UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Prácticas de redes neuronales

Fundamentos de los Sistemas Inteligentes

Isac Añor Santana

Raúl Mateus Sánchez

Tabla de contenido

Tabla de ilustraciones

Introducción

Este documento es una memoria donde se resaltan los diferentes resultados obtenidos. Primero se busca un modelo adecuado y después continuamos modificando hiperparámetros de ese modelo para obtener los mejores resultados posibles.

Todas las ejecuciones se encuentran recogidas en este repositorio de GitHub:

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores

El repositorio tiene 7 ramas, una para cada una de las versiones que se han hecho. Los archivos que hay por rama dependen de las versiones. Por ejemplo, adelantando el contenido del documento, se prueban varios modelos cuyo funcionamiento ha sido demostrado, esto modelos son: AlexNet, VGG16 y GoogleNet. Dado que la modificación de hiperparámetros para estos modelos es relativamente complicada, lo que se ha hecho ha sido variar los optimizadores, por lo tanto, las ramas correspondientes a estos modelos contienen las diferentes ejecuciones con los optimizadores seleccionados.

Dataset

El dataset empleado es el siguiente: https://www.kaggle.com/alxmamaev/flowers-recognition .

Es un dataset que trata sobre el reconocimiento de flores. Hay cinco clases: margaritas, dientes de león, rosas, girasoles y tulipanes. Cuenta un total de 4317 imágenes. La razón por la que se eligió este dataset reside principalmente en la gran cantidad de muestras y en que se hizo en el formato especificado en https://nbviewer.org/url/cayetanoguerra.github.io/ia/nbpy/redneuronal7.ipynb. Cuenta con una carpeta para cada una de las cinco clases mencionadas, permitiendo usar las funciones de alto nivel que generan objetos de tipo dataset con facilidad, puesto que esta estructura de directorios es la adecuada.

Primera versión – planteamiento del código

El motivo de esta sección es comentar de dónde se ha partido para la realización de la práctica, así como observaciones y cambios en el código.

La primera versión, que se encuentra en esta rama del repositorio,

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_1_piloto/150_epochs_con_fit_generator.ipynb

adapta principalmente el código del documento "Redes neuronales 7 + Enunciado práctica". Los únicos cambios en el código residen principalmente en la carga del dataset, y un ligero cambio en la última capa del modelo reduciendo el número de neuronas a 5, puesto que hay 5 clases.

Sin embargo, para nuestra sorpresa, nos encontramos con el siguiente mensaje de error, o más bien aviso:

```
"/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10:
UserWarning: `Model.fit_generator` is deprecated and will be removed in a
future version. Please use `Model.fit`, which supports generators.
  # Remove the CWD from sys.path while we load stuff."
```

Además, el tiempo por época era más largo que en los códigos ejecutados para los datasets de MNIST, FASHION MNIST, y el de lenguaje de signos; por lo que decidimos continuar con la práctica adaptando el código que se encuentra en "Práctica 2: Red convolutiva en Keras".

En esta primera versión se terminan las 150 épocas con un 66.55% de exactitud en el dataset de validación.

Para una visualización del modelo que se usa en cada una de las versiones, se utiliza la función "plot_model" de la librería "keras.utils.vis_utils". Un ejemplo de visualización del modelo es el que se encuentra en la siguiente página. Para no sobrecargar este documento con representaciones del modelo, no se agregarán más de estas imágenes. Dicha representación está visible en el código empleado para cada versión en el repositorio.

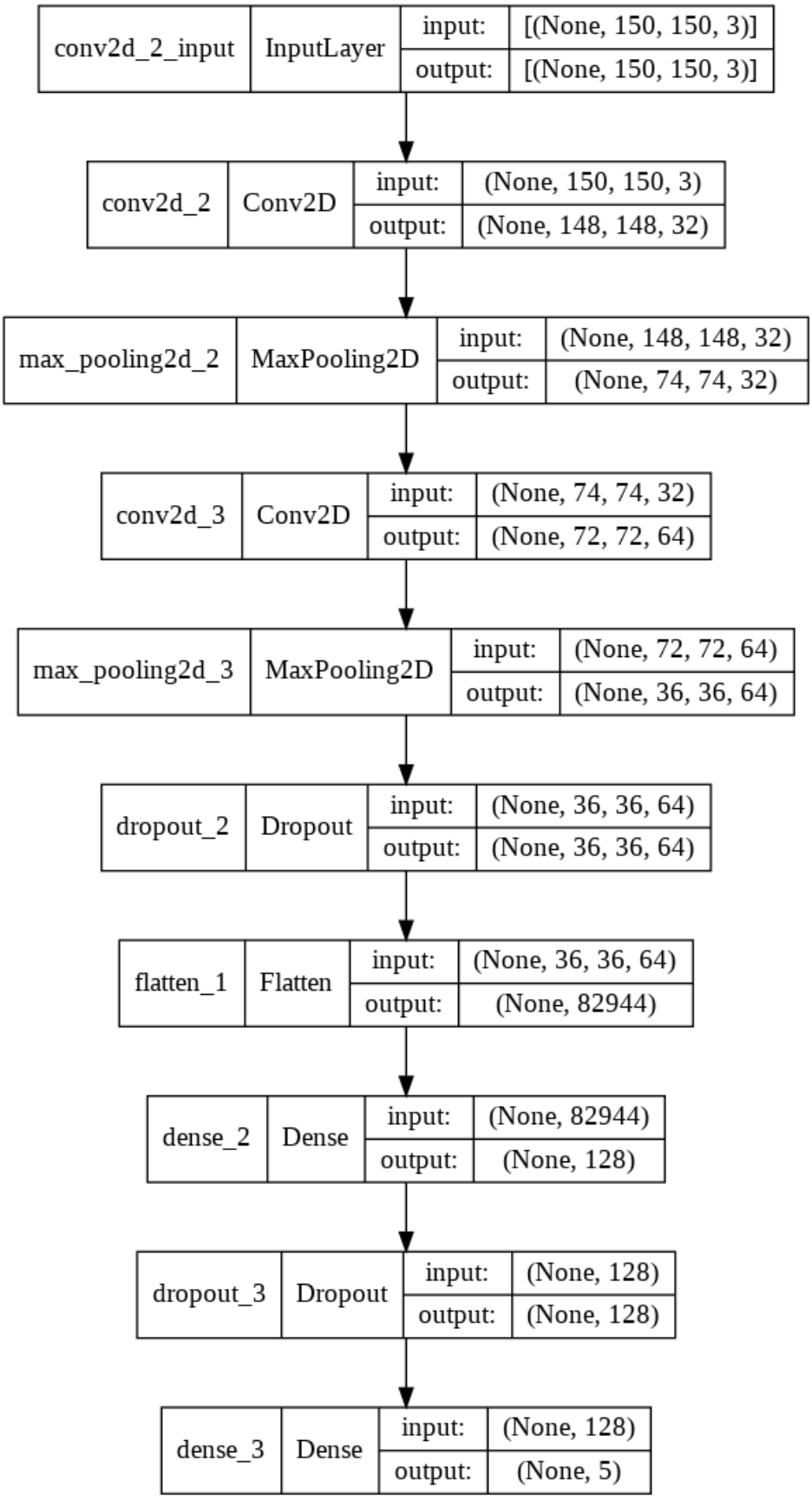


Ilustración 1 - Visualización del primer modelo

Segunda version – modelo básico

Esta segunda version adapta el código comentado en la sección anterior. El enlace es el siguiente https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_2_Base/Version_2.ipynb y tiene como mensaje de commit “Version 1”.

Los aspectos principales para comentar acerca de esta ejecución, es que vemos como obtiene mejores resultados que la version anterior en menos épocas y como tiene lugar una parada anticipada (Early Stopping). Esto se debe a que las últimas diez épocas tenían unas tasas de exactitud lo suficientemente similares para determinar que no va a haber cambios significativos y por lo tanto se para la ejecución. He comentado las últimas diez épocas, esto es porque el número de épocas suficientemente similares para que pare la ejecución debe ser diez, valor pasado como parámetro de tolerancia a la hora de instanciar el “EarlyStopping”.

Con respecto a la interpretación de los resultados, se destaca la diferencia entre la tasa de exactitud de entrenamiento y la de validación. La de entrenamiento es significativamente superior a la de validación, lo que implica que es bastante posible que estemos ante un caso de sobreajuste. Es decir, que la red “memoriza” imágenes del conjunto de entrenamiento y falla cuando se le presentan nuevos.

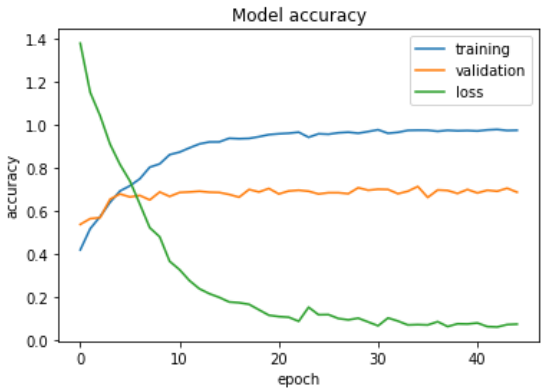


Ilustración 2 - Segunda versión - tasas del modelo

En la matriz de confusión se observa como las clases que más confunde la red son rosas y tulipanes (2 y 4) y margaritas y dientes de león (0 y 1).

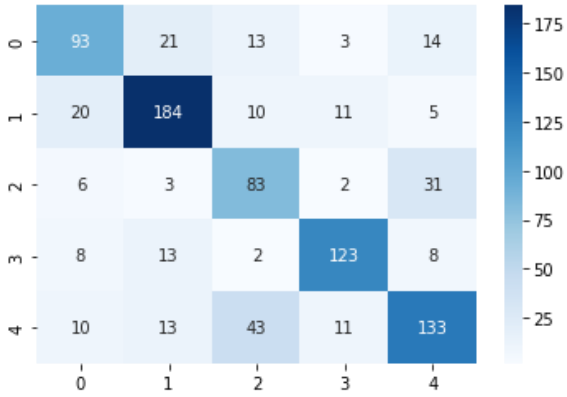


Ilustración 3 - Segunda versión - matriz de confusión

Tercera versión – AlexNet

Introducción

Es a partir de esta version donde probamos con diferentes hiperparámetros y comprobamos resultados. Sin embargo, antes que ponernos a añadir y eliminar capas de forma aleatoria, decidimos, siguiendo la información que se encuentra en Redes neuronales 9 observar el funcionamiento de redes han demostrado ser buenas a la hora de clasificar imágenes, implementándolas en este código.

En esta version, como dice el título de la sección, hemos cambiado la celda del modelo de tal forma que concuerde con el modelo de capas que propone AlexNet. Hemos visto implementaciones por internet (https://towardsdatascience.com/implementing-alexnet-cnn-architecture-using-tensorflow-2-0-and-keras-2113e090ad98) para hacernos una idea de cómo hacerlo, y nos hemos encontrado con una serie de parámetros y funciones nuevas cuyo funcionamiento vamos a comentar.

Stride

Lo primero que vamos a comentar es el parámetro llamado stride, o "paso" en español. El material empleado para comprender que es ha sido sacado del siguiente enlace:

https://towardsdatascience.com/visualizing-the-fundamentals-of-convolutional-neural-networks-6021e5b07f69

El artículo es una introducción a las redes neuronales convolucionales y cuenta con ejemplos visuales, que ayudan a *ver* qué es lo que realmente está ocurriendo. Resumiendo, sabemos que una capa convolucional extrae características de una imagen aplicando la operación de convolución sobre ésta con un tamaño del kernel determinado. El kernel sería la característica a extraer o destacar y cada región de la imagen original en cada iteración donde se aplica la operación de convolución se denomina campo receptivo (receptive field). Con estos conceptos claros, definimos el paso o stride como la distancia entre campos receptivos. En los ejemplos que hemos visto en clase, con un kernel de 3x3 que se aplica para cada píxel no borde o esquina de la imagen, tenemos que entre operaciones de convolución se solapan 6 píxeles de la imagen original. Si se adopta un paso de dos en este kernel 3x3, se solaparían solo una fila y una columna en campos receptivos adyacentes. Es importante destacar que el solapamiento es necesario para garantizar que los campos receptivos no se salten información importante. Aumentar el paso reduce el coste computacional de las convoluciones. Se pasa como parámetro una tupla de dos enteros especificando el paso en el ancho y el alto.

Batch normalization

De las fuentes que se citan abajo en la Webgrafía llegamos a la conclusión que Batch normalization, o batch norm es una técnica que ayuda a coordinar la actualización de las múltiples capas de un modelo. Lo hace escalando la salida de una capa, estandarizando las activaciones de cada entrada variable por mini-batch, así como las activaciones de un nodo de la capa anterior. El proceso también es llamado "blanqueo" cuando se aplica a imágenes en el ámbito de visión por computador. Tiene el efecto de estabilizar el proceso de aprendizaje y reduce dramáticamente el número de épocas de entrenamiento.

Padding

Significa rellenar y puede tener dos valores, "valid" que implica que no hay relleno o "same" que rellena con ceros la salida para que sea de las mismas proporciones que la entrada.

Formato de los resultados

Hemos implementado el modelo y lo hemos ejecutado con distintos optimizadores, pues los resultados variaban bastante en función del optimizador. Por lo tanto, se comentarán brevemente los resultados con cada optimizador. Además, nos hemos quedado sin cuota de uso de GPU en el Google Collab, por lo que hay algunas ejecuciones (la mayoría) hechas en Kaggle (para nuestra sorpresa, con ejecuciones más rápidas que en el Collab, lo que implican que nos "prestan mejores GPUs). Las secciones tendrán el nombre del optimizador seguido de un guion y el sitio donde se ejecutó (Collab o Kaggle).

Adam – Kaggle

En el gráfico de tasas de exactitud se observa una alta tasa de aprendizaje[1]. Hay sobreajuste y la ejecución termina con EarlyStopping. La mejor tasa de exactitud de entrenamiento fue 0.9519 y de validación, 0.7393 en la misma época

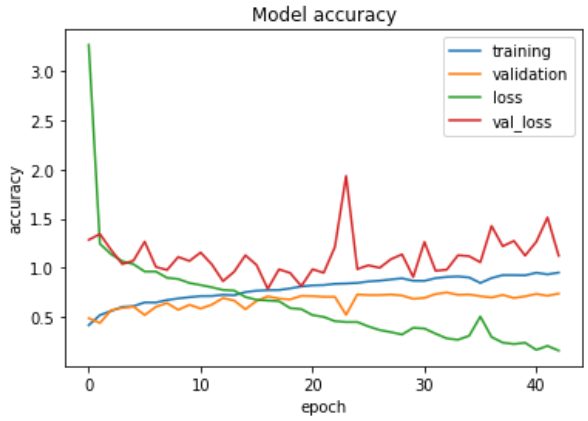


Ilustración 4 - Exactitud del modelo - AlexNet - Adam

La matriz de confusión mejora con respecto a la ejecución anterior, habiendo menos predicciones fuera de la diagonal. Se observa que donde más se equivoca es prediciendo tulipanes (4) como rosas (3).

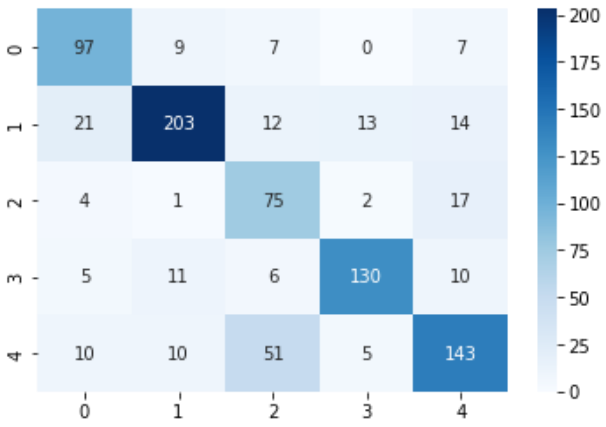


Ilustración 5 - Matriz de confusión - AlexNet - Adam

[1] https://nbviewer.org/url/cayetanoguerra.github.io/ia/nbpy/redneuronal8.ipynb

Adadelta - Kaggle

En este optimizador se observa un menor sobreajuste y unos descenso más progresivos y menos violentos (menos picos) que con respecto al optimizador anterior. Tienen unas tasas de aprendizaje menores. La mejor tasa de exactitud de entrenamiento fue 0.7568 y de validación, 0.6501 en distintas épocas.

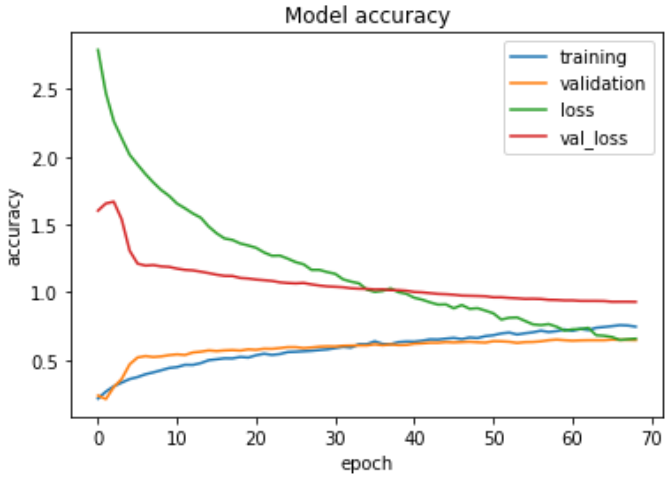


Ilustración 6 – Exactitud del modelo – AlexNet – Adadelta

Se observa una matriz de confusión muy similar a Ilustración 3 - Segunda versión - matriz de confusión, solo que incluso con más error. Se observa como también confunde dientes de león (1) y girasoles (3).

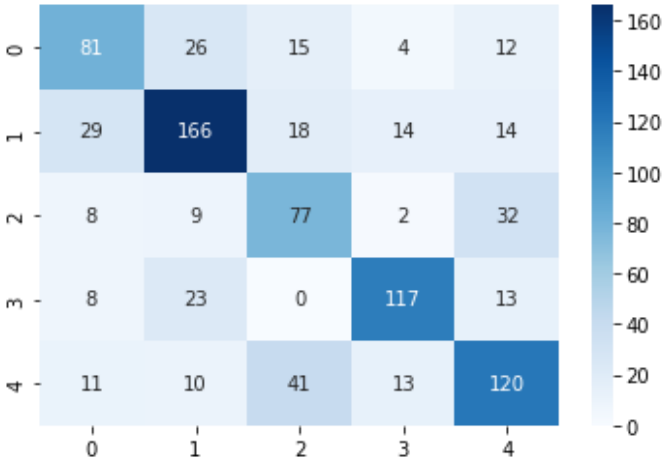


Ilustración 7 - Matriz de confusión - AlexNet - Adadelta

Adagrad – Kaggle

Para este optimizador tenemos una buena tasa de aprendizaje. Hay un importante sobreajuste. La mejor tasa de exactitud de entrenamiento fue 0.9861 y de validación, 0.7277 en distintas épocas.

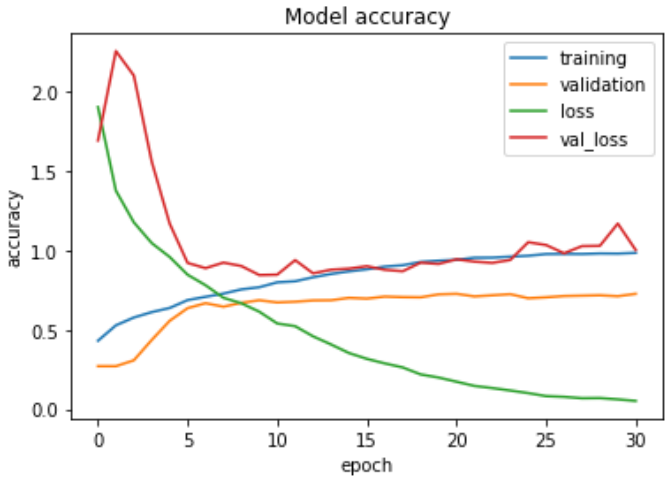


Ilustración 8 - Exactitud del modelo - AlexNet – Adagrad

La matriz de confusión es muy similar a las anteriores, quizás con los errores más distribuidos.

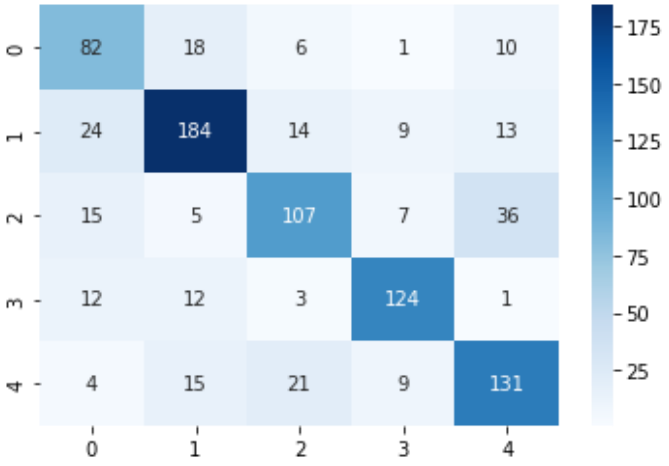


Ilustración 9 - Matriz de confusión - AlexNet - Adagrad

SGD – Kaggle

Para este optimizador se observa una buena tasa de aprendizaje y sobreajuste. La mejor tasa de exactitud de entrenamiento fue 0.9916 y de validación, 0.7462 en distintas épocas.

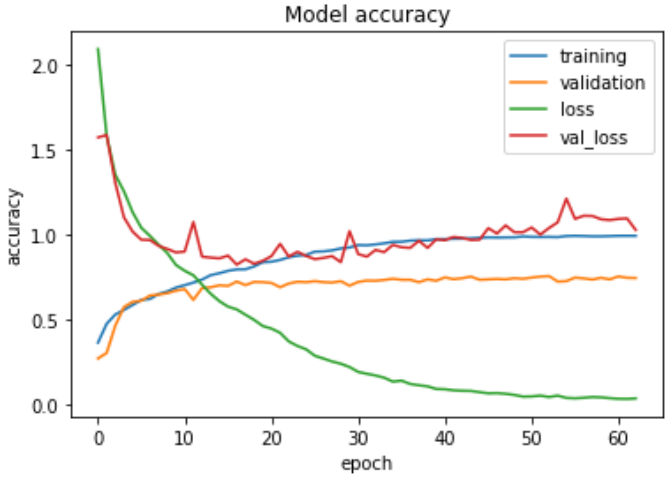


Ilustración 10 - Exactitud del modelo - AlexNet - SGD

La matriz de confusión es similar a las anteriores, no hay nada nuevo que comentar.

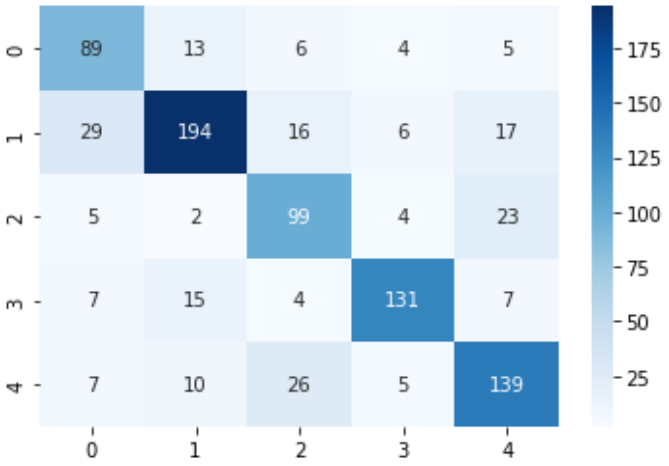


Ilustración 11 - Matriz de confusión - AlexNet - SGD

Cuarta versión – VGG16

En este caso, también se ha probado este modelo con los cuatro optimizadores que hemos visto.

Adam – Kaggle

Comenzamos probando este modelo y lo primero que vemos es que se obtienen unos resultados nuevos, y es que en ningún momento la red ha aprendido. Se ha quedado con una tasa de exactitud de entrenamiento de 0.23xx y una de validación 0.2711 constantes a lo largo de la ejecución.

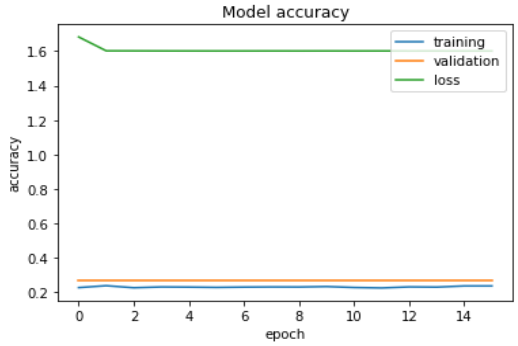


Ilustración 12 - Exactitud del modelo - VGG16 - Adam

En la matriz de confusión vemos un resultado totalmente nuevo. Sabemos que en la matriz de confusión las filas representan las clases, y las columnas las predicciones[2]. En este caso tendríamos que para 137 imágenes que sabemos que son de la clase 1 han sido predichas como clase 0; 234 como clase 1, 151 como clase 2, 150 como clase 3 y 191 como clase 4. Entonces, ¿significa esto que a la hora de armar los datasets no ha separado bien las clases? O tal vez que la ejecución ha sido correcta y que simplemente no ha clasificado imágenes de otras clases, es decir, de alguna forma se ha saltado la predicción para una imagen que se sabe que es de otra clase, en ese caso, ¿cómo es eso posible?

Otra alternativa puede ser que hayamos estado viendo la matriz de confusión al revés y que las columnas sean las clases y la filas las predicciones, de esta forma, en este caso, para todas las clases, ha realizado la misma predicción, es decir, predecir que es de la clase 1, por eso tiene en torno a un 20% de porcentaje de exactitud, como son 5 clases si para todas las imágenes se dice que es como en este caso de la quinta clase, un quinto de las veces estarás en lo cierto.

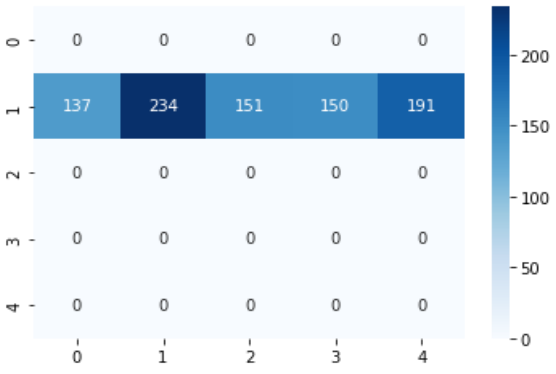


Ilustración 13 - Matriz de confusión - VGG16 - Adam

[2] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

Adadelta – Kaggle

Para este optimizador, también ocurre algo similar. No hay mucho que comentar puesto que ocurre prácticamente lo mismo. Parece que hay una ligera mejora en las tasas de exactitud, pero casi no se disminuye la pérdida.

Unas posibles razones por las cuales tiene lugar este fenómeno pueden ser la explosión o desvanecimiento del gradiente, o que se quede estancado en un punto silla (cosa que no debería pasar con este optimizador).

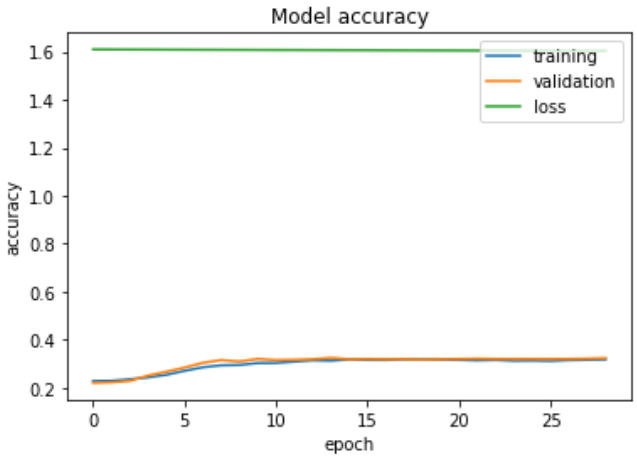


Ilustración 14 - Exactitud del modelo - VGG16 - Adadelta

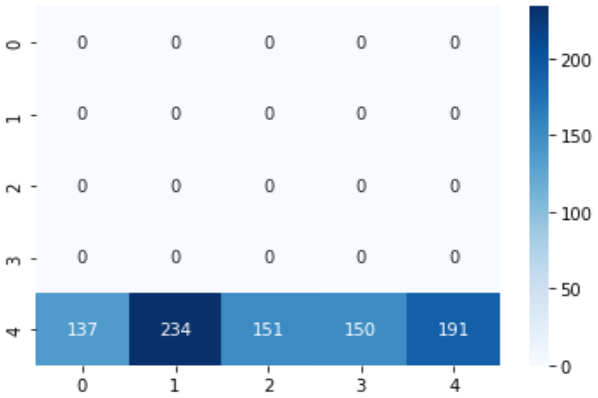


Ilustración 15 - Matriz de confusión - VGG16 - Adadelta

Adagrad – Kaggle

En este caso vemos una progresión más normal, en la que se reduce el valor de la función de error a lo largo del entrenamiento. Se aprecia un ligero sobreajuste La mejor tasa de exactitud de entrenamiento fue 0.7224 y de validación, 0.6161 en la misma época.

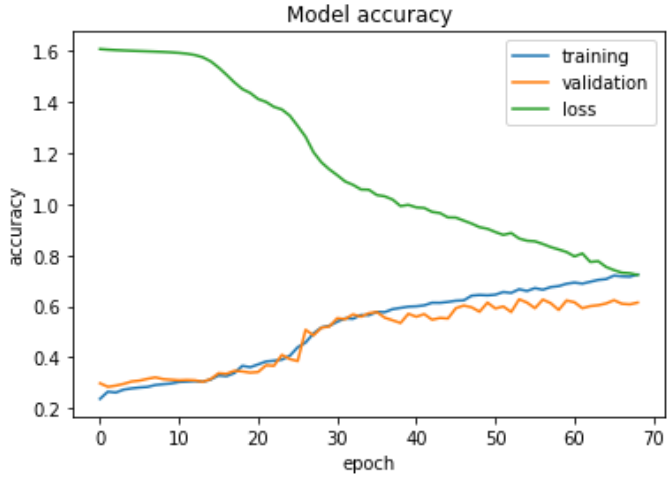


Ilustración 16 - Exactitud del modelo - VGG16 - Adagrad

La matriz de confusión es más normal, similar a todas las ejecuciones anteriores (que no son de este modelo).

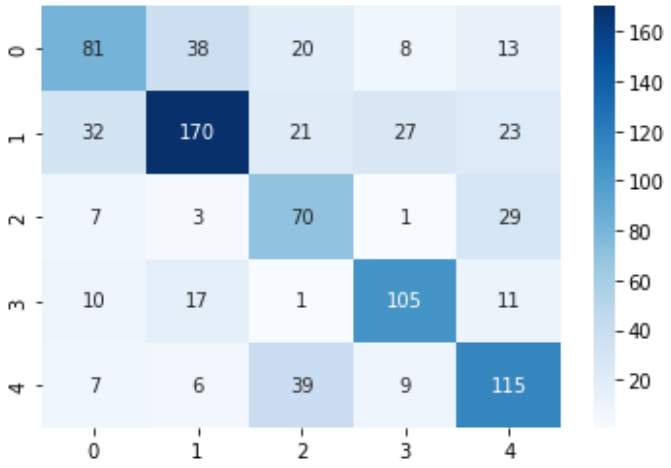


Ilustración 17 - Matriz de confusión - VGG16 - Adagrad

SGD – Kaggle

En este caso, los resultados son muy similares que la ejecución con los optimizadores Adam y Adadelta.

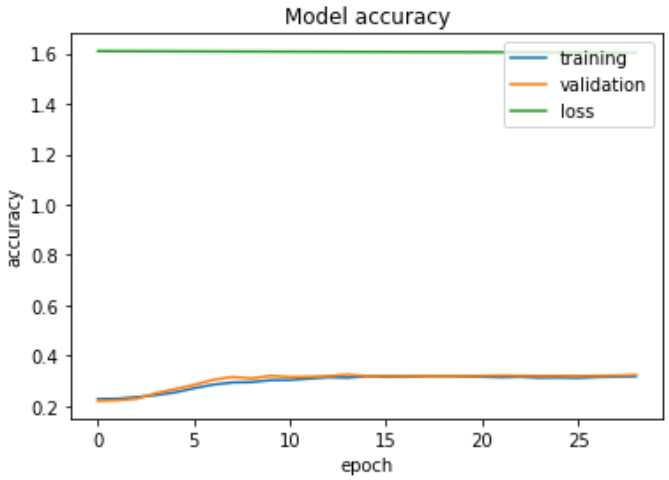


Ilustración 18 - Exactitud del modelo - VGG16 - SGD

La matriz de confusión es algo distinta puesto que sólo clasifica como dos clases.

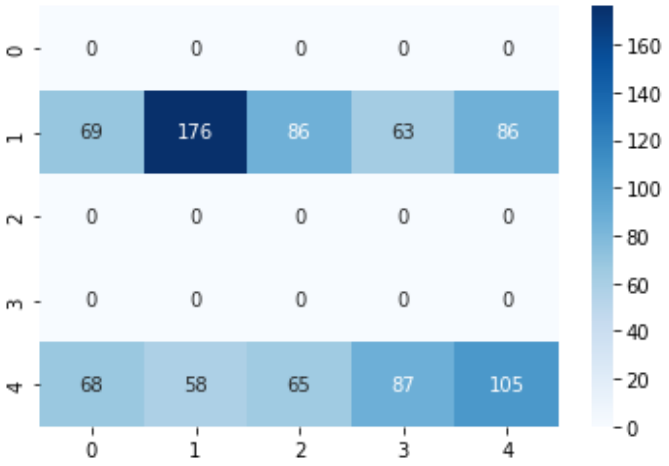


Ilustración 19 - Matriz de confusión - VGG16 - SGD

Quinta versión – VGG16, pero importado de keras.applications

Dado que algunos de los resultados que obtuvimos con nuestro modelo VGG16 se salían de lo esperado, decidimos repetir la ejecución, pero con el modelo que viene dado por la librería keras.applications, para confirmar que no nos habíamos equivocado poniendo las capas.

La implementación del modelo requería hacer unos ligeros cambios en el código. Estos cambios eran el cambio del tamaño de la imagen de (150, 150) a (224, 224) y unos cambios en los argumentos de la función que crea el modelo VGG16. A que poner los pesos (parámetro weights) a “None”, y el parámetro “classes” a 5, puesto que es el número de clases que contiene este dataset.

Adam – Kaggle

Empezamos probando el modelo con un resultado interesante, pues es muy similar a la ejecución con Adadelta y SGD del modelo anterior. No aprende y se queda en los mismos valores, una tasa de exactitud de entrenamiento de 0.023xx y una de validación de 0.02711.

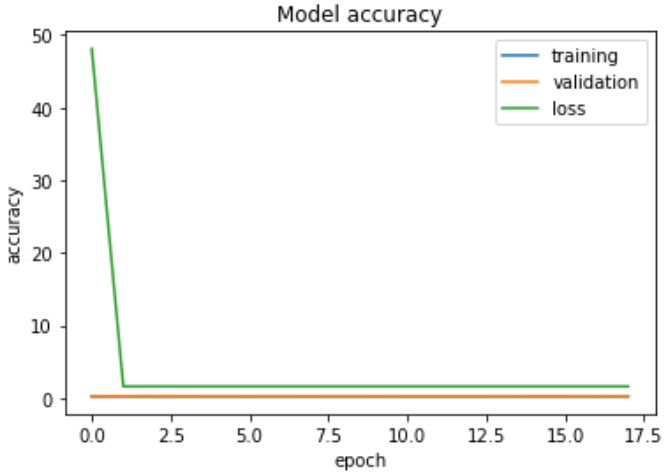


Ilustración 20 - Exactitud del modelo - VGG16 import - Adam

En la matriz de confusión también vemos lo ocurrido en la ejecución con SGD del modelo anterior, y es que clasifica imágenes o bien como de la clase 1 (dientes de león) o de la clase 4 (tulipanes) en menor medida.

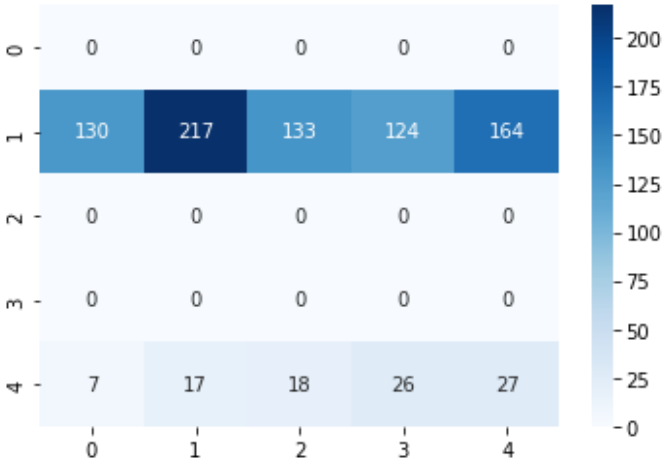


Ilustración 21 - Matriz de confusión - VGG16 import - Adam

Adadelta – Kaggle

En este caso vemos una ejecución más “normal”. Se aprecia un importante sobreajuste sobre todo en las épocas más avanzadas. La mayor tasa de exactitud para el conjunto de entrenamiento es de un 0.9076 y para el de validación, un 0.6176 en épocas distintas.

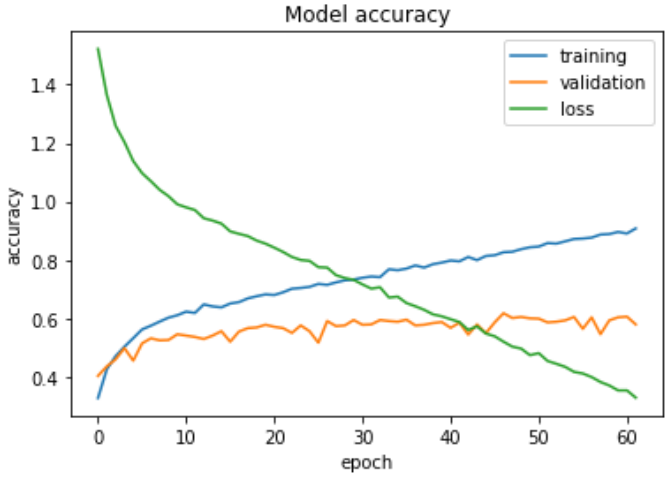


Ilustración 22 - Exactitud del modelo - VGG16 import - Adadelta

En la matriz de confusión vemos como hay bastante error, es decir hay bastantes predicciones fuera de la diagonal, pero al menos, esta tiene una densidad mayor que cualquiera que hemos visto en este modelo.

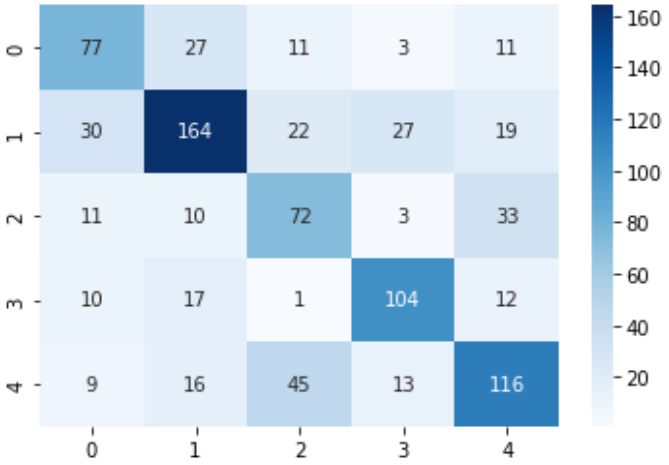


Ilustración 23 - Matriz de confusión - VGG16 import - Adadelta

Adagrad – Kaggle

Para el caso de este optimizador ha ocurrido algo similar, es decir, el protagonista ha sido el sobreajuste. Se ha llegado a un 0.9977 de tasa de exactitud en el conjunto de entrenamiento, pero después es seguido por un 0.6593 de tasa de exactitud en el conjunto de validación en la misma época.

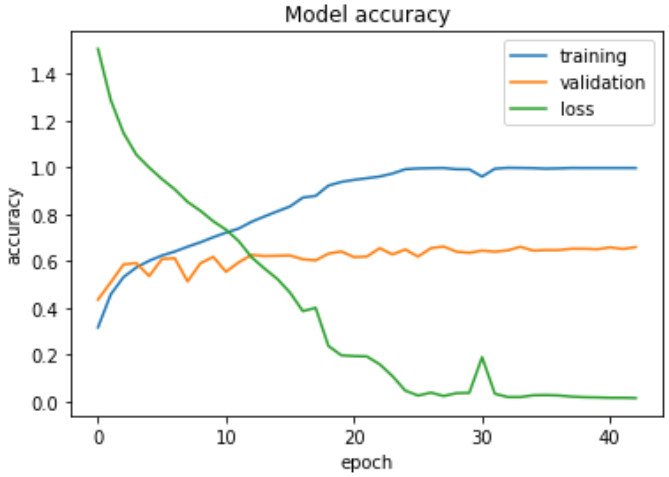


Ilustración 24 - Exactitud del modelo - VGG16 import - Adagrad

La matriz de confusión mejora con respecto a al anterior puesto que acumula más predicciones correctas. (Si sumamos las predicciones hechas en la diagonal, la suma en esta ejecución es mayor que en la anterior; 571 a 533 respectivamente)

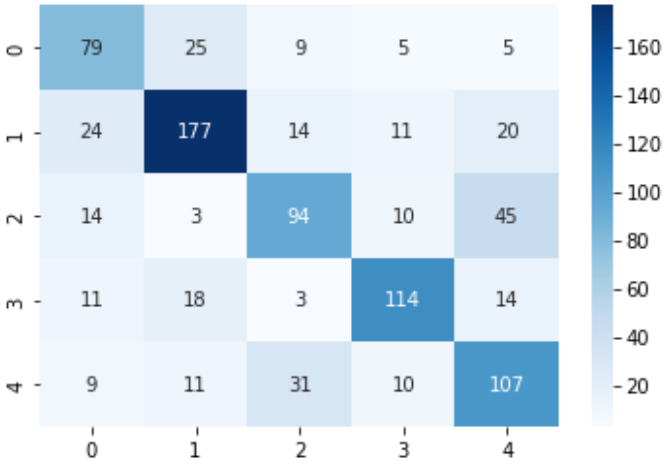


Ilustración 25 - Matriz de confusión - VGG16 import - Adagrad

SGD – Kaggle

Este caso es bastante similar a los anteriores, especialmente al anterior, llegando la tasa de exactitud del conjunto de entrenamiento a un 99.59%, pero con un importante sobreajuste que deja la tasa de exactitud del entorno de validación en un 63.73% en la misma época.

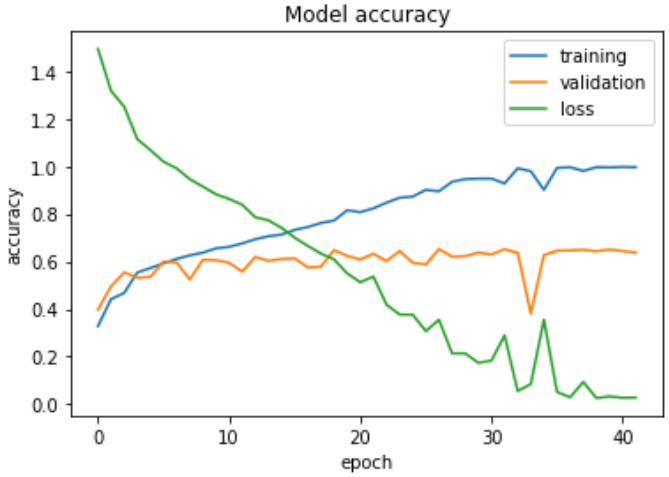


Ilustración 26 - Exactitud del modelo - VGG16 import - SGD

La matriz de confusión es muy similar a las anteriores, no hay nada nuevo que comentar.

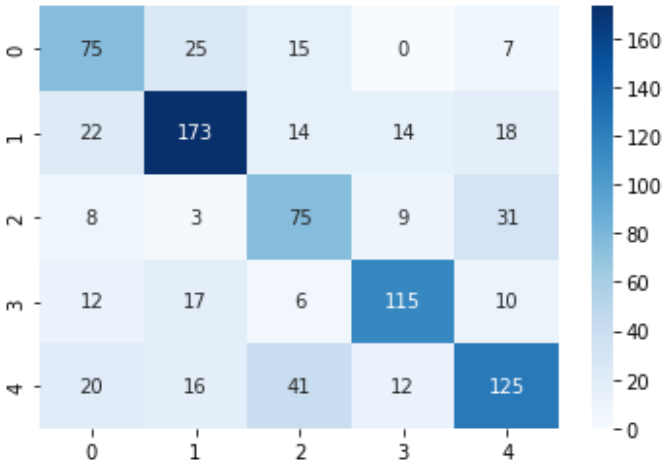


Ilustración 27 - Matriz de confusión - VGG16 import - SGD

Sexta versión - GoogleNet

La implementación de esta red es mucho más compleja que las anteriores, fue sacada del siguiente enlace:

https://github.com/KhuyenLE-maths/Implementation-of-GoogLeNet-on-Keras/blob/main/Implementation_of_GoogLeNet_on_Keras.ipynb

Para hacer funcionar al modelo, hubo que hacer unos ligeros cambios en el código. A priori, el modelo compilaba, pero no entrenaba, esto se debe a la diferencia de clases de salida del modelo. Se solucionó cambiando el primer parámetro de las capas fully connected que se activan con softmax por 5 el número de clases.

Después en la ejecución vemos la función de pérdida, la tasa de exactitud del conjunto de entrenamiento y de validación tres veces, esto se debe a las tres posibles salidas que tiene el modelo, se ven como dense_26, dense_28, dense_29. En la primera ejecución.

Adadelta – Kaggle

La primera ejecución ha sido sorprendente. Pasó por las 200 épocas y concluyó con unos porcentajes de exactitud en el entrenamiento y en la validación que en algunos casos han superado el 70%. No ha habido sobreajuste.

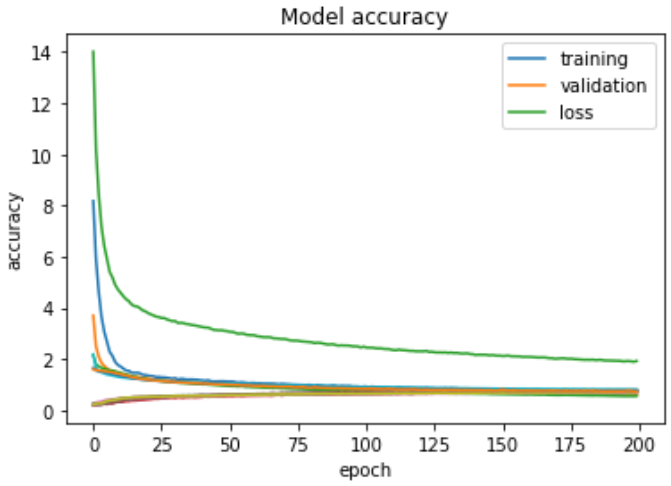


Ilustración 28 - Exactitud del modelo - GoogleNet - Adadelta

SGD – Kaggle

En este caso se observa como a medida que pasa de un número de épocas, empieza a notarse un mayor sobreajuste. En los tres conjuntos de entrenamiento se llega a unas tasas de acierto cercanas al 98% mientras que los conjuntos de validación se quedan entorno al 80%.

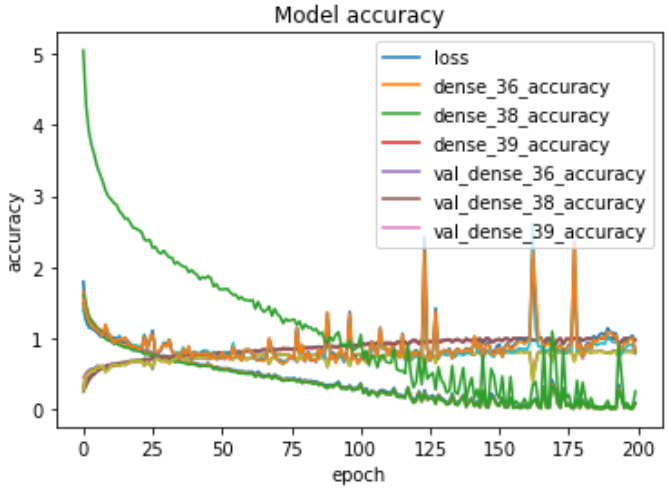


Ilustración 29 - Exactitud del modelo - GoogleNet - SGD

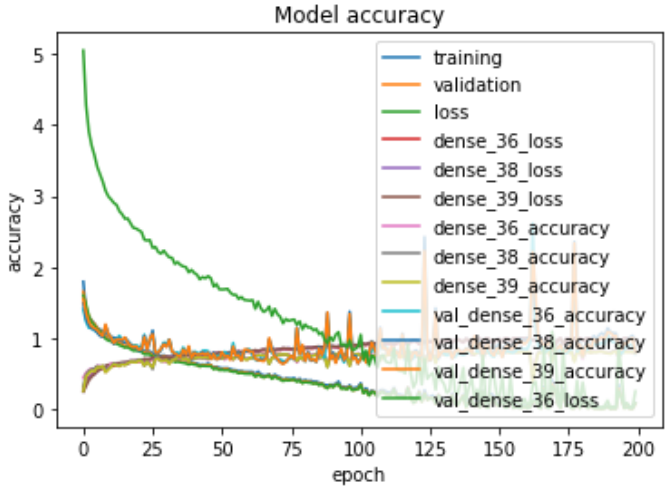


Ilustración 30 - Exactitud del modelo - GoogleNet - SGD con más leyenda

Séptima version – vuelta a los orígenes

La búsqueda del modelo quizás se nos ha ido un poco de las manos y hemos acabado probando GoogleNet, que ha dado de los mejores resultados, pero imposibilita una fácil y rápida modificación de hiperparámetros. En esta versión se parte de la primera. Se toma el modelo de la primera versión (tres capas convolutivas y dos densas), y le alteramos los hiperparámetros para ver hasta dónde puede llegar. La forma que hemos decido probar distintos hiperparámetros ha sido influenciada por un par de artículos[3].

Con el fin de probar hiperparámetros más relativos al modelo, los siguientes cambios se harán progresivamente con la imagen redimensionada a (150,150) y se usará el mismo optimizador, Adam con ritmo de aprendizaje de 0.001. La razón por la que se elige este optimizador es porque en los artículos se menciona como un favorito de propósito general. También se dejará la paciencia del EarlyStopping a 10.

[3] https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b
https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

Ejecución base

En esta primera ejecución del modelo se observan unos resultados en general bajos. Si nos fijamos en la evolución de la pérdida, se observa cómo tiene un ritmo de aprendizaje bajo, con varios picos. No hay sobreajuste, y consigue unas tasas de exactitud máximas de 0.9870 para el conjunto de entrenamiento y 0.7034 para el de validación.

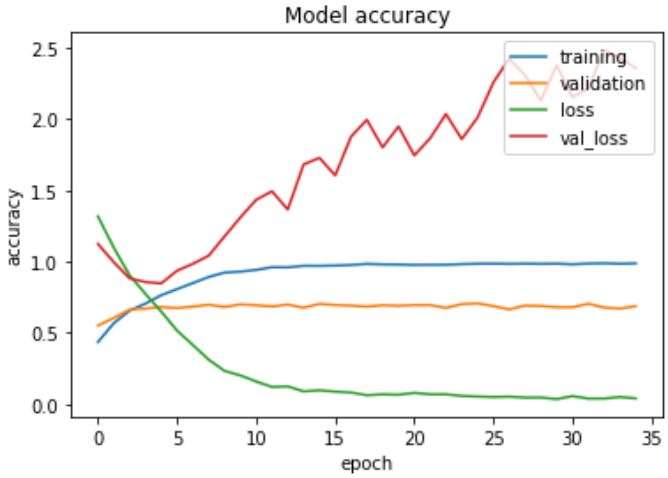


Ilustración 31 - Versión 7 - Ejecución base - Gráfica de error y exactitud

La matriz de confusión es bastante normalita. Resalta el número de predicciones de imágenes de la clase 4 como clase 2.

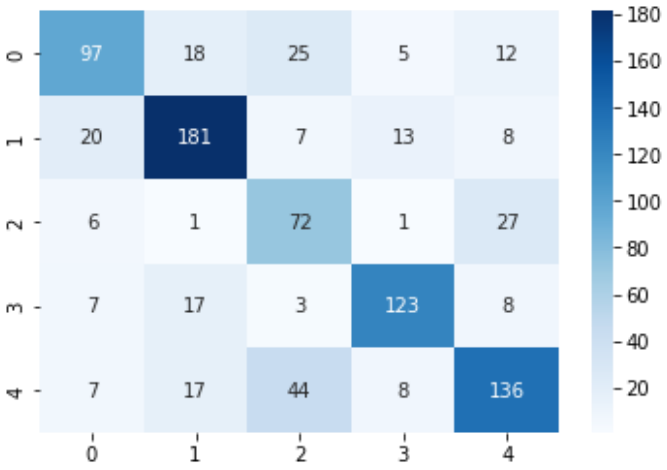


Ilustración 32 - Versión 7 - Ejecución base - Matriz de confusión

Primera mejora

Para esta primera mejora lo que se hace es añadir una nueva capa convolutiva con u kernel más grande. La razón detrás del cambio, según el artículo, es comenzar usando filtros más pequeños para recolectar la mayor cantidad de información local posible, para cuando después se incremente, obtener información que represente un nivel más alto y global. Se vuelve a observar sobreajuste.

Las evoluciones de las tasas son similares, sin embargo, mejora con respecto a la anterior en que llega a un 97.42% de tasa máxima de entrenamiento y 72.77% de validación.

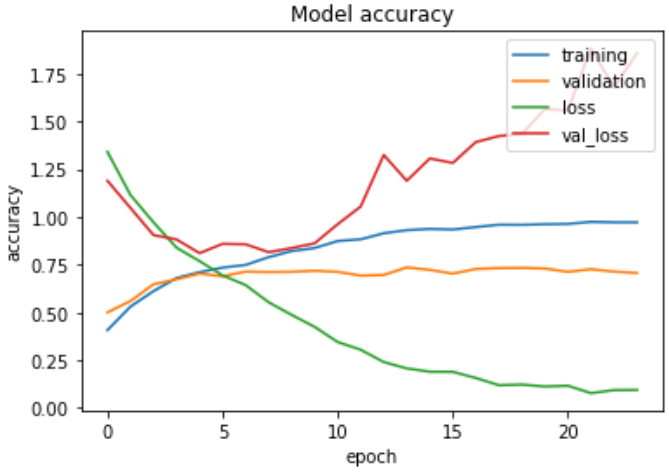


Ilustración 33 - Versión 7 - Primera mejora - Gráfica de error y exactitud

La matriz de confusión es muy similar a la anterior.

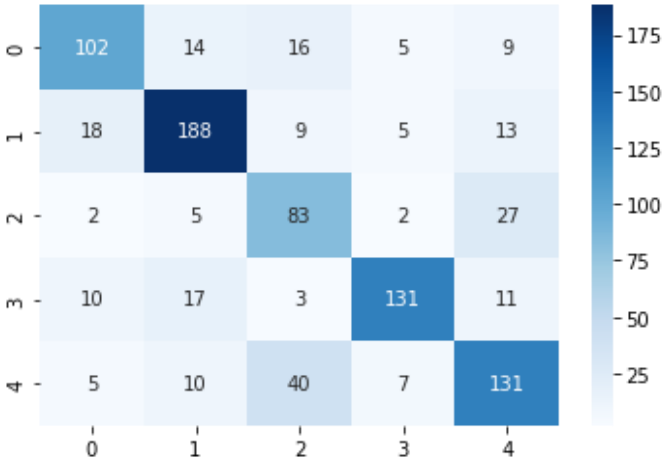


Ilustración 34 - Versión 7 - Primera mejora - Matriz de confusión

Segunda mejora

Para esta mejora se utilizan los mismos tamaños de kernel, se añaden más capas de droptou para reducir el sobreajuste y se usa el GlobalAveragePooling para reemplazar las capas densas de la red neuronal convolucional. Se reduce la dimensionalidad tal y como se comenta en Redes neuronales 9.

El resultado es una mejora en la tasa de validación, llegando 77.97%. Se sigue observando sobreajuste.

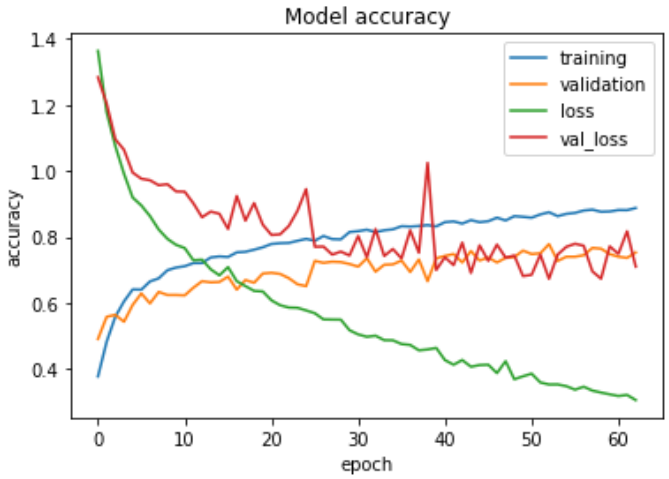


Ilustración 35 - Versión 7 - Segunda mejora - Gráfica de error y exactitud

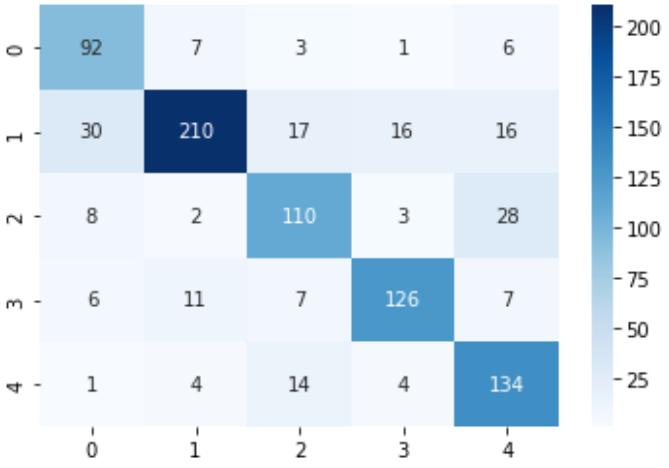


Ilustración 36 - Versión 7 - Segunda mejora - Matriz de confusión

Tercera mejora

En esta mejora, a la hora de hacer el rescalado, volvemos a la base de hacerlo sobre 255, obteniendo valores entre 0 y 1. Además, en la última capa convolutiva, subimos los filtros hasta 128 y el kernel_size a (5, 5). Por último, ajustamos los parámetros del optimizador (Adam), con los ajustes que Keras recomienda por defecto.[4]

Se reduce el sobreajuste y se llega hasta el 78.79% de tasa de exactitud de validación.

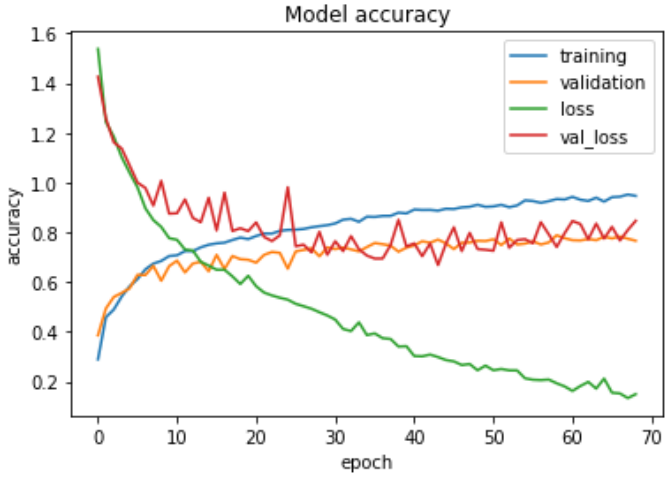


Ilustración 37 - Versión 7 - Tercera mejora - Gráfica de error y exactitud

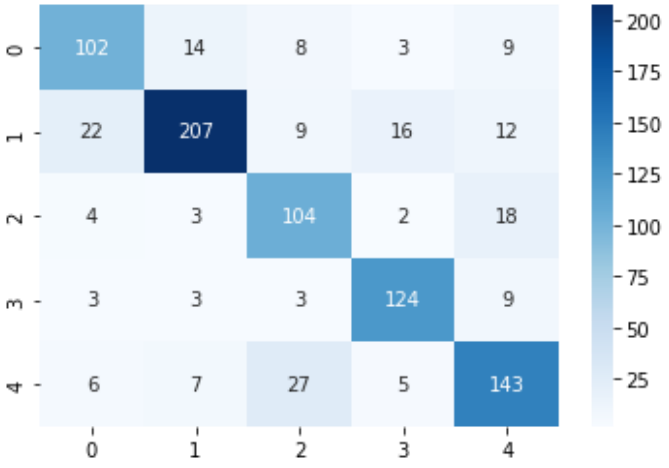


Ilustración 38 - Versión 7 - Tercera mejora - Matriz de confusión

[4] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Cuarta mejora

En esta cuarta mejora se realizan las siguientes modificaciones:

- Se agrega una capa de Batch norm después de cada capa convolutiva.
- El primera MaxPooling tiene un campo receptivo más grande (3, 3) y todas las capas maxpooling caminan dos pasos en ambas direcciones, strides = (2, 2)
- Se usa GlobalAveragePooling en vez de capas densas y el flatten.

Aun así, sigue habiendo bastante sobreajuste, con el conjunto de entrenamiento llegando al 99.31%, pero se alcanza un pico de 80.42% de tasa de acierto para el conjunto de validación.

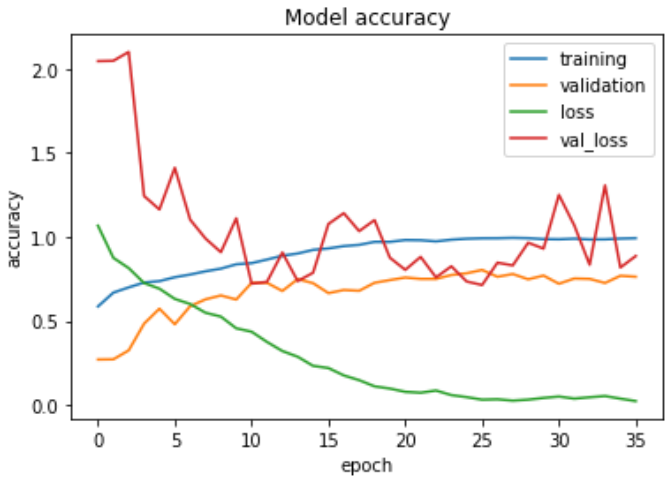


Ilustración 39 - Versión 7 - Cuarta mejora - Gráfica de pérdida y exactitud

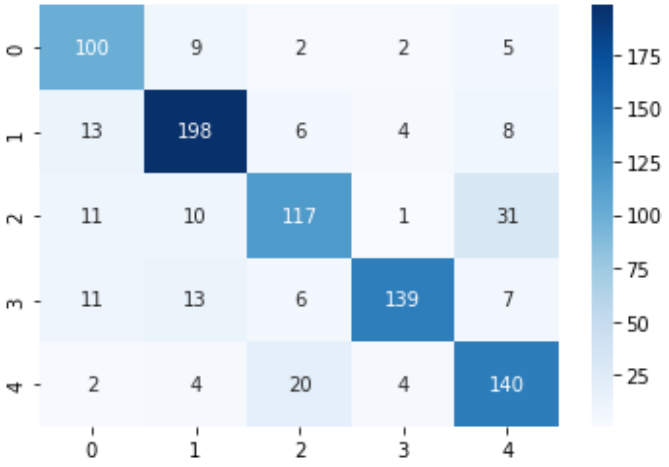


Ilustración 40 - Versión 7 - Cuarta mejora - Matriz de confusión

Quinta mejora

Para esta quinta mejora se implementa la técnica de Data Augmentation. Para ello se debe modificar el dataset, dividiendo en conjuntos de entrenamiento y validación. Después, se instancia la clase ImageDataGenerator para aplicar las transformaciones pertinentes a las imágenes. Se realizan dos ejecuciones. El motivo es que la primera progresaba adecuadamente, pero fue detenida por el EarlyStopping y pensamos que quizás podía dar algo más. La segunda ejecución se le añade una transformación al ImageDataGenerator del conjunto de entrenamiento y se incrementa la paciencia con respecto al early stopping. Dado que cambia la forma en la que se crean los datasets no se ha podido reutilizar el bloque de código que genera la matriz de confusión, por lo que se muestra sólo las tasas de exactitud. Utilizando Data Augmentation se han obtenido los mejores resultados, llegando a un 82.25% de tasa de exactitud de validación en la segunda ejecución.

Para la primera ejecución se observa una mejora con respecto al sobreajuste.

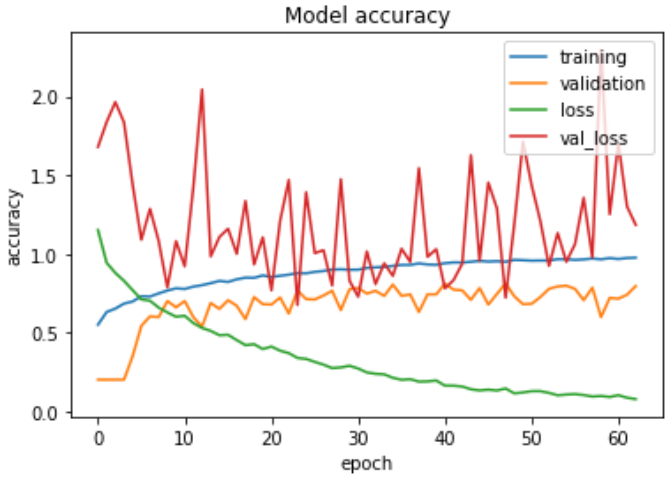


Ilustración 41 - Versión 7 - Quinta mejora - Primera ejecución - Gráfica de pérdida y exactitud

Para la segunda ejecución se observa más sobreajuste y unas tasas de exactitud de validación muy inestables con tendencia a la baja.

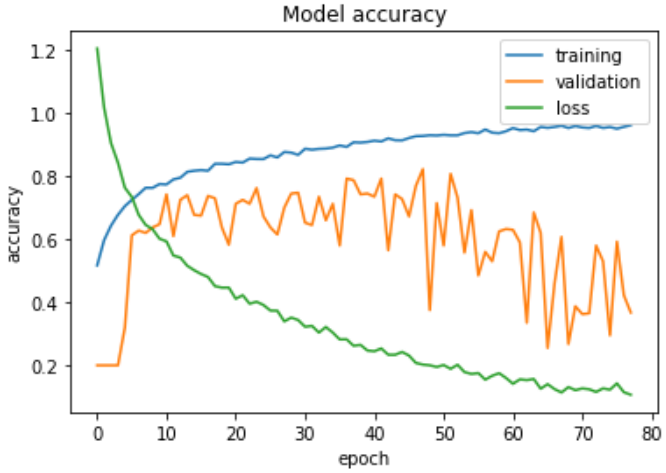


Ilustración 42 - Versión 7 - Quinta mejora - Segunda ejecución - Gráfica de pérdida y exactitude

Cross entropy

La explicación que viene a continuación acerca de la comparación de funciones de error está sacada de esta fuente[5]. Se muestra el mismo ejemplo:

Supongamos el caso más reducido, donde la salida de la red es binaria y viene dada por la activación de una neurona en la capa de salida. Sea y la etiqueta e y' la predicción. Para el caso en que:

$$y = 0\,; y' = 0.9$$

Sabiendo que el error cuadrático medio **para este caso** viene dado por:

$$E = (y - y')^2$$

Y la pérdida de entropía cruzada binaria, también simplificada para este caso, es:

$$E = -(ylog(y') + (1 - y)\log(1 - y'))$$

El error cuadrático sería:

$$E_{Cuadrático} = (0 - 0.9)^2 = 0.81$$

Mientras que el error de entropía cruzada sería:

$$E_{CE} = -(0\log(0.9) + \log(0.1)) = 2.3$$

Se observa que el error de entropía cruzada es mucho mayor que el error cuadrático, por lo tanto, penaliza más los errores de la red haciendo que esta aprenda más rápido. Estos valores los vemos representados a continuación.

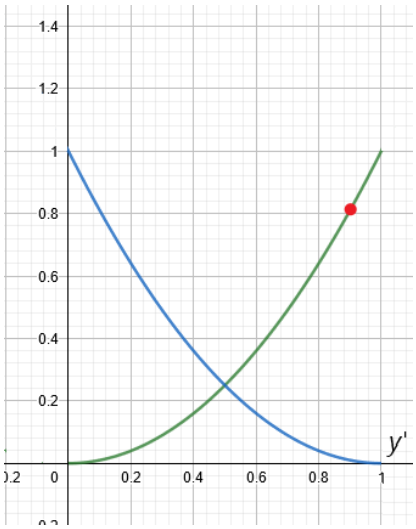


Ilustración 43 - Error cuadrático

[5] https://www.youtube.com/watch?v=gIx974WtVb4

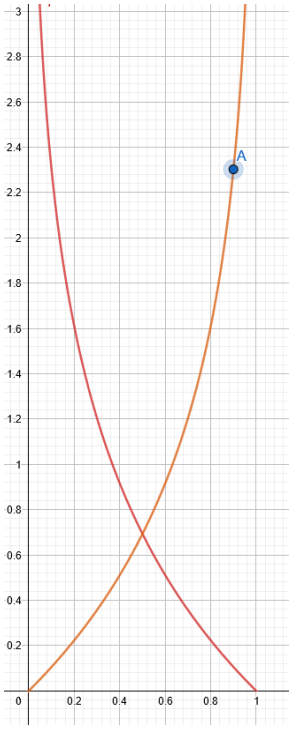


Ilustración 44 - Error calculado por entropía cruzada

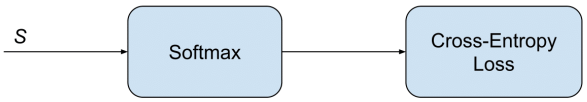
Las correspondientes pendientes son:

$\frac{\partial E_C}{\partial y\prime} = 2 * 0.9 = 1.8$; para el error cuadrático.

$\frac{\partial E_{CE}}{\partial y\prime} = \frac{1}{0.1} = 10$; para la función de entropía cruzada.

Se observa claramente como la pendiente para la función de entropía cruzada es en este caso casi 6 veces mayor que para el error cuadrático, teniendo como consecuencia un descenso más rápido por el gradiente.

La pérdida "Categorical Cross-Entropy" no es más que una activación softmax más una pérdida de entropía cruzada. Si usamos esta pérdida, entrenaremos una red neuronal convolucional para que saque una probabilidad sobre un conjunto de clases para una imagen, es la extensión a múltiples clases.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \quad CE = -\sum_i^C t_i log(f(s)_i)$$

Tabla mostrando los resultados

Modelos	Tasa máxima de entrenamiento	Tasa máxima de validación
Modelo básico	97.89%	70.57%
AlexNet – Adam	95.19%	73.93%
AlexNet – Adadelta	75.68%	65.01%
AlexNet – Adagrad	98.61%	72.77%
AlexNet – SGD	99.16%	74.62%
VGG16 – Adam	23.63%	27.11%
VGG16 – Adadelta	31.88%	32.44%
VGG16 – Adagrad	72.24%	61.61%
VGG16 – SGD	31.88%	32.44%
VGG16 Imp – Adam	23.78%	27.11%
VGG16 Imp – Adadelta	90.76%	61.76%
VGG16 Imp – Adagrad	99.77%	65.93%
VGG16 Imp – SGD	99.59%	63.73%
GoogleNet – Adadelta	78.92%	72.77%
GoogleNet – SGD	97.89%	80.53%

Hiperparámetros	Tasa máxima de entrenamiento	Tasa máxima de validación
Ejecución base	98.70%	70.34%
Primera mejora Capa conv 5x5	97.42%	73.35%
Segunda mejora GAP	88.80%	75.32%
Tercera mejora GAP + conv 5x5	95.17%	78.79%
Cuarta mejora BatchNorm + Strides	99.51%	80.42%
Quinta mejora DataAugmentation	96.20%	82.25%

Webgrafía

Material de la asignatura

https://cayetanoguerra.github.io/ia/

Dataset

https://www.kaggle.com/alxmamaev/flowers-recognition

Keras y conocimiento en general

https://stackoverflow.com/questions/44747343/keras-input-explanation-input-shape-units-batch-size-dim-etc

https://keras.io/api/layers/convolution_layers/convolution2d/

https://keras.io/api/applications/

https://en.wikipedia.org/wiki/Confusion_matrix

https://en.wikipedia.org/wiki/Stochastic

https://en.wikipedia.org/wiki/Stochastic_gradient_descent

https://www.youtube.com/watch?v=vMh0zPT0tLI

https://paperswithcode.com/method/adadelta

https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Primera versión

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_1_piloto/150_epochs_con_fit_generator.ipynb

Redes neuronales 7 + Enunciado práctica

Práctica 2: Red convolutiva en Keras

Segunda versión

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_2_Base/Version_2.ipynb

Tercera versión – AlexNet

Strides

https://towardsdatascience.com/visualizing-the-fundamentals-of-convolutional-neural-networks-6021e5b07f69

Batch Normalization

https://towardsdatascience.com/batch-normalization-in-practice-an-example-with-keras-and-tensorflow-2-0-b1ec28bde96f

https://arxiv.org/pdf/1502.03167v3.pdf

https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

https://en.wikipedia.org/wiki/Batch_normalization

https://en.wikipedia.org/wiki/Whitening_transformation

ZeroPadding2D

https://keras.io/api/layers/reshaping_layers/zero_padding2d/

Adam – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_3_AlexNet/AlexNet_Adam_Kaggle.ipynb

Adadelta - Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_3_AlexNet/AlexNet_Adadelta_Kaggle.ipynb

Adagrad – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_3_AlexNet/AlexNet_Adagrad_Kaggle.ipynb

SGD – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_3_AlexNet/AlexNet_SGD_Kaggle.ipynb

Cuarta version – VGG16

https://keras.io/api/applications/

https://keras.io/api/applications/vgg/

Adam – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_4_VGG16/new_vgg16-adam-kaggle.ipynb

Adadelta – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_4_VGG16/modelo-vgg16-adadelta-kaggle.ipynb

Adagrad – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_4_VGG16/modelo-vgg16-adagrad-kaggle.ipynb

SGD – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_4_VGG16/modelo-vgg16-sgd-kaggle.ipynb

Quinta version – VGG16, pero importado de keras.applications

Adam – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_5_VGG16_importado/VGG16-Imp-Adam.ipynb

Adadelta – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_5_VGG16_importado/VGG16%20Importado%20Adadelta.ipynb

Adagrad – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_5_VGG16_importado/VGG16%20Importado%20Adagrad.ipynb

SGD – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_5_VGG16_importado/VGG16%20Importado%20SGD.ipynb

Sexta version – GoogleNet

Fuente

https://medium.com/mlearning-ai/implementation-of-googlenet-on-keras-d9873aeed83c

https://github.com/KhuyenLE-maths/Implementation-of-GoogLeNet-on-Keras/blob/main/Implementation_of_GoogLeNet_on_Keras.ipynb

Adadelta – Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_6_GoogleNet/GoogleNet%20Adadelta.ipynb

SGD - Kaggle

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_6_GoogleNet/GoogleNet%20SGD.ipynb

Séptima versión – vuelta a los orígenes

Artículos para tener una visión subjetiva de cómo cambiar hiperparámetros

https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b

https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

Versión Base

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/Version_7_Ejecucion_base.ipynb

Primera mejora

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_1_CapaConv_Con_Kernel_Mas_Grande.ipynb

Segunda mejora

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_2_GAP.ipynb

https://paperswithcode.com/method/global-average-pooling

Tercera mejora

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_3_compiler_and_layers_adjustment.ipynb

Cuarta mejora

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_4.ipynb

Quinta mejora

Primera ejecución

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_5_DataAug1.ipynb

Segunda ejecución

https://github.com/Isac-AS/FSI-PL-RedesNeuronales-Flores/blob/Version_7_Modificacion_de_hiperpar%C3%A1metros_a_partir_de_la_base/V7_Mejora_5_DataAug2.ipynb

Cross entropy

https://www.youtube.com/watch?v=gIx974WtVb4