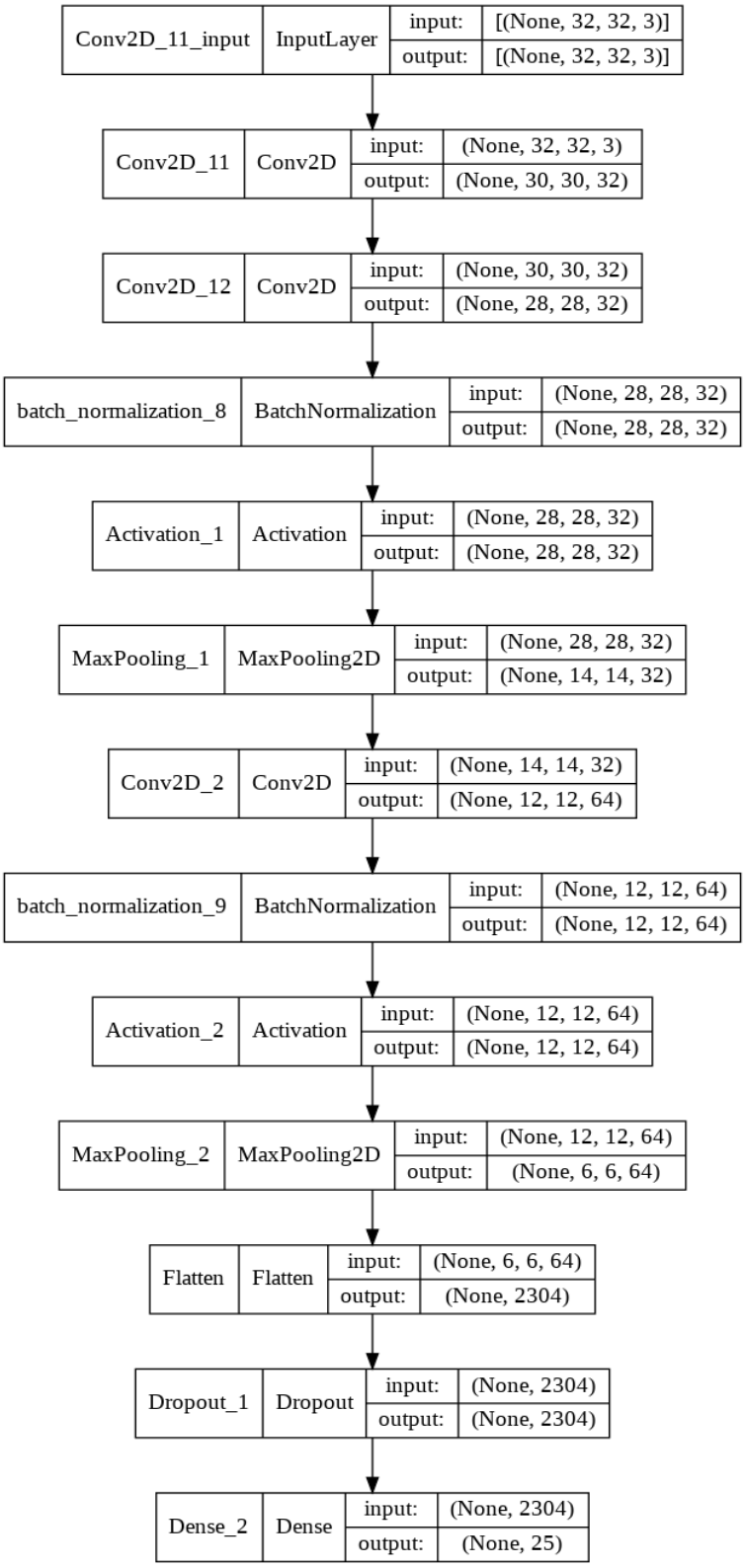


Figura 5: Arquitectura Versión 2

3.2. Versión 2

Esta segunda versión del ejercicio radica en cambios principalmente realizados al modelo en cuanto a profundidad con respecto a la segunda versión del apartado anterior por ser la que mejores resultados aportaba. También se han realizado 2 sub-versiones y la diferencia estriba en la profundidad de los filtros. La primera versión tiene filtros de 32, 62, 128 y 256 mientras que la segunda tiene filtros con profundidad 128, 256, 512. (Véase Fig 6 y 7)

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Conv2D_1 (Conv2D)            (None, 32, 32, 32)        896

batch_normalization_10 (Bat  (None, 32, 32, 32)        128
chNormalization)

Activation_1 (Activation)    (None, 32, 32, 32)        0

Conv2D_2 (Conv2D)            (None, 32, 32, 64)        18496

batch_normalization_11 (Bat  (None, 32, 32, 64)        256
chNormalization)

Activation_2 (Activation)    (None, 32, 32, 64)        0

MaxPooling_1 (MaxPooling2D)  (None, 16, 16, 64)        0

up_sampling2d (UpSampling2D  (None, 32, 32, 64)        0
)

Conv2D_3 (Conv2D)            (None, 16, 16, 128)       73856

batch_normalization_12 (Bat  (None, 16, 16, 128)       512
chNormalization)

Activation_3 (Activation)    (None, 16, 16, 128)       0

Conv2D_4 (Conv2D)            (None, 16, 16, 128)       147584

batch_normalization_13 (Bat  (None, 16, 16, 128)       512
chNormalization)

Activation_4 (Activation)    (None, 16, 16, 128)       0

Conv2D_5 (Conv2D)            (None, 16, 16, 256)       295168

batch_normalization_14 (Bat  (None, 16, 16, 256)       1024
chNormalization)

Activation_5 (Activation)    (None, 16, 16, 256)       0

MaxPooling_2 (MaxPooling2D)  (None, 8, 8, 256)         0

Flatten (Flatten)            (None, 16384)             0

Dropout_1 (Dropout)          (None, 16384)             0

Classifier (Dense)           (None, 25)                409625

=================================================================
Total params: 948,057
Trainable params: 946,841
Non-trainable params: 1,216
_________________________________________________________________
```

Figura 6: Arquitectura Versión 1

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Conv2D_1 (Conv2D)            (None, 32, 32, 128)       3584

batch_normalization_15 (Bat  (None, 32, 32, 128)       512
chNormalization)

Activation_1 (Activation)    (None, 32, 32, 128)       0

Conv2D_2 (Conv2D)            (None, 32, 32, 256)       295168

batch_normalization_16 (Bat  (None, 32, 32, 256)       1024
chNormalization)

Activation_2 (Activation)    (None, 32, 32, 256)       0

MaxPooling_1 (MaxPooling2D)  (None, 16, 16, 256)       0

up_sampling2d_1 (UpSampling  (None, 32, 32, 256)       0
2D)

Conv2D_3 (Conv2D)            (None, 16, 16, 128)       295040

batch_normalization_17 (Bat  (None, 16, 16, 128)       512
chNormalization)

Activation_3 (Activation)    (None, 16, 16, 128)       0

Conv2D_4 (Conv2D)            (None, 16, 16, 256)       295168

batch_normalization_18 (Bat  (None, 16, 16, 256)       1024
chNormalization)

Activation_4 (Activation)    (None, 16, 16, 256)       0

Conv2D_5 (Conv2D)            (None, 16, 16, 512)       1180160

batch_normalization_19 (Bat  (None, 16, 16, 512)       2048
chNormalization)

Activation_5 (Activation)    (None, 16, 16, 512)       0

MaxPooling_2 (MaxPooling2D)  (None, 8, 8, 512)         0

Flatten (Flatten)            (None, 32768)             0

Dropout_1 (Dropout)          (None, 32768)             0

Classifier (Dense)           (None, 25)                819225

=================================================================
Total params: 2,893,465
Trainable params: 2,890,905
Non-trainable params: 2,560
```

Figura 7: Arquitectura Versión 2

Los resultados a través del procedimiento experimental usando cross-validation mejoran a los de la versión anterior. Además, podemos observar (Tabla 4) que son parecidos entre sí. Aunque la segunda versión presenta mejores resultados que la primera. La diferencia en el accuracy es aproximadamente de 0.03, por lo que al no existir una diferencia demasiado grande, consideramos la primera versión como más preferible que la segunda. Esto se debe a que la segunda

versión tiene aproximadamente 2 millones de parámetros más que la primera y obtiene muy poca mejora con respecto a ésta.

Tabla 4: Resultados

	Versión1	**Versión2**
Accuracy:	0.5168000161647797	0.5417599976062775
Loss:	1.6215078592300416	1.553694748878479

Dentro de las 2 versiones realizadas (4 modelos en total) vemos que los que mejores resultados presentan son las sub-versiones de la versión 2 y de ésta nos quedamos con la primera por lo anteriormente comentado (pese a que no sea mejor que la segunda). Por lo que se procede a realizar un entrenamiento del modelo con todos los datos de entrenamiento exceptuando un 10 % para validación y se va a evaluar en el conjunto de test para ver como de bien generaliza.

Como se puede observar en la tabla 6, tenemos una accuracy aproximadamente de 0.6 siendo mayor que en el entrenamiento. Esto quiere decir que el modelo generaliza bien, algo que realmente es deseable. Sin embargo, no suele ser habitual que esto ocurra, aún así podemos inferir que se debe a las capas de BatchNormalization, DropOut y sobre todo a la técnica de data-aumentation todo lo cual contribuye a disminuir el sobreajuste en los datos de entrenamiento. De todas formas, también hay que notar que hemos entrenado con un menor número de ejemplos de entrenamiento en la fase experimental, pues en el cross-validation cogíamos en cada fold un 20 % de los datos de entrenamiento para usarlos como test, y dentro del 80 % restante tomábamos un 10 % para validación provocando un entrenamiento menos preciso.

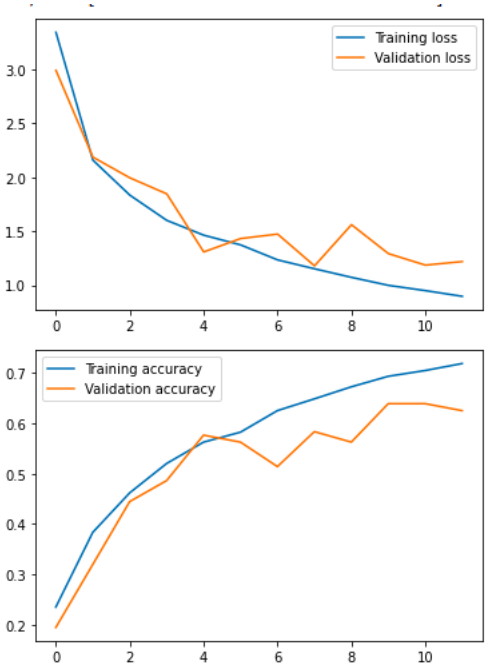


Figura 8: Esta gráfica se corresponde a la validación realizada durante el entrenamiento (no al test). Podemos ver que pese a ser distintas tienen una forma parecida y no muy alejadas entre sí, por lo que el sobreajuste no es demasiado pronunciado.

3.3. Versión 3 - BONUS

Este apartado se ha realizado expresamente para el Bonus aunque no deja de cumplir la finalidad del ejercicio 2. La idea se basa en el aumento de profundidad y en la introducción de módulos residuales aunque no al estilo de ResNet. Se sabe que los módulos residuales permiten aumentar enormemente el número de capas de una red neuronal sin llegar a overfiting ni a que se desvanezcan los gradientes y es hoy en día un elemento básico en la mayor parte de las redes modernas (Paper: Deep Residual Learning for Image Recognition 5)

Se han hecho también dos versiones, aunque son prácticamente las mismas exceptuando un ligero matiz. En la primera versión, cada modulo residual tiene una capa convolucional más otra de BatchNormalization seguida de una de activación, es decir, se ha aplicado la función de activación al final de cada bloque, antes de hacer la concatenación. Mientras que en la segunda versión,que contiene dos capas convolucionales, la primera seguida de BatchNormalization y una capa de activación, y la segunda solo de una capa de batchNormalization, la función de activación se ha realizado tras la concatenación. En la literatura se expone esta segunda aproximación aún así hemos optado también por la primera simplemente para comprobar como esto afecta a los resultados (mera curiosidad).

Algo que también destaca en estas dos versiones con respecto a las versiones del ejercicio dos, en donde la reducción de la dimensión no es tan alta para ser significativa, es el uso de padding para mantener la dimensionalidad de la entrada a la salida. Si no tenemos esto en cuenta tenemos que tras cada cada convolución 3x3 la dimensión de la salida es dos unidades menor que la de la entrada. Al ser una red algo más profunda y tener entradas de 32x32(x3) ocurre que la dimensión se va a reducir a lo largo del modelo que con el efecto de las capas de pooling dará lugar a una salida de un tamaño muy pequeño, lo cual no es deseable y por eso se considera el padding.

En las figuras 9, 10, 11 y 12 podemos ver la arquitectura que hemos tomado y los módulos residuales de las distintas versiones.

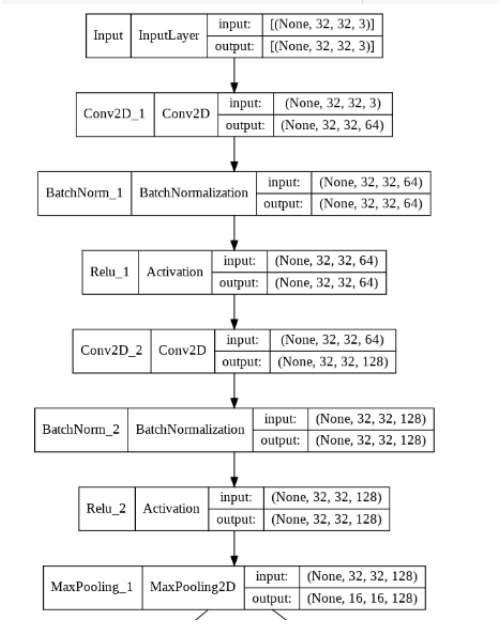


Figura 9: (Inicio) Esta parte es común a las dos versiones

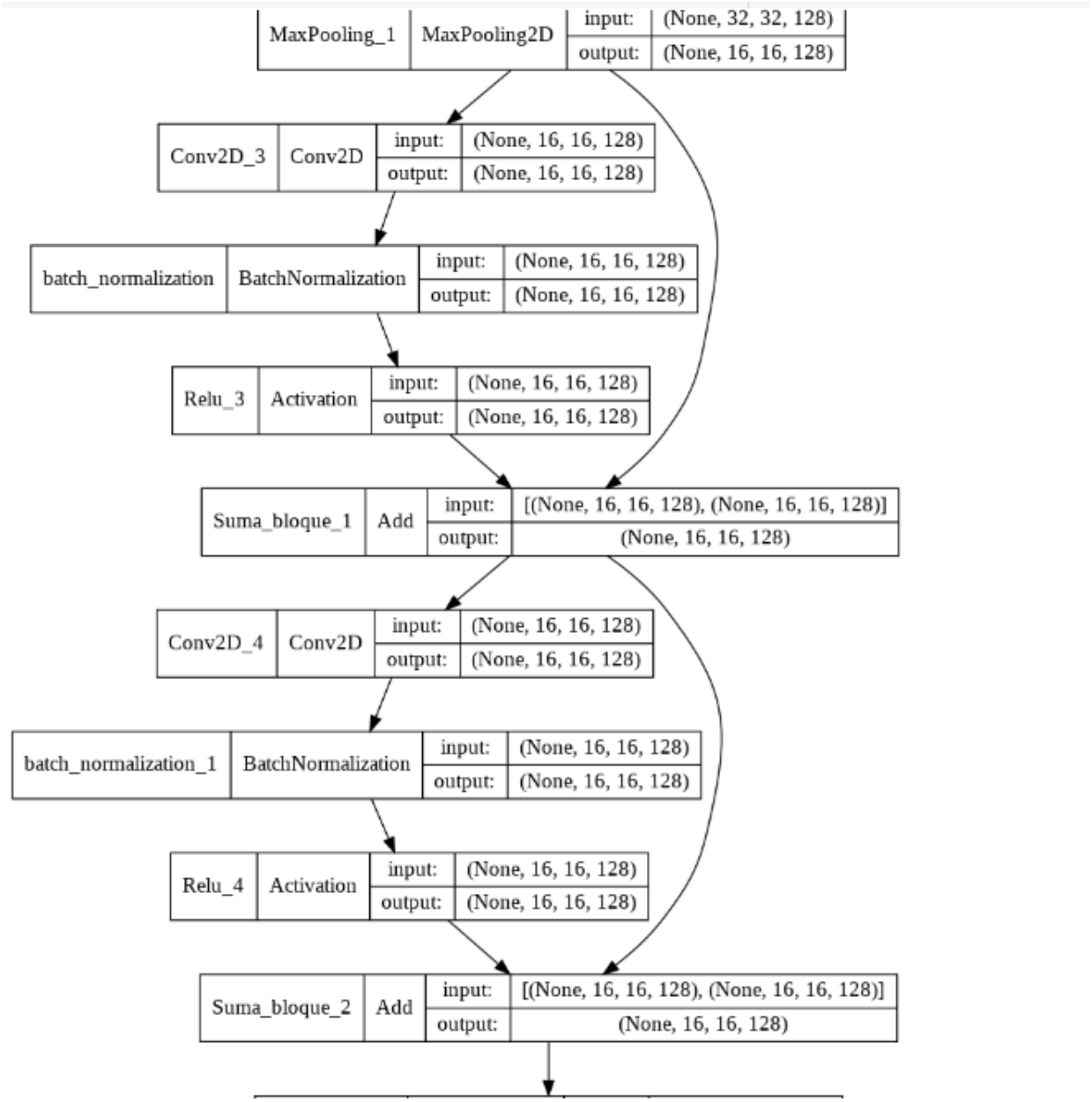


Figura 10: Esta parte pertenece a la primera versión. Podemos ver cómo en cada módulo residual tenemos una capa de convolución 2D, seguida de BatchNormalization y la función de activación ReLu antes de la concatenación.

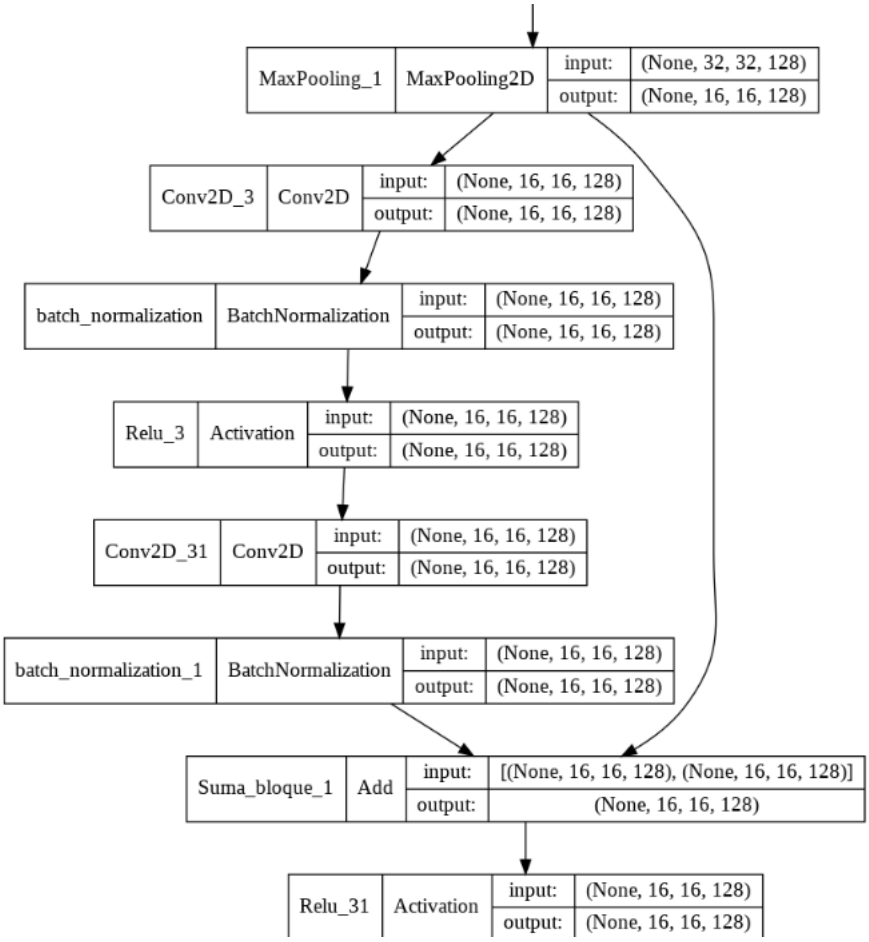


Figura 11: Esta parte pertenece a la segunda versión. En total son dos bloques residuales (iguales) pero hemos detallado solo uno. En él se puede ver que tras la capa de pooling el bloque residual tiene una capa de convolución seguida de BatchNormalization + ReLu. Después presenta otra capa convolucional con BatchNormalization antes de cerrar el bloque y tras la concatenación, aplicamos la capa ReLu.

Tabla 5: Resultados en el proceso de experimentación usando cross-validation

	Versión_1-1	**Version-1-2**	**Versión-2-1**	**Version-2-2**	**Versión-3-1**	**Versión-3-2**
Accuracy:	0.486319	0.50608	0.51680	0.541759	0.6226400	0.6269600
Loss:	1.76065	1.68259	1.62150	1.55369	1.3241237	1.2348142

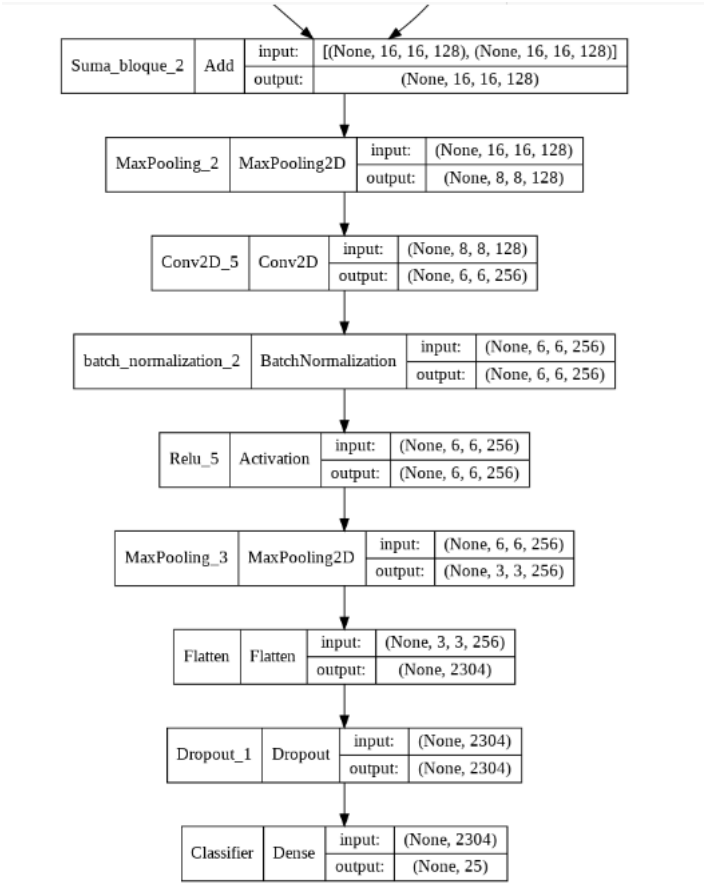


Figura 12: (Fin) Esta parte es común a las dos versiones

	Final-Model_v2	**Final-Model_v3**
Accuracy:	0.6068000197410583	0.7063999772071838
Loss:	1.2824790477752686	0.9805776476860046

Tabla 6: Resultados-Test: Los resultados de la izquierda corresponden al modelo elegido para evaluar dentro de los modelos del ejercicio 2 y los resultados de la derecha corresponden al modelo elegido dentro de los del bonus para evaluar con los datos de test.

Los resultados en el proceso experimental quedan visibles en la tabla 5. En ella se puede ver como los resultados mejoran con respecto a todas las versiones realizadas. También se puede apreciar que los resultados para estas dos nuevas versiones son muy similares, luego para este problema podemos concluir que las diferencias entre ellas no tienen efectos significativos.

Se escoge ahora el segundo modelo para entrenarlo con un 90 % de los datos de entrenamiento (10 % se dejan para validación) y evaluarlo con los datos de test, la tabla 6 muestra los resultados de esta evaluación en la columna correspondiente a *Final-Model-v3*. Podemos ver que el accuracy es aproximadamente de 0.7, obteniendo bastante capacidad de generalización. Los posibles motivos por los que hemos llegado a obtener mejores resultados en la evaluación con el test que con la validación ya se comentaron en el apartado anterior.

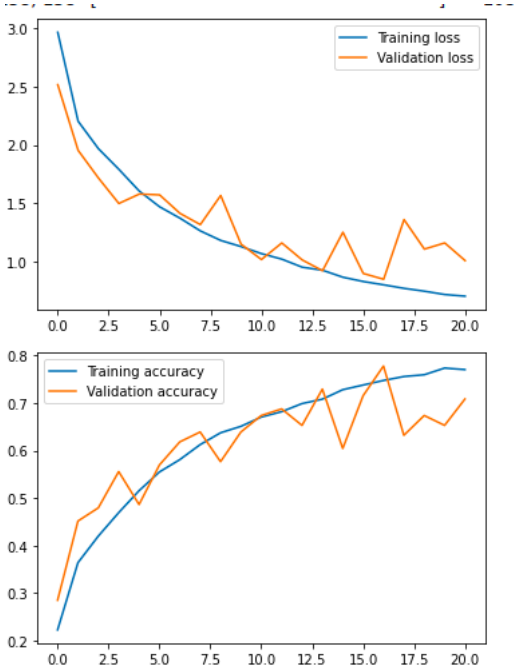


Figura 13: Gráfica en donde se muestra la curva de entrenamiento y validación tanto para métrica de accuracy como la función de pérdida para el último modelo realizado. Aquí se puede ver que el sobreajuste es pequeño y por ello generaliza bien. Cosa que puede ocurrrir gracias a los bloques residuales inroducidos

4. Ejercicio 3

En este ejercicio usamos una base de datos distinta, en concreto con *Caltech-UCSD*. Contiene 6033 imágenes de 200 (clases) especias de pájaros, de las cuales 3000 imágenes sson para entrenamiento y 3033 son para test. Se usará el modelo de redNest50 preentrenado con Imagenet y disponble en Keras.

Vamos a comentar como hemos tratado los datos; Se han creado dos conjuntos de datos, se denotarán como primer conjunto y segundo conjunto. En el *primer conjunto* se dividen los datos de entrenamiento en dos subconjuntos $[entrenamiento, Test\_val]$ a razón de 85-15. Posteriormente, el conjunto de entrenamiento lo partimos en otros dos subconjuntos $[T-training, T-Validacion]$. Este *primer conjunto* de datos será usado para comparar varios modelos con el fin de buscar mejoras o tunear parámetros. Por tanto, *Test_val* hará la función de conjunto de test, *T-training* hará la función de conjunto de entrenamiento y *T-Validación* juega de papel de la evaluación en el conjunto de entrenamiento.

En el *segundo conjunto* dividimos los datos de entrenamiento en dos partes: [entranamiento - validacion] a razón de 85-15. Este conjunto de datos tiene el propósito de usar entrenamiento para entrenar el modelo, validar con validación y evaluar finalmente el modelo con los datos de test. Ambos conjuntos de datos cumplen papeles distintos.

También hemos usado el optimizador Adam en vez de Nadam, ya que el segundo se estableció en apartados anteriores de acuerdo a un proceso experimental sobre unos modelos aplicados a una base de datos distinta de la que estamos trabajando ahora, debido a ello, no tiene por qué seguir siendo el más óptimo. Por otra parte, la literatura recomienda usar Adam por sus buenos resultados y por ello lo hemos escogido.

Se vuelve a reiterar que en este ejercicio, los resultados en el proceso experimental son poco significativos pues no hemos usado cross-validation por lo expuesto en el apartado de *Aclaraciones y comentarios.*

4.1. Apartado 1

En este apartado se adaptará el modelo ResNet50 entrenado con ImageNet a nuestros datos (ResNet_v1) quitando la última capa y añadiendo otra de salida. Posteriormente, se añadirán más capas totalmente conectadas al modelo anterior (ResNet_v2). Tras ello, se eliminarán las capas de salida, FC y AveragePooling (ResNet_v3), la comparación se hará en el apartado de *Comparación* más adelante.

RESNET_v1

Primeramente, para adaptar el modelo ResNet50 entrenado con ImageNet a nuestros datos, se ha usado la función:

keras.applications.resnet.ResNet50(include_top = False, weights = imagenet, pooling = avg)

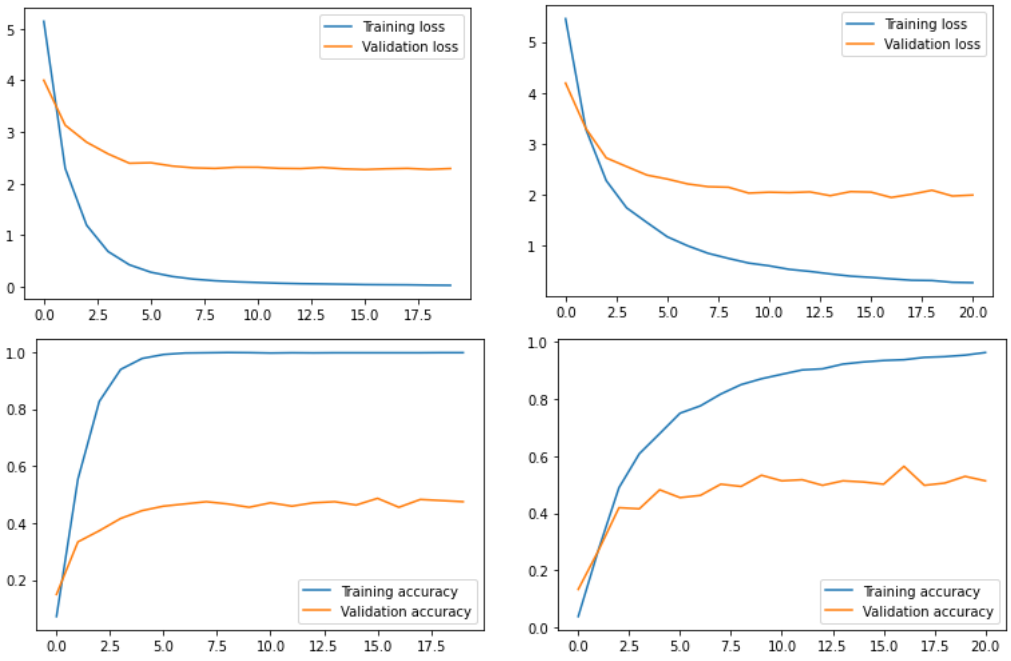
en donde el primer argumento indica que no se incluya una capa totalmente conectada al final de la red, el segundo argumento hace referencia a que use los pesos adaptados a ImageNet y el tercer argumento que indica el pooling opcional para la extracción de características.

Tras ello hemos indicando que los pesos adaptados por ImageNet no cambien durante el entrenamiendo pues queremos ver el desempeño que tiene la red usando esos pesos y además, se ha añadido una capa totalmente conectada con 200 perceptrones por ser este el número de clases que hay.

Se han aplicado dos ejecuciones con el *primer conjunto de datos* para probarlo con y sin aumento de datos en el conjunto de entrenamiento logramos mejores resultados o no. Y como se aprecia en la tabla 7 observamos que el aumento de datos provoca un efecto negativo con respecto a la anterior ejecución. Por lo que se usará el primer modelo para evaluarlo con los datos de test mas adelante.

	Ejec. I	**Ejec. II**
Accuracy:	0.5088889002799988	0.41999998688697815
Loss:	2.0756959915161133	2.354219675064087

Tabla 7: Resultados de las ejecuciones del modelo basado en ImageNet adaptado a los datos Caltech-UCSD. La columna de la izquierda se corresponde al modelo en el que no se han aumentado los datos en el conjunto de entrenamiento y la columna de la derecha al modelo en el que se han aumentado los datos.



(a) Ejec. I (b) Ejec. II

Figura 14: Curvas de error de entrenamiendo y validacion y curvas de accuracy en el entrenamiendo y validación. La columna de la izquierda corresponde al modelo en el que no se han aumentado los datos y la de la derecha al modelo en el que sí. Se pude apreciar un claramente que en ambas ejecuciones la precisión del entrenamiendo llega a valer 1 distanciándose de la precisión en la validación, esto es, un claro sobreentrenamiendo. También se puede observar en las gráficas relativas al error, donde el error de entrenamiento tiende a cero mientras que el de validación se mantiene muy por encima.

RESNET_v2

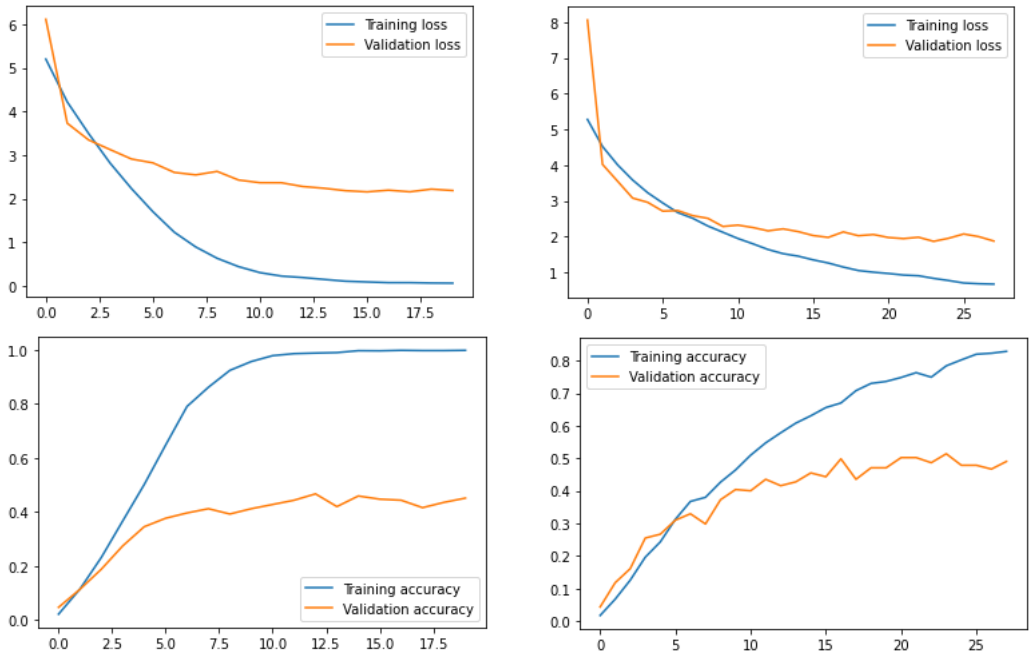
Ahora vamos a añadir las siguientes capas al modelo anterior: (La elección del rate en la capa de DropOut ha sido en base a un proceso experimental el cual no vamos a especificar)

- DropOut con rate = 0.6
- Dense con 4096 neuronas
- DropOut con rate = 0.6
- Dense con 200 neuronas y softmax

Se han realizado dos ejecuciones con el *primer conjunto de datos* de la misma forma que la anterior, una sin aumento de datos en el entrenamiento y la otra con. Observamos (tabla 8) que ambas ejecuciones son más o menas parecidas. En las gráficas, (Fig. 15) podemos apreciar que pare el caso de la ejecución en la que no aumentamos datos (Ejec. III) la curva del la función de error en el entrenamiento tiende a cero distanciándose mucho de la de validación, lo que se traduce en un posible claro sobreentrenamiendo, de hecho, se puede apreciar en en las curvas de la precisión, el entrenamiento tiende a 1 estando muy alejada de la de validación. Algo distinto ocurre en la ejecución en la que aumentamos los datos (Eje. IV) ahora las curvas de training loss y validation loss van muy a la par, aunque en la curva del accuracy se separan algo más.

	Ejec. III	**Ejec. IV**
Accuracy:	0.4511111080646515	0.46000000834465027
Loss:	2.1056854724884033	2.165424108505249

Tabla 8: Resultados de las ejecuciones del modelo ampliado basado en ImageNet adaptado a los datos Caltech-UCSD. La columna de la izquierda se corresponde al modelo en el que no se han aumentado los datos en el conjunto de entrenamiento y la columna de la derecha al modelo en el que se han aumentado los datos.



(a) Ejec. III (b) Ejec. IV

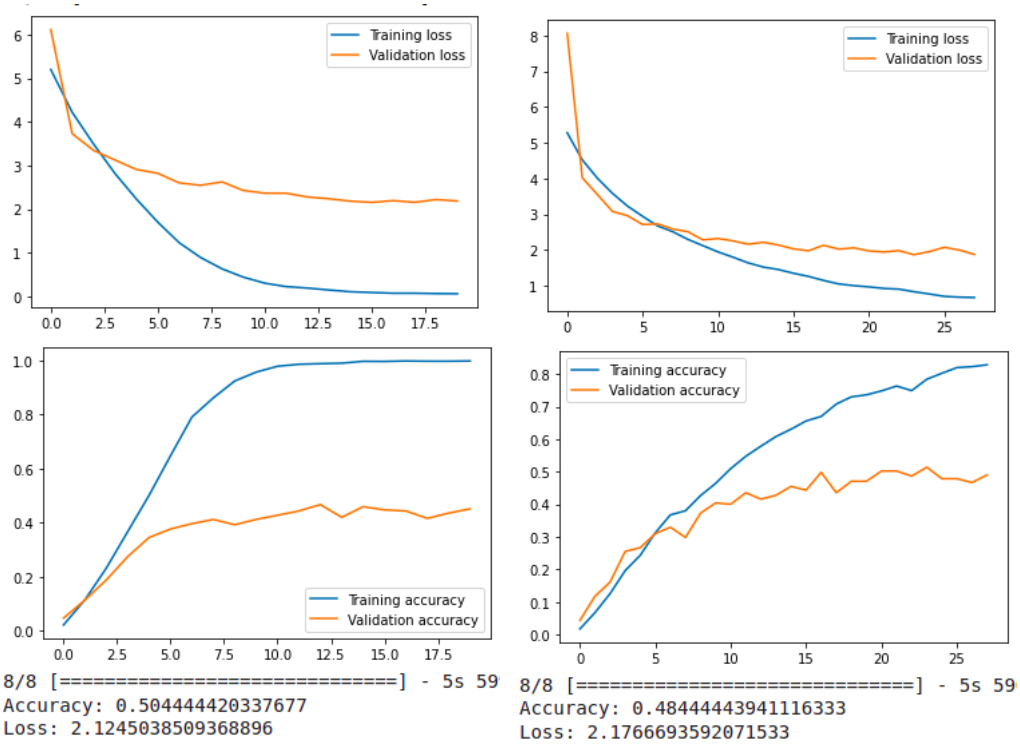
Figura 15: Curvas de error de entrenamiento y validación y curvas de accuracy en el entrenamiento y validación. La columna de la izquierda corresponde al modelo en el que no se han aumentado los datos y la de la derecha al modelo en el que sí.

RESNET_v3

Partimos de la red ResNet con los siguientes parámetros (include_top = False, weights = imagenet, pooling = None) en donde tomamos los pesos de imagenet, eliminamos la última capa y no elegimos opción de pooling. A esta red le añadimos las siguientes capas elegidas en base a un proceso experimental el cual no vamos a comentar (para no alargar demasiado la memoria):

- Capa convolucional de 512 filtros de 3x3
- Capa de BatchNormalization con parametros por defecto
- Capa de activación usando la ReLu como funcion de activación
- Capa convolucional de 256 filtros de 3x3
- Capa de BatchNormalization con parametros por defecto
- Capa de activación usando la ReLu como funcion de activación
- Capa convolucional de 256 filtros de 3x3
- Capa de BatchNormalization con parametros por defecto
- Capa de activación usando la ReLu como funcion de activación
- Capa de Global Average Pooling
- Capa de Dropout con rate=0.6
- Capa Dense con 200 neuronas y softmax

Como puede verse en la Fig. 16 los resultados sin el aumento de datos son un poco mejores que con el aumento de datos. Sin embargo, cuando se valida en el entrenamiento se puede apreciar ciertos indicios de sobreentrenamiendo cuando no se aplica Data Augmentation. Como con los datos aumentados los resultados son similares, de hecho no hay mejora y además el tiempo de ejecución es mucho mayor, concluimos que el tipo de aumento de datos producidos no tiene un gran impacto en este caso, por lo que no se usará aumento de datos con este modelo. Sin embargo, cabe notar que con el aumento de datos el sobreajuste disminuye considerablemente, pues en las gráficas (ResNet_v3 con Data Augmentation) podemos ver que la diferencia entre los resultados en el conjunto de training y validación es menor en este caso.



(a) ResNet_v3 sin Data Augmentation (b) ResNet_v3 con Data Augmentation

Figura 16: Curvas de error de entrenamiento y validación y curvas de accuracy en el entrenamiento y validación. La columna de la izquierda corresponde al modelo en el que no se han aumentado los datos y la de la derecha al modelo en el que sí.

4.2. Apartado 2

En este apartado haremos lo que se conoce como ajuste fino o *fine tuning* a la red ResNet adaptada a nuestro problema. Según en la documentación de Keras (Ver Bibliografía 5) para hacer el ajuste fino hay que:

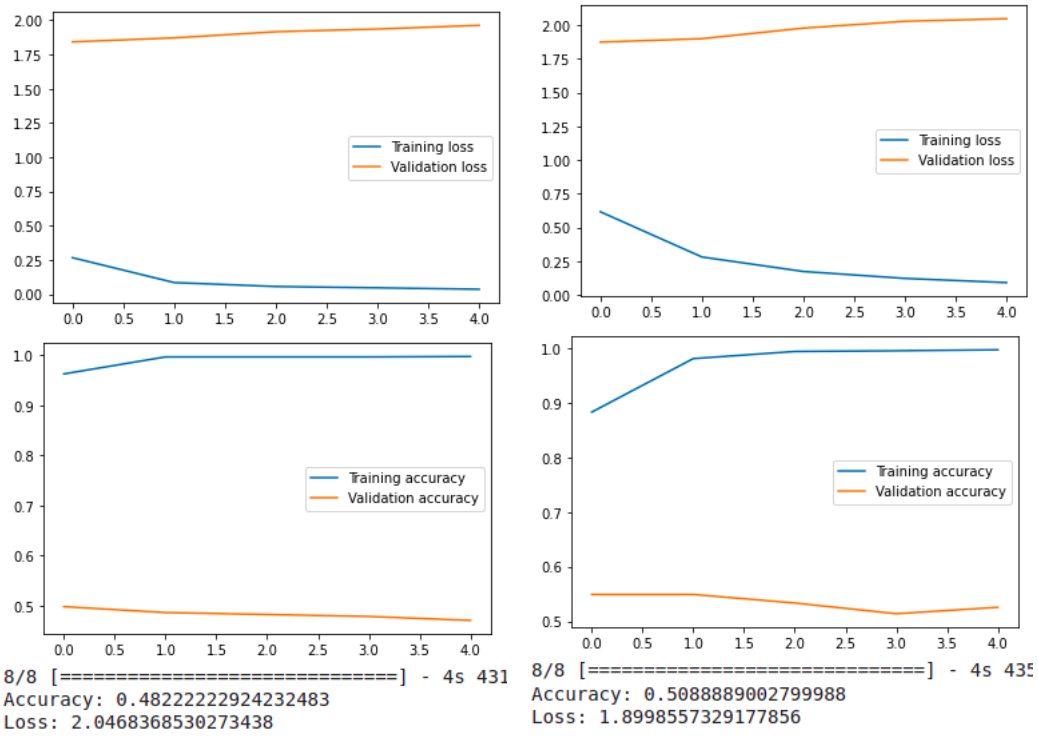
1. Instanciar el modelo base y cargar los pesos preentrenados
2. Congelar todas las capas del modelo base haciendo trainable = False
3. Crear un nuevo modelo en el tope del modelo base
4. Entrenar al nuevo modelo con el conjunto de datos
5. Descongelar todo el modelo base o una parte de él
6. Reentrenar todo el modelo (end-to-end) con un learning rate muy bajo

Se han creado dos versiones diferenciándose únicamente en el paso (3), en los otros pasos son iguales. El primer paso es tal cual se ha hecho anteriormente, para el segundo paso únicamente tenemos que activar el atributo *trainable* a *True* en el modelo base de ResNet. En el tercer paso, al primer modelo se le ha quitado la última capa del modelo base de ResNet50 y se ha añadido una FC con 200 unidades (pues es el número de clases que tenemos) mas la función softmax. A el segundo modelo partimos de ResNet50 sin la última capa y añadimos las capas que se añadieron en *restNet_v3* del apartado anterior.

El paso 4 (Enternar al nuevo modelo con el conjunto de datos) se ha realizado tal cual lo hemos ido haciendo hasta ahora. En el paso 5, cambiamos el atributo *trainable* a *False* en el model base de ResNet para descongelar las capas. Finalmente en el paso 6 reentrenamos el modelo entero pero teniendo en cuenta algunos detalles.

En primer lugar reducimos considerablemente el número de *épocas*, pasados de tener 30 a tener 10 con el propósito de no sobreentrenar a la red, también se ha escogido un *learning_rate* bastante bajo, con un valor de 1e(-5) ya que como dice en la web de Keras (Ver BiblioGrafía 5) *Es fundamental utilizar una tasa de aprendizaje muy baja en esta fase, porque se está entrenando un modelo mucho mayor que en la primera ronda de entrenamiento, en un conjunto de datos que suele ser muy pequeño. Como resultado, se corre el riesgo de sobreajustar muy rápidamente si se aplican grandes actualizaciones de pesos. En este caso, sólo se desea readaptar los pesos preentrenados de forma incremental.* También se ha optado por reducir el tamaño del batch pasando de tener 64 a 32 ya que pese a que ResNet use tamaños de Batch muy grandes, tenemos que tener en cuenta que la base de datos de ImageNet es inmensa en comparación con la que estamos trabajando en estos momentos y por lo comentado en un estudio (Ver Bibliografía 5) es favorable usar tamaños de batch pequeños, entonces se ha optado por usar dicho tamaño de batch y se han normalizado los datos para ese tamaño.

Establecemos ahora un proceso experimental en el que ejecutaremos con el *primer conjunto de datos* cada versión, con y sin *Data Augmentation*. Vemos que en la primera versión obtenemos mejores resultados haciendo *Data Aumentation* aún así también se aprecia un claro sobreentrenamiendo en ambas ejecuciones, pues en las gráficas observamos que la curva de error en el entrenamiendo tiende a cero mientras que la de validación está en valores comprendidos del intervalo $[1{,}75, 2]$ existiendo una clara lejanía entre ambas, análogo para la curva del accuracy.



(a) Primer Modelo, sin Data Aumentation (b) Primer Modelo, con Data Aumentation

Figura 17: Arriba: Curvas Training Loss, Validation Loss. Abajo: Training Accuracy y Validation Accuracy.

4.3. Comparación

Ahora vamos a seleccionar uno de los dos modelos de los realizados en los anteriores apartados de este ejercicios y vamos a emplear el *segundo conjunto de datos* para así evaluarlos en el conjunto de test y ver cual de ellos tiene más desempeño.

Del modelo *ResNet_v1* se ha elegido el que no se ha usado aumento de datos , de *ResNet_v2* se ha optado por el que sí se ha empleado esa técnica, de *ResNet_v3* se ha escogido la versión en la que no se ha empleado el aumento de datos y finalmente se ha tomado el modelo en el que no se ha usado Data Augmentation en el apartado de *fine tuning*.

Tabla 9: Resultados evaluando con el conjunto de test

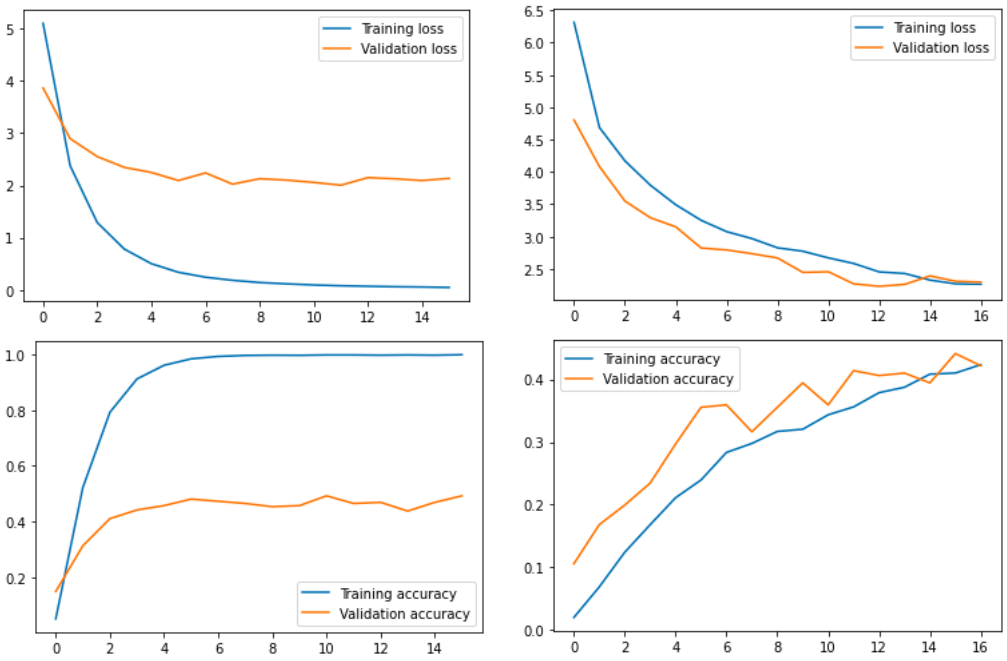
	ResNet_v1	**ResNet_v2**	**ResNet_v3**	**Fine_Tuning**
Accuracy:	0.420705586671	0.378173410892	0.390372574329	0.4437850415706
Loss:	2.384929895401	2.483037233352	2.584685802459	2.281478404998

Como se observar en la tabla de arriba (9) tenemos que dentro de las tres versiones de ResNet, sorprendentemente la que mejores resultados aporta en el conjunto de test es la primera, pese a ser la más básica de las tres. Le sigue la tercera y finalmente la segunda. También observamos que el la versión del fine tuning es la que mejores resultados aporta entre las tres.

En las gráficas de abajo (Fig. 18 y 19) tenemos que para ResNet_v1 hay en el proceso de validación se produce sobreentrenamieto ya que la distancias entre las curvas de accuracy es

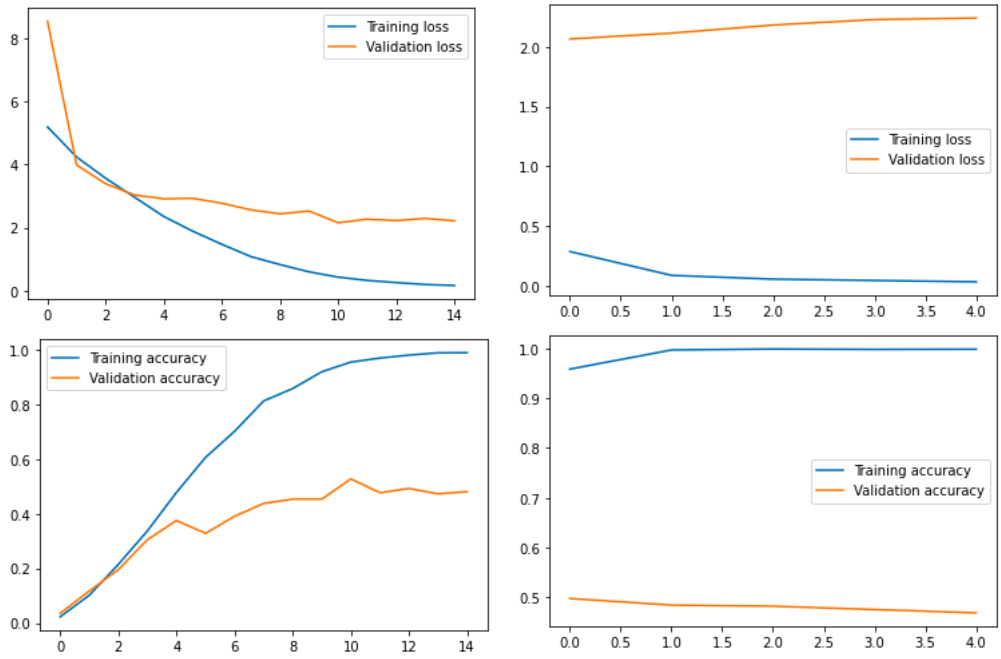
bastante, así como en ResNet_v3 y en el fine tune aunque en el caso de este último es más exagerado que en cualquier otro. En cambio, en ResNet_v2 no se produce sobreaprendizaje, aunque pese a ello los resultados siguen siendo malos.

Para concluir, decir que los resultados son malos en todos los casos, esto puede deberse en parte a que ResNet ha sido entrenada con una base de datos distinta a la que estamos usando y no está totalmente adaptada salvo las últimas capas de los tres primeros modelos. El que mejor capacidad de generalizar tiene es el modelo del fine tuning, esto puede deberse a que también se entrena el modelo base de ResNet y los pesos se adaptan un poco más a la base de datos que estamos usando.



(a) ResNet_v1 (b) ResNet_v2

Figura 18: Arriba: Curvas Training Loss, Validation Loss. Abajo: Training Accuracy y Validation Accuracy.



(a) ResNet_v3 (b) Fine Tune

Figura 19: Arriba: Curvas Training Loss, Validation Loss. Abajo: Training Accuracy y Validation Accuracy.

5. Bibliografía

BatchNormalization: https://arxiv.org/pdf/1502.03167.pdf

DropOut: https://jmlr.org/papers/v15/srivastava14a.html

Deep Residual Learning for Image Recognition: https://arxiv.org/pdf/1512.03385v1.pdf

Ajuste Fino: https://keras.io/guides/transfer_learning/

Small Batch Size : https://arxiv.org/pdf/1804.07612.pdf