Visión por Computador

Práctica 2

Curso 2020-2021

Javier Gálvez Obispo
javiergalvez@correo.ugr.es

Comentarios previos

La práctica ha sido desarrollada en Colab ya que es más sencillo configurar el uso de la GPU. En todo caso, se ha entregado tanto el cuaderno de Colab como el archivo .py generado al exportarlo desde Colab. Es por esta razón que el archivo .py no esta configurado para utilizar la GPU.

A la hora de ejecutar los apartados de la práctica en la que se han tenido que hacer varios experimentos, sólo se ejecuta el modelo final que ha dado mejores resultados para reducir el tiempo de ejecución. De todas formas, es posible ejecutar la mayoría de las demás pruebas realizadas descomentando la líneas de código que están indicadas.

Debido a la cantidad de generadores de números aleatorios que se utilizan para probar las distintos modelos no se ha conseguido fijarlos de forma que siempre se obtuviesen los mismos resultados para todas las ejecuciones. Es por esto que los resultados al ejecutar pueden variar respecto a los mostrados aquí, no obstante, las mejoras respecto unas variantes y otras se mantienen y se puede comprobar que todo los descrito en este documento ocurre en la práctica.

Ejercicio 1

Definimos una función *basenet* que nos devuelva el modelo definido por la siguiente tabla:

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

Y vamos a entrenarlo para que clasifique imágenes de la base de datos CIFAR100. Este conjunto de datos consta de 60K imágenes en color con dimensión 32x32x3 de 100 clases distintas. En nuestro problema tan sólo vamos a utilizar 25 de las 100 clases de las que disponemos, por tanto, el número de imágenes que usamos se reduce a 12500 imágenes.

Nuestra red BaseNet tiene como salida un vector de longitud 25 en el que cada posición indica la probabilidad de pertenencia a la clase correspondiente, luego, se manda este vector a un clasificador. Probamos a utilizar dos gradientes descendentes estocásticos (SGD) como clasificadores:

- Un SGD con *learning rate* 0.01, *decay* 10^{-6} y *momentum* 0.9.
- El algoritmo Adam.

Ambos clasificadores comparten una serie de parámetros. La métrica a utilizar será la *accuracy*, o precisión, es decir, el porcentaje de imágenes bien etiquetadas. La función de perdida que va a ser minimizada es la *categorical crossentropy*. Vamos a separar un 10% del conjunto de entrenamiento como conjunto de validación. El tamaño de los batchs es 64 y el número de épocas que se realizan 25.

Los resultados de BaseNet utilizando el SGD descrito son los siguientes:

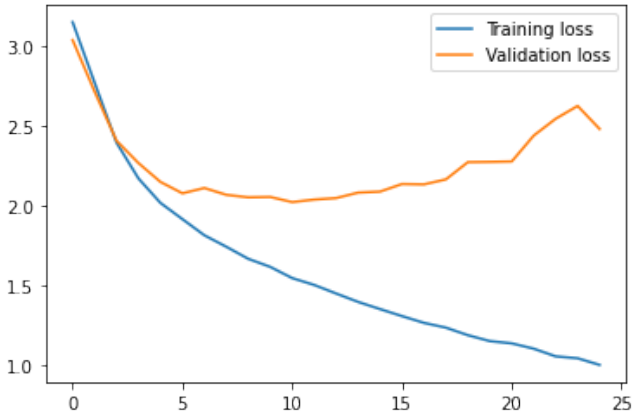

Figure 1: Pérdida BaseNet usando SGD

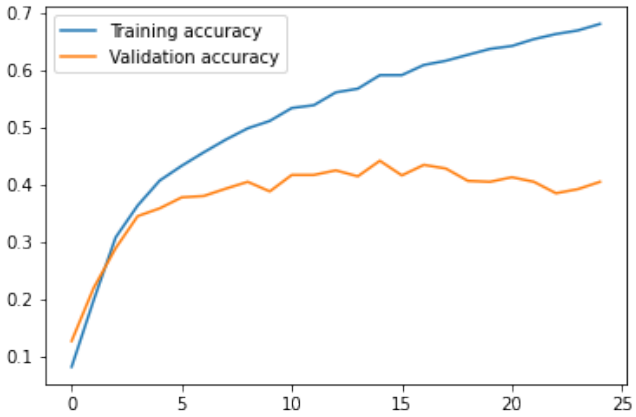

Figure 2: Precisión BaseNet usando SGD

Pérdida en el conjunto de test: 2.3230772018432617

Precisión en el conjunto de test: 0.4415999948978424

Los resultados de BaseNet utilizando Adam son los siguientes:

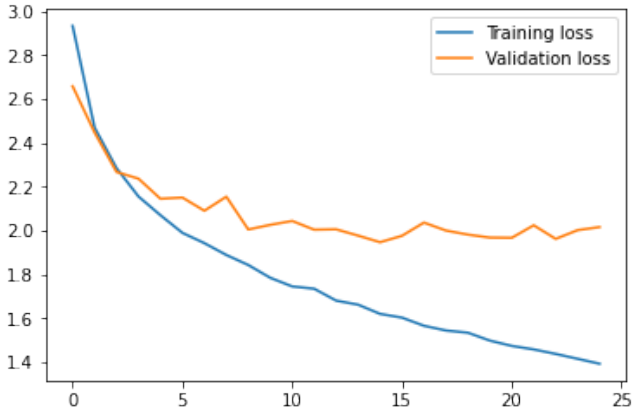

Figure 3: Pérdida BaseNet usando Adam

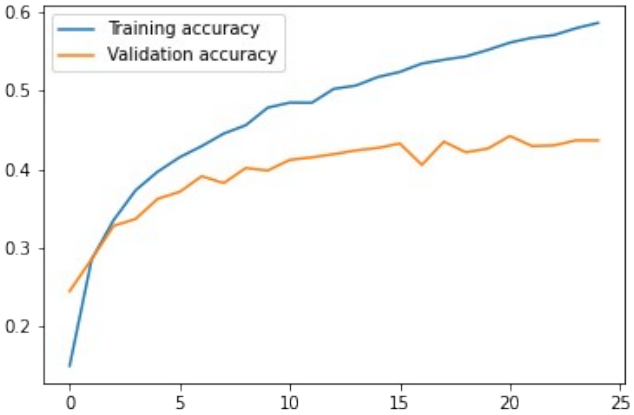

Figure 4: Precisión BaseNet usando Adam

Pérdida en el conjunto de test: 1.9802031517028809

Precisión en el conjunto de test: 0.451200008392334

Como podemos observar, el algoritmo Adam obtiene mejores resultados que el SGD descrito, es por esto, que vamos a utilizar Adam para todas las variantes posteriores de la red BaseNet. Aún así, la diferencia entre ambos clasificadores no es destacable.

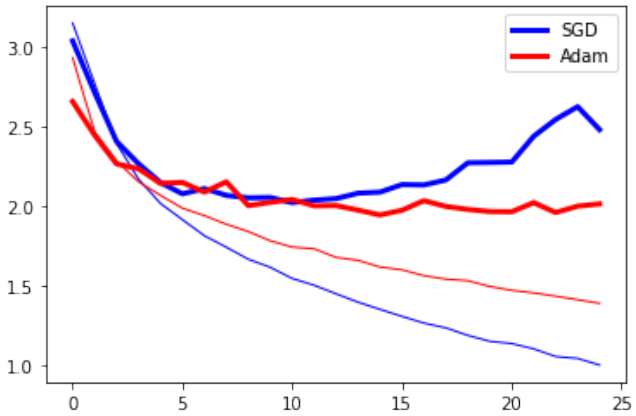

Figure 5: Comparativa de la pérdida

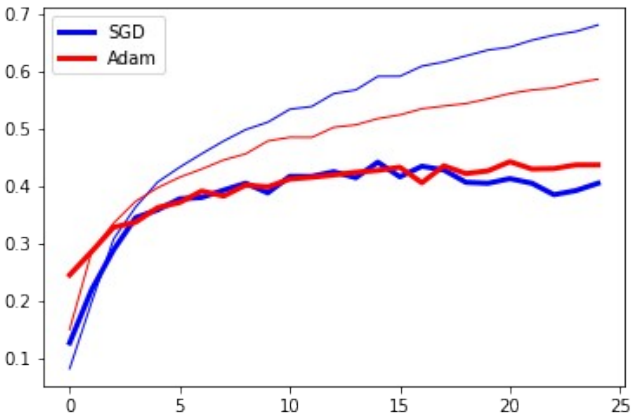

Figure 6: Comparativa de la precisión

Las líneas finas muestran los resultados en el conjunto de entrenamiento mientras que las gruesas muestran los resultados del conjunto de validación.

Cabe destacar que en ambos casos, se alcanza un punto en el que los resultados para el conjunto de validación se mantienen estables mientras que para el conjunto de entrenamiento mejoran, es decir, se produce *overfitting* para los dos clasificadores.

Ejercicio 2

En este ejercicio nos proponemos mejorar nuestra red BaseNet añadiendo nuevas capas y preprocesado de datos para así obtener un mejor porcentaje de aciertos en el conjunto de test. Nuestro objetivo será obtener al menos un 50% de aciertos en este conjunto.

Los experimentos realizados son acumulativos, es decir, la normalización de datos ha sido la primera mejora realizada lo que significa que en el resto de pruebas también se normalizan los datos antes de trabajar con ellos.

Normalización de datos

Comenzamos normalizando los datos. Para ello hacemos uso de la clase *ImageDataGenerator* de Keras. A esta clase le pasamos el conjunto de entrenamiento y hacemos que los parámetros *featurewise_cente* y *featurewise_std_normalization* sean *True* para que la media sea 0 y la varianza 1. Luego le pasamos el conjunto de test al método *standardize* para que realice las mismas operaciones en este conjunto. *ImageDataGenerator* también es utilizado para separar el conjunto de validación con el parámetro *validation_split*.

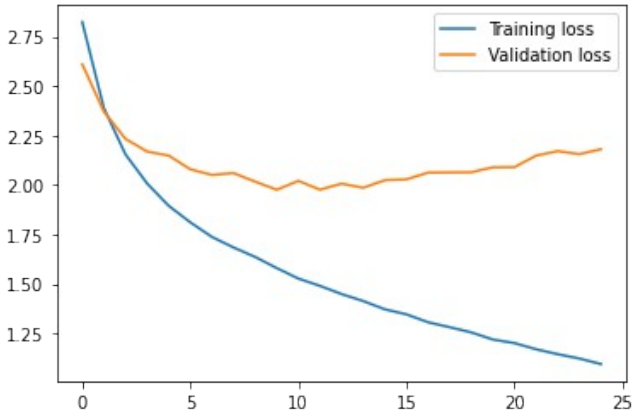

Figure 7: Pérdida con normalización

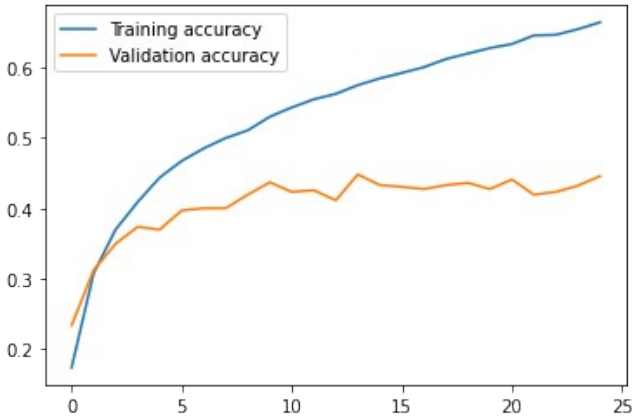

Figure 8: Precisión con normalización

Pérdida en el conjunto de test: 2.190531015396118

Precisión en el conjunto de test: 0.44279998540878296

Los resultados no han mejorado respecto a los obtenidos anteriormente, de hecho, han empeorado un poco, no obstante, no es una gran diferencia y siempre es bueno tener los datos normalizados.

Aumento de datos

Volvemos a configurar parámetros de la clase *ImageDataGenerator* [1] pero esta vez para crear nuevas imágenes de entrenamiento a partir de las que tenemos. Los parámetros utilizados son:

horizontal_flip = True	Voltea la imagen horizontalmente
zoom_range = 0.2	Añade un zoom en el rango [1-0.2, 1+0.2] a la imagen
width_shift_range = 0.1	Desplaza la imagen horizontalmente, como máximo un 10% de la anchura original.
height_shift_range = 0.1	Desplaza la imagen verticalmente, como máximo un 10% de la altura original.
rotation_range = 20	Gira la imagen en un rango de [-20, 20] grados

Claramente, queremos que estos cambios aleatorios que se realizan a las imágenes sólo ocurran en el conjunto de entrenamiento para mejorar la extracción de características de la red.

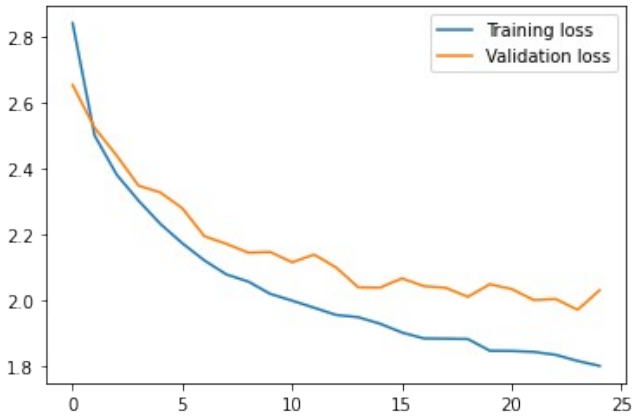

Figure 9: Pérdida con data augmentation

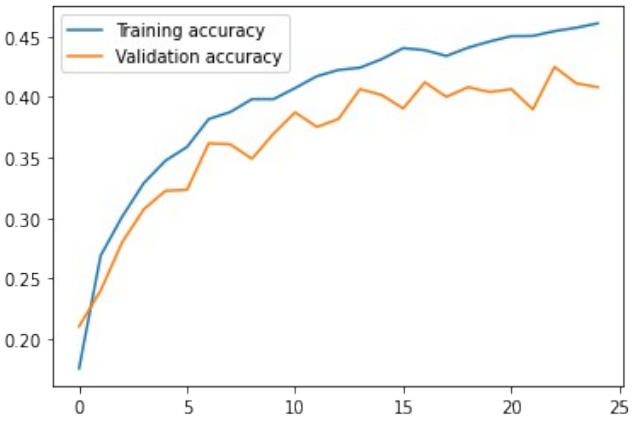

Figure 10: Precisión con data augmentation

Pérdida en el conjunto de test: 1.698938250541687

Precisión en el conjunto de test: 0.5012000203132629

Los resultados obtenidos son muy buenos. No solo hemos mejorado los resultados obtenidos en el conjunto de test, sino que también, hemos conseguido reducir el *overfitting* que se producía en las pruebas anteriores como era de esperar al aumentar el conjunto de entrenamiento.

Aunque ya hemos logrado llegar al 50% de aciertos, seguimos probando todas las mejoras propuestas en este apartado.

Batch Normalization

Comenzamos a añadir nuevas capas a la red BaseNet empezando con capas de *BatchNormalization*. Las introducimos justo después de las convoluciones y antes de las capas *ReLU* con el objetivo de que la media y la varianza queden cercanas a 0 y 1 respectivamente.

También se han realizado pruebas con las capas de *BatchNormalization* después de las capas de *ReLU* pero los resultados eran similares y se ha optado por la primera opción. Además, seguimos el ejemplo de las referencias consultadas [2][3] en las que estas capas se colocan justo después de las capas convolucionales.

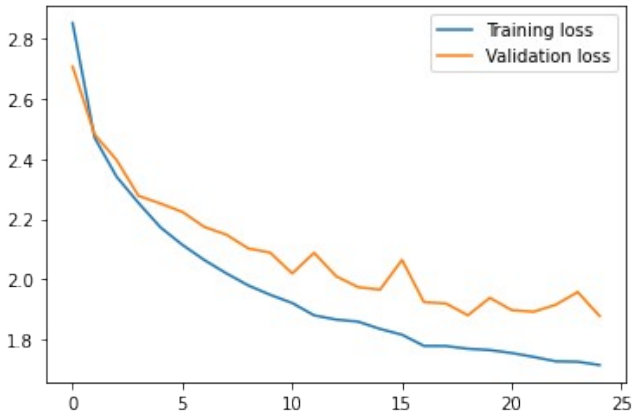

Figure 11: Pérdida con BatchNormalization

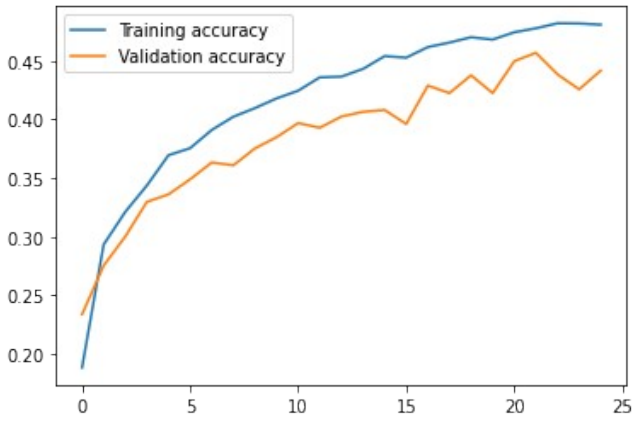

Figure 12: Precisión con BatchNormalization

Pérdida en el conjunto de test: 1.6722393035888672

Precisión en el conjunto de test: 0.49959999322891235

Los resultados son muy similares a los obtenidos en pruebas anteriores. Aun así, al igual que pasa con la normalización de datos, es bueno tener estas capas aunque no lleguen a mejorar los resultados.

Aumento de profundidad

Añadimos una capa de convolución antes de cada *MaxPooling*. Estas convoluciones tienen el objetivo de aumentar el número de filtros de los tensores antes de cada *pooling* multiplicando por 4 los filtros de las convoluciones que les preceden. El tamaño del *kernel* que se utiliza para ambas es 3 y se usa *padding = same* para no variar el tamaño de las imágenes. Por último, para que la primera capa *fully-connected* sigua recibiendo un vector de longitud 400 introducimos otra convolución con tamaño de *kernel* 1 cuyo objetivo es reducir el número de filtros de 64 a 16, tal y como era originalmente.

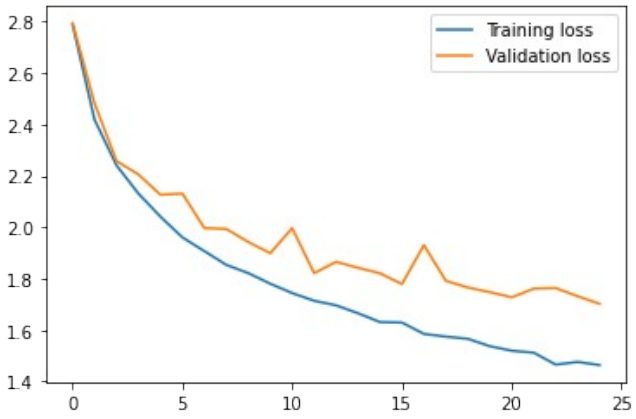

Figure 13: Pérdida con mayor profundidad

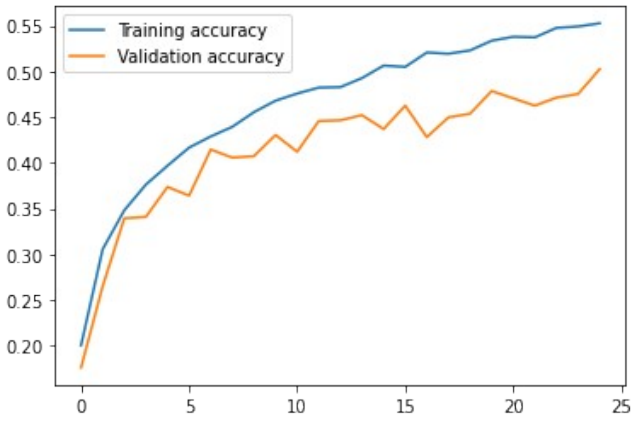

Figure 14: Precisión con mayor profundidad

Pérdida en el conjunto de test: 1.4685183763504028

Precisión en el conjunto de test: 0.5523999929428101

Como podemos ver, el aumento de profundidad de la red así como el incremento de filtros ha tenido un gran impacto en los resultados, pasando del 50% que obteníamos al 55% de aciertos que tenemos ahora.

Dropout

Probamos ahora a añadir *dropout* a las capas *fully-connected* para especializar las neuronas. La configuración que mejores resultados ha obtenido ha sido añadir un *dropout* del 20% en ambas capas.

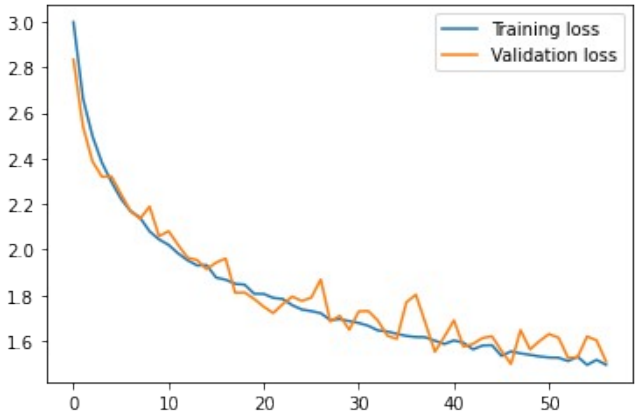

Figure 15: Pérdida con dropout

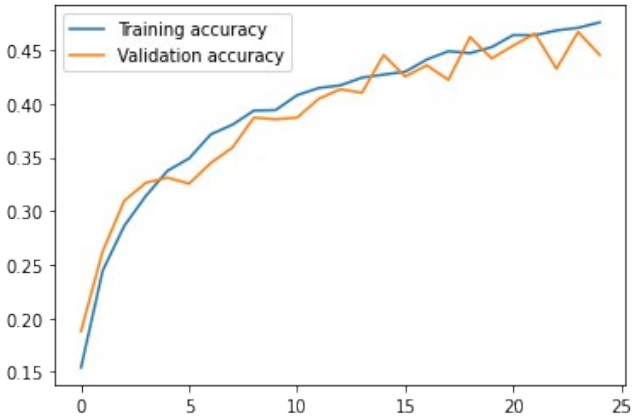

Figure 16: Precisión con dropout

Pérdida en el conjunto de test: 1.557310700416565

Precisión en el conjunto de test: 0.5267999768257141

Al añadir *dropout* hemos empeorado los resultados en el conjunto de test respecto a la prueba anterior aunque también, hemos logrado reducir el *overfitting* que se producía. Esto puede deberse a dos motivos, *BatchNormalization* puede sustituir a *dropout* [3] haciéndolo innecesario y al incluir ambos empeoramos la red, o bien, al introducir *dropout* ralentizamos el aprendizaje de la red al especializar las neuronas, por lo que tenemos que aumentar el número de épocas.

Early Stopping

Tenemos que aumentar el número de épocas para comprobar si *dropout* ralentiza el aprendizaje de nuestra red o directamente lo empeora. Aumentamos el número de épocas de 25 a 80. Vamos a ir comprobando en cada época si la pérdida y la precisión en el conjunto de validación mejoran. En el caso de que no haya mejoras durante 10 épocas paramos el aprendizaje y recuperamos el estado de la red en la que obtuvo los mejores resultados.

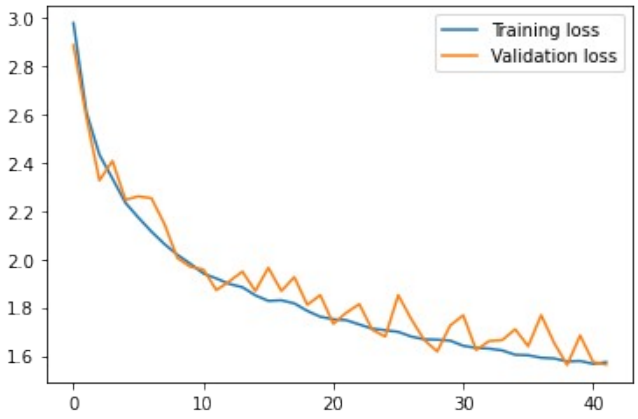

Figure 17: Pérdida con early stopping

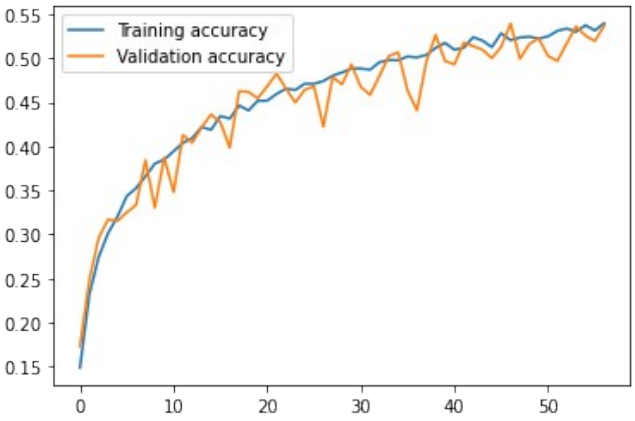

Figure 18: Precisión con early stopping

Pérdida en el conjunto de test: 1.319244623184204

Precisión en el conjunto de test: 0.5964000225067139

Claramente los resultados mejoran al aumentar el número de épocas que se realizan durante el entrenamiento. En este caso, la red se ha detenido en la época 57. Con estos resultados podemos concluir que efectivamente *dropout* ralentiza el aprendizaje aunque lo mejora. Aun así, queda la duda de si haciendo éste mismo experimento pero sin *dropout* obtendremos mejores resultados. Probamos *early stopping* sin *dropout:*

Pérdida en el conjunto de test: 1.2524240016937256

Precisión en el conjunto de test: 0.6140000224113464

Al quitar *dropout* obtenemos mejores resultados pero en este caso, se han realizado las 80 épocas, no ha habido *early stopping*.

Entonces, al añadir *dropout* hemos conseguido que se necesite menos tiempo para llegar al mejor estado de la red, es decir, necesitamos más épocas para mejorar los resultados pero llegamos al mejor estado antes que si no lo tuviéramos. Por otro lado, la red sin *dropout* todavía puede seguir aprendiendo más, ya que no se ha producido *early stopping,* mejorando aún más sus resultados.

En conclusión, para este ejercicio, debido a que el tiempo de ejecución es importante para nosotros, nos quedamos con la versión que si tiene *dropout* aunque obtengamos resultados un poco peores. Por tanto, la tabla que define al modelo con todos los cambios aplicado es el siguiente:

Layer No.	Layer Type	Kernel Size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Conv2D	5	32 | 28	3 | 6
2	BatchNormalization	-	28 | 28	-
3	Relu	-	28 | 28	-
4	Conv2D	3	28 | 28	6 | 24
5	BatchNormalization	-	28 | 28	-
6	Relu	-	28 | 28	-
7	MaxPooling2D	2	28 | 14	-
8	Conv2D	5	14 | 10	24 | 16
9	BatchNormalization	-	10 | 10	-
10	Relu	-	10 | 10	-
11	Conv2D	3	10 | 10	16 | 64
12	BatchNormalization	-	10 | 10	-
13	Relu	-	10 | 10	-
14	MaxPooling2D	2	10 | 5	-
15	Conv2D	1	5 | 5	64 | 16
16	BatchNormalization	-	5 | 5	-
17	Relu	-	5 | 5	-
18	Dropout (0.2)	-	5 | 5	-
19	Linear	-	400 | 50	-
20	BatchNormalization	-	50 | 50	-
21	Relu	-	50 | 50	-
22	Dropout (0.2)	-	50 | 50	-

23	Linear	-	50 | 25	-

Ejercicio 3

En este ejercicio vamos a utilizar la red ResNet50 pre-entrenada con *ImageNet* como extractor de características para el conjunto de datos *Caltech-UCSD* [4] que contiene 6033 imágenes de 200 especies de pájaros distintas. También re-entrenaremos la red para clasifique directamente las imágenes de esta base de datos, es decir, haremos *fine tunning*.

ResNet50 como extractor de características

Eliminamos la capa de salida de ResNet50 para que al acabar nos devuelva un vector de longitud 2048. Luego, pasamos este vector a un modelo que será el encargado de clasificar las imágenes.

Para el primer caso, definimos un modelo simple en el que sólo hay una capa *fully connected* con 200 neuronas y que nos devuelve un vector con las probabilidades de que la imagen pertenezca a cada clase, y que luego utiliza el clasificador. La tabla del modelo es la siguiente:

Layer No.	Layer Type	Kernel Size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Linear	-	2048 | 200	-

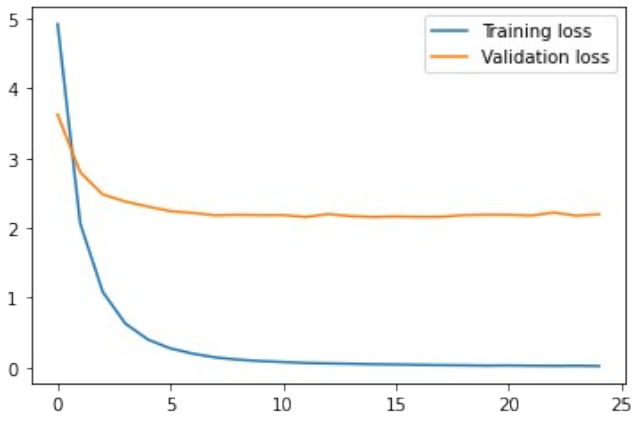

Figure 19: Pérdida cambiando la capa de salida de ResNet50

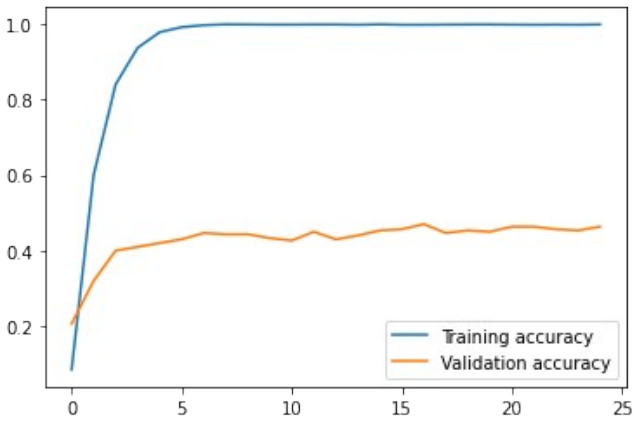

Figure 20: Precisión cambiando la capa de salida de ResNet50

Pérdida en el conjunto de test: 2.349853754043579

Precisión en el conjunto de test: 0.4385097324848175

Podemos observar que se produce un gran sobreajuste rápidamente y que a partir de las época 4-5 ya no hay ninguna mejora en el conjunto de validación

Vamos ahora a añadir ahora dos capas *fully conected* más con un 40% de *dropout* cada una para ver si podemos mejorar el modelo simple.

Layer No.	Layer Type	Kernel Size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Dropout (0.4)	-	2048 | 2048	-
2	Linear	-	2048 | 1024	-
3	BatchNormalization	-	1024 | 1024	-
4	Relu	-	1024 | 1024	-
5	Dropout (0.4)	-	1024 | 1024	-
6	Linear	-	1024 | 512	-
7	BatchNormalization	-	512 | 512	-
8	Relu	-	512 | 512	-
9	Linear	-	512 | 200	-

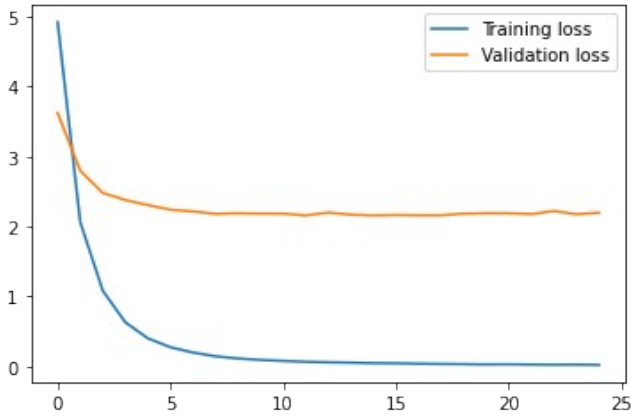

Figure 21: Pérdida añadiendo capas FC al modelo simple

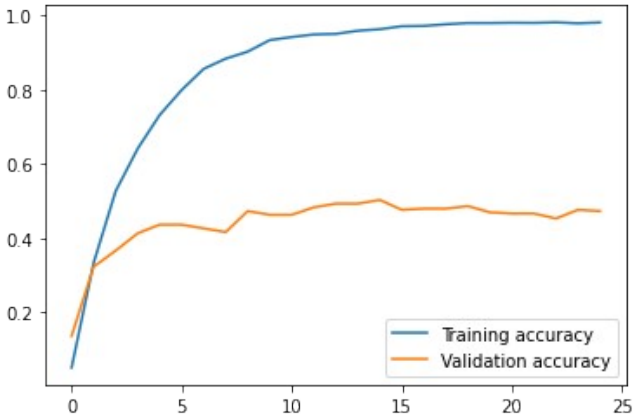

Figure 22: Precisión añadiendo capas FC al modelo simple

Pérdida en el conjunto de test: 2.340416669845581

Precisión en el conjunto de test: 0.4493900537490845

Aunque ha habido un ligero incremento en el porcentaje de aciertos, el *overfitting* que se produce sigue siendo similar, después de unas pocas épocas ya no hay mejoras en el conjunto de validación. Se ha probado con distintas configuraciones de capas *fully connected* y distintos valores de *dropout* pero todos los resultados han sido semejantes a estos.

Por último, además de quitar la capa de salida de ResNet50 vamos a quitar la capa de *AveragePooling* y la sustituimos por una capa de convolución.

Layer No.	Layer Type	Kernel Size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Conv2D	3	7 | 7	2048 | 64
2	BatchNormalization	-	7 | 7	-
3	Relu	-	7 | 7	-
4	Dropout (0.4)	-	7 | 7	-
5	Linear	-	3136 | 1024	-
6	BatchNormalization	-	1024 | 1024	-
7	Relu	-	1024 | 1024	-
8	Dropout (0.4)	-	1024 | 1024	-
9	Linear	-	1024 | 512	-
10	BatchNormalization	-	512 | 512	-
11	Relu	-	512 | 512	-
12	Linear	-	512 | 200	-

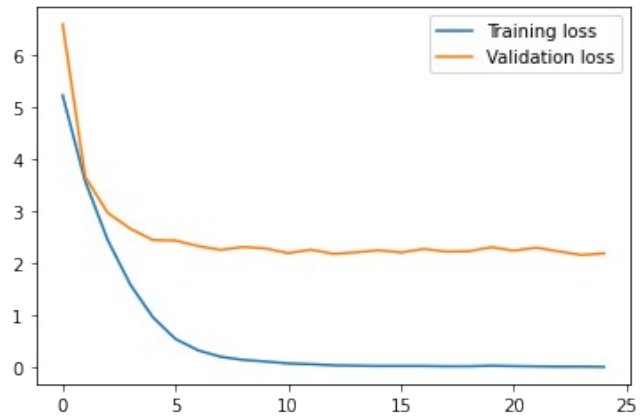

Figure 23: Pérdida añadiendo una capa de convolución al segundo modelo

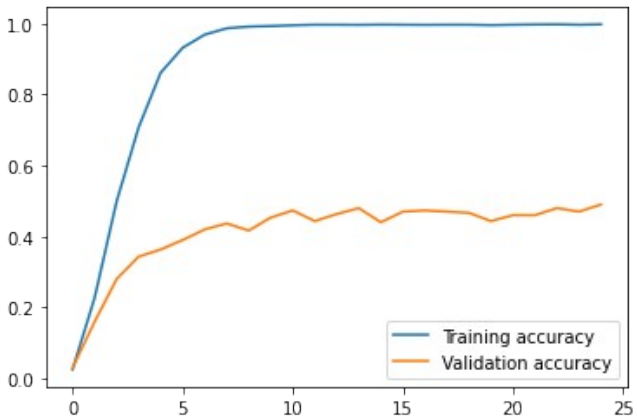

Figure 24: Precisión añadiendo una capa de convolución al segundo modelo

Pérdida en el conjunto de test: 2.577955961227417

Precisión en el conjunto de test: 0.381140798330307

El *overfitting* se mantiene y los resultados empeoran. En un principio es mejor dejar la capa de *AveragePooling* aunque es posible que implementando un modelo lo suficientemente sofisticado se

puedan mejorar los resultados del modelo anterior. Aun así, estaríamos complicando en exceso el modelo y aumentando el tiempo de cómputo necesario al sustituir una capa de *AveragePooling* por varias convoluciones sin garantías de que fuésemos a mejorar los resultados.
En cualquier caso, en los experimentos realizados, no se ha conseguido encontrar una configuración de capas de convolución que llegase a obtener resultados similares al modelo con varias capa *fully connected* que mantiene la capa de *AveragePooling*.

Fine Tunning

Vamos ahora a re-entrenar la red ResNet50 para que funcione con el conjunto de datos *Caltech-UCSD*. Para ello, sustituimos la capa de salida por una capa *fully connected* de 200 neuronas y entrenamos ResNet50 pasándole el nuevo conjunto de datos. Ya que se requiere mucho tiempo de ejecución por época vamos a entrar el modelo durante sólo 10 épocas.

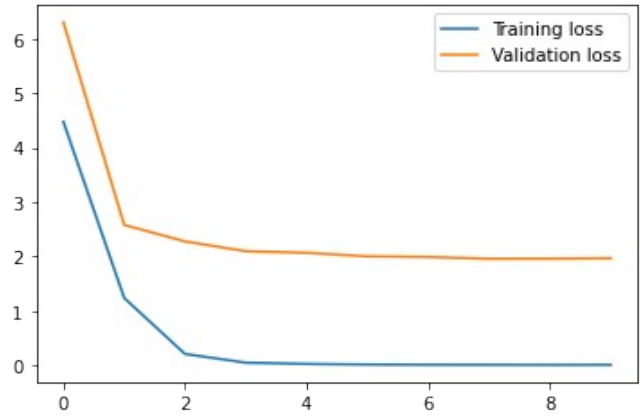

Figure 25: Pérdida añadiendo una capa de convolución al segundo modelo

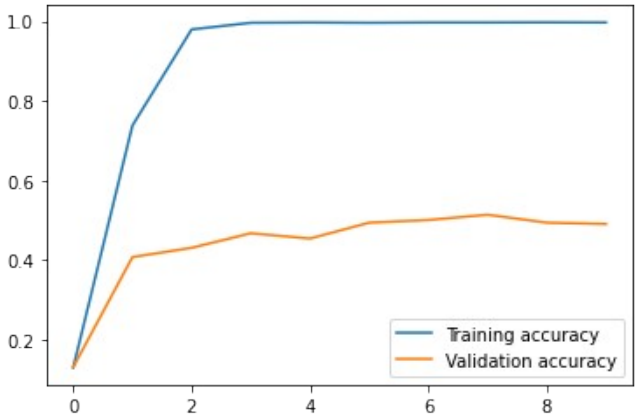

Figure 26: Precisión añadiendo una capa de convolución al segundo modelo

Pérdida en el conjunto de test: 2.258345603942871

Precisión en el conjunto de test: 0.4477415084838867

Vemos que obtenemos un resultado similar al que obteníamos en el segundo caso usando ResNet50 como extractor de características. Tanto la precisión como la pérdida obtenidas en el conjunto de test son muy parecidas y se produce *overfitting* incluso más rápido que antes.

Cabe destacar que el algoritmo Adam no obtenía buenos resultados y sólo para este apartado se ha utilizado el SGD descrito anteriormente como clasificador.

Teniendo en cuenta los resultados y los tiempos de ejecución, para este problema es mejor utilizar ResNet50 como extractor de características y un modelo sencillo con unas pocas capas *fully connected*.

Bonus

Volvemos con la red BaseNet y partiendo del modelo con todas las mejoras implementadas vamos a intentar conseguir mejores resultados.

Hemos obtenido buenos resultados al aumentar el número de filtros que tenemos. Entonces, el primer cambio que realizamos es aumentar los filtros iniciales a 32, en vez de 6. Además, después de realizar *pooling* vamos a duplicar el número de filtros para mantener la misma carga computacional.

Las convoluciones originales con tamaño de *kernel* 5 ahora no reducirán el tamaño de las imágenes. Las dimensiones de una imagen son 32x32, al ser pequeñas puede que perder 4 píxeles en anchura y altura, sobretodo al principio de la red, nos empeore el aprendizaje.

Al aumentar la profundidad en el ejercicio 2, añadimos una capa convolucional antes de hacer *MaxPooling*, volvemos a hacer lo mismo de forma que tenemos dos convoluciones con tamaño de *kernel 3*, pero en este caso no aumentamos el número de filtros.

Por último, ahora podemos modificar el número de neuronas en la primera capa *fully connected* por lo que ya no es necesaria la convolución que reducía los filtros antes de esta. Incrementamos el número de neuronas a 512. Además, al tener más neuronas podemos incrementar el *dropout* y siguiendo el ejemplo de *mnist* proporcionado, introducimos *dropout* del 25% para la primera capa *fully connected* y 50% para la segunda.

Los resultados obtenidos son los siguientes:

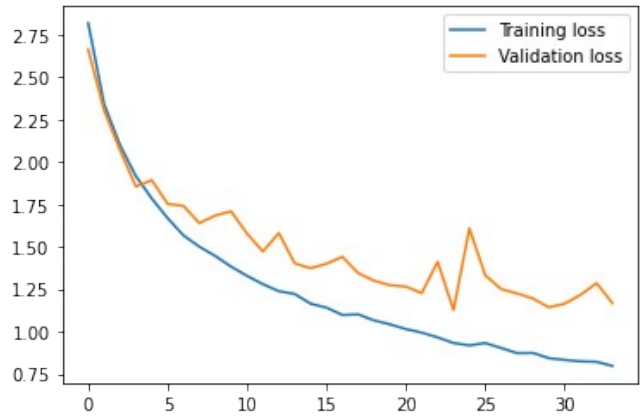

Figure 27: Pérdida BaseNet bonus

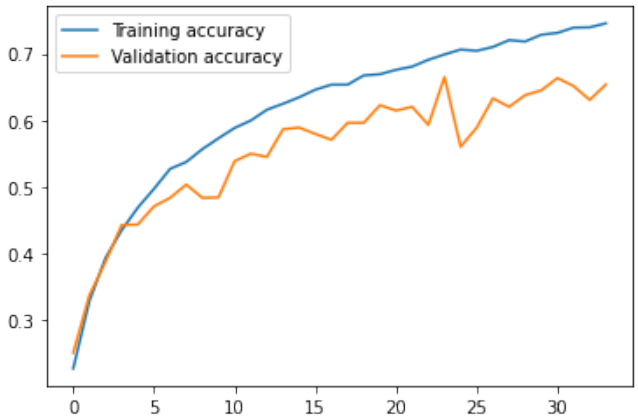

Figure 28: Precisión BaseNet bonus

Pérdida en el conjunto de test: 0.9592517614364624

Precisión en el conjunto de test: 0.7027999758720398

Como podemos comprobar, los resultados son mucho mejores a los obtenidos en el ejercicio 2, pasando de 59.6% de aciertos a 70.2%. Además, hemos conseguido reducir el tiempo de aprendizaje,

necesitando menos de 40 épocas comparadas a las 57 que se necesitaban antes. Simplemente teniendo una buena combinación de capas hemos logrado obtener resultados mucho mejores.

Aun así, todavía podemos implementar más mejoras al modelo. Vamos a pasar de utilizar un modelo plano a uno residual [2], en el que las dos convoluciones con *kernel* de tamaño 3 que realizamos antes del *pooling* van convertirse en un módulo residual, es decir, antes de la última capa de *BatchNormalization* más *ReLU* sumamos la entrada que teníamos antes de la primera convolución.

Los resultados obtenidos son los siguientes:

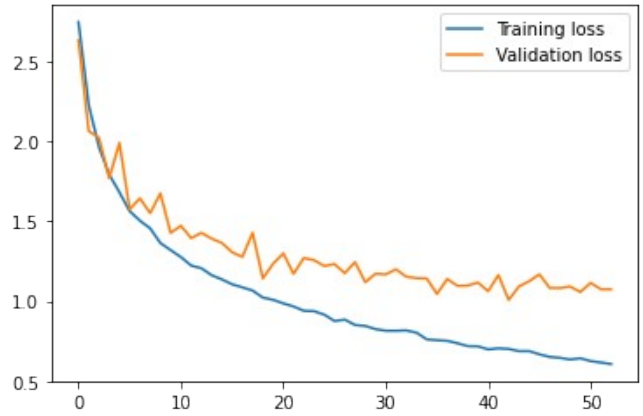

Figure 29: Pérdida BaseNet Residual

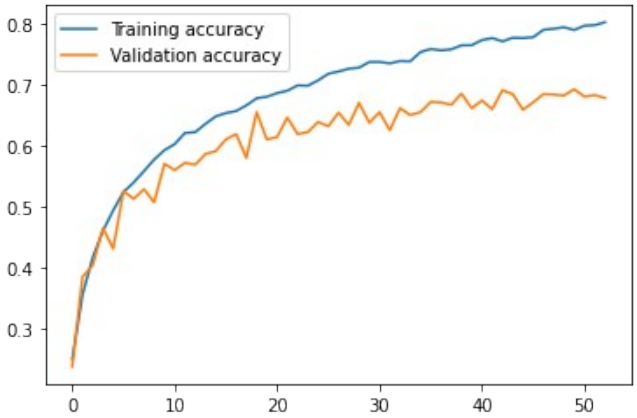

Figure 30: Precisión BaseNet Residual

Pérdida en el conjunto de test: 0.8284435272216797

Precisión en el conjunto de test: 0.745199978351593

Vemos que al añadir un cambio tan simple, hemos introducido dos capas de suma de tensores, logramos mejorar la precisión del modelo. Cabe destacar que se han necesitado más épocas para entrenar el modelo con este cambio.

Todavía se puede mejorar el rendimiento de BaseNet por ejemplo sustituyendo los módulos residuales por módulos residuales con *bottleneck* [2], o simplemente aumentando el número de módulos que tenemos. También sería interesante cambiar los módulos residuales por módulos con *self-calibration* [5] para ver con que opción se obtiene mejores resultados.

Hay muchos cambios posibles que se podrían realizar además de estos mencionados pero, con el objetivo de mantener la simplicidad del modelo y no aumentar en exceso los tiempos de ejecución, no se han realizado.

Entonces, la tabla que define nuestro modelo es la siguiente:

Layer No.	Layer Type	Kernel Size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Conv2D	5	32 | 32	32 | 32
2	BatchNormalization	-	32 | 32	-
3	Relu	-	32 | 32	-
4	Conv2D	3	32 | 32	32| 32
5	BatchNormalization	-	32 | 32	-
6	Relu	-	32 | 32	-
7	Conv2D	3	32 | 32	32| 32
8	Add	-	32 | 32	-
9	BatchNormalization	-	32 | 32	-
10	Relu	-	32 | 32	-
11	MaxPooling2D	2	32 | 16	-
12	Conv2D	5	16 | 16	32 | 64
13	BatchNormalization	-	16 | 16	-
14	Relu	-	16 | 16	-
15	Conv2D	3	16 | 16	64 | 64
16	BatchNormalization	-	16 | 16	-
17	Relu	-	16 | 16	-
18	Conv2D	3	16 | 16	64 | 64
19	Add	-	16 | 16	-
20	BatchNormalization	-	16 | 16	-
21	Relu	-	16 | 16	-
22	MaxPooling2D	2	16 | 8	-
23	Dropout (0.25)	-	8 | 8	-
24	Linear	-	4096 | 512	-
25	BatchNormalization	-	512 | 512	-
26	Relu	-	512 | 512	-
27	Dropout (0.5)	-	512 | 512	-
28	Linear	-	512 | 25	-

Referencias

[1] Data Augmentation con Keras https://fairyonice.github.io/Learn-about-ImageDataGenerator.html

[2] Deep Residual Learning For Image Recognition https://arxiv.org/abs/1512.03385

[3] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift https://arxiv.org/abs/1502.03167

[4] Caltech-UCSD Birds 200 http://www.vision.caltech.edu/visipedia/CUB-200.html

[5] Improving Convolutional Networks With Self-Calibrated Convolutions https://openaccess.thecvf.com/content_CVPR_2020/html/Liu_Improving_Convolutional_Networks_With_Self-Calibrated_Convolutions_CVPR_2020_paper.html