UNIVERSIDAD
DE GRANADA

Visión por Computador
grado en ingeniería informática

Práctica 2

Redes Neuronales Convolucionales

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

1. BaseNet en CIFAR100

Antes de empezar con la traducción de la arquitectura proporcionada de *BaseNet*, hace falta establecer la forma de la entrada de la primera capa de la red. Esto es necesario, ya que el modelo necesita conocer dicho tamaño para poder ser compilado sin ningún tipo de error. Como las imágenes de *CIFAR100* tienen un tamaño de 32×32 píxels, y tienen 3 canales, la dimensión de la entrada va a ser la siguiente:

```
# Tamaño de la entrada
input_shape = (32, 32, 3)
```

Una vez definida la forma de la entrada, ya se puede empezar a hacer la traducción a código. El resultado es el siguiente:

```
# Creacion del modelo
model = Sequential()
model.add(Conv2D(6, kernel_size=(5, 5), padding='valid',
                 input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(16, kernel_size=(5, 5), padding='valid'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(units=50))
model.add(Activation('relu'))
model.add(Dense(units=25))
model.add(Activation('softmax'))
```

BaseNet es un modelo secuencial, así que empezamos indicando eso. A continuación, añadimos el primer módulo convolucional. Este se compone de una convolución 2D con un *kernel* de 5×5, una función de activación no lineal (RELU en este caso) y un MaxPooling de tamaño 2×2. El parámetro $padding = valid$ de $Conv2D$ indica que solo se tiene que aplicar la convolución allá donde se pueda ajustar el *kernel*; es decir, como en las regiones de los bordes no se puede, se van a ignorar estas zonas, lo cuál implica que la salida no va a tener el mismo tamaño que la entrada. Este módulo convolucional se repite otra vez. Después de eso, nos encontramos con las capas densas, las cuáles van a actuar como clasificador. La capa *Flatten* es necesario ponerla, ya que coge la salida de la anterior, la cuál es un bloque de tamaño $5 \times 5 \times 16$ (16 imágenes 5×5), y aplana dicho bloque, convirtiéndolo en un vector de 400 características, el cuál sirve como entrada al modelo denso. La última capa, la de activación *softmax* es la que va a dar la salida, un vector de probabilidades para cada clase. En este caso, la salida va a ser un vector de 25 posiciones, ya que estamos en un problema donde hay 25 clases.

Con el modelo ya definido, y antes de compilarlo, tenemos que establecer algunas cosas más:

- **Optimizador**. El optimizador que se va a utilizar en este caso es el **SGD** o *Stochastic Gradient Descent*. Este es uno de los optimizadores más populares e importantes dentro del *machine learning*. Se utiliza mucho con redes neuronales, ya sean redes normales o profundas, y es conocido por su robustez y por ofrecer unos muy buenos resultados en general, además de ser bastante rápido a diferencia de otros optimizadores, como por ejemplo Adam. En este caso, se va a utilizar con los parámetros por defecto. Es decir, se tendrá un *learning rate* de 0.01, y no se utilizará momentum ni el momentum de Nesterov. Estos parámetros parecen razonables, ya que el *learning rate* no es ni demasaido pequeño ni demasiado grande. Además, como es un poco difícil ajustar el momento, se ha preferido no tocar este parámetro.
- **Tamaño del *batch***. Otro elemento muy importante a determinar es el tamaño del *batch*, si bien no es necesario conocerlo en el momento en el que se compila el modelo. Para un problema como este, teniendo en cuenta que tenemos unos 11250 datos de entrenamiento, un tamaño de *batch* razonable es de 32. Este es un tamaño muy utilizado, ya que en general ofrece unos buenos resultados, ya que permite converger a buenos óptimos y hace que el modelo generalice bastante bien. Con un tamaño menor se estaría explorando demasiado el espacio, mientras que con uno mayor se estaría explotando una zona del espacio, lo cuál puede llegar a producir problemas, como que no se generalice demasiado bien [1], cosa que no nos interesa.
- **Número de épocas**. Éste es quizás el parámetro más difícil de decidir a priori, ya que no tenemos mucha información. Por tanto, ya que a medida que vayamos haciendo pruebas podremos ver las curvas de entrenamiento y validación, podremos decidir en función de éstas cuántas épocas debemos entrenar los modelos. Para empezar, podemos fijar unas 30 épocas, ya que parece un número razonable.

Con esto ya visto, podemos compilar nuestro modelo. Lo primero que tenemos que hacer es definir el optimizador. Como vamos a utilizar SGD, lo hacemos de la siguiente forma:

```
# Establecer optimizador a utilizar
optimizer = SGD()
```

Para compilar el modelo, lo haremos de la siguiente forma:

```
# Compilar el modelo
model.compile(
    loss=keras.losses.categorical_crossentropy,
```

```
    optimizer=optimizer,
    metrics=['accuracy']
)
```

Como estamos en un problema de clasificación y la salida que va a dar el modelo es un vector de probabilidades para múltiples clases, especificamos que la función de pérdida que se utilizará es la entropía cruzada o *Categorical Cross-Entropy*, la cuál es muy utilizada en problemas de clasificación para múltples clases. Especificamos también cuál será el optimizador a utilizar, e indicamos que la métrica que nos interesa es la precisión o *accuracy*, que representa la proporción de aciertos sobre el número total de elementos. Existen muchas otras métricas que se pueden utilizar, pero la *accuracy* es la más sencilla de entender.

Con el modelo ya compilado, podemos visualizarlo de la siguiente forma:

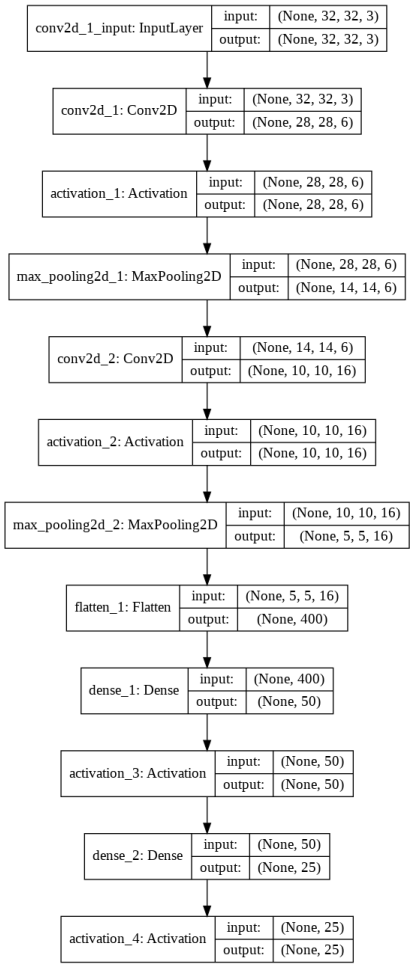


Figura 1: Esquema que representa el modelo *BaseNet*.

Aquí es donde podemos ver mejor la estructura del modelo. Se puede ver de forma más clara que antes la estructua secuencial que tiene, además de cada una de las capas y de los tamaños de las entradas y de las salidas de éstas.

Teniendo el modelo ya construido y compilado, para hacernos una idea de como de bueno es, podemos entrenarlo y probarlo con el conjunto de test. Es muy importante, antes de empezar, guardar los pesos que tiene el modelo. De esta forma, podremos restablecerlos posteriormente, para poder reentrenar el modelo. Para guardar los pesos, podemos hacerlo de la siguiente forma:

```
# Guardar los pesos iniciales del modelo
weights = model.get_weights()
```

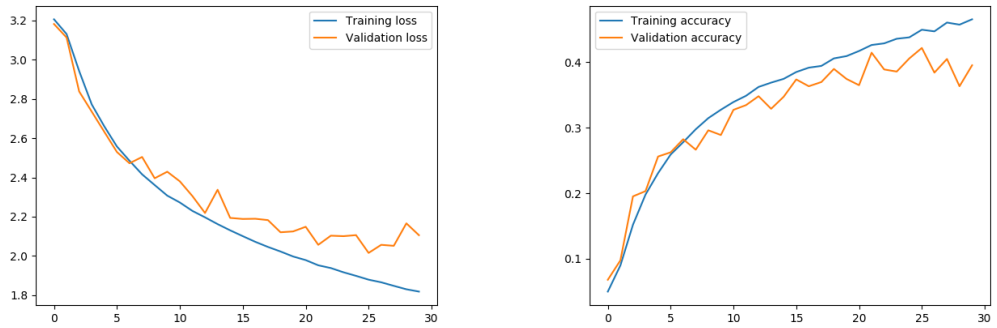
Ahora ya podemos proceder al entrenamiento. Es muy importante destacar que, con los datos de entrenamiento de los que disponemos, solo se tiene que entrenar con el 90 % de éstos; el 10 % restante se tiene que dejar para validar el modelo, y de esta forma poder obtener unas gráficas para el error y la *accuracy* en los conjuntos de entrenamiento y de validación. Estos valores que se obtienen para el conjunto de validación son, en general, una buena aproximación de lo que se puede obtener en el conjunto de test, si la muestra es lo suficientemente representativa de la población total, claro está.

Para entrenar el modelo, lo hacemos de la siguiente forma:

```
# Entrenar el modelo
history = model.fit(
    x_train,
    y_train,
    validation_split=0.1,
    epochs=epochs,
    batch_size=batch_size,
    verbose=1
)
```

Especificamos que se utilizan las particiones de entrenamiento x_train (las imágenes) e y_train (la etiqueta asociada a cada una de las imágenes del conjunto de entrenamiento). Además, con $validation\_split = 0.1$ indicamos que solo el 10 % de los datos se utilizarán para validar el modelo. Se especifica también el tamaño del *batch* (recordemos que lo habíamos fijado a 32) y el número de épocas (30 inicialmente). El parámetro *verbose* es solo para mostrar el progreso del entrenamiento; no tiene ningún otro fin.

Este método devuelve una historia, la cuál se almacena en la variable *history*. Esta historia contiene trazas de la evolución de los valores de la función de pérdida y de *accuracy* en los conjuntos de entrenamiento y de validación. Para este caso, hemos obtenido los siguientes resultados:



(a) Evolución de la función de pérdida. (b) Evolución de la *accuracy*.

Figura 2: Historia del modelo *BaseNet*.

Podemos ver que no se produce *overfit*, ya que a medida que el valor del error o de la función de pérdida va bajando en el conjunto de entrenamiento, también lo hace en el de validación, hasta que llega a las últimas épocas, donde dicho valor se queda más o menos se queda un poco por encima del de entrenamiento, pareciendo que se estanca. En ningún momento el error en validación llega a subir. Si esto hubiese sucedido, podríamos haber afirmado de forma clara que se ha producido *overfit* en nuestro modelo. Si miramos también la *accuracy*, podemos ver que, a medida que va subiendo dicho valor en el conjunto de entrenamiento, también lo hace en el de validación. Aquí de nuevo sucede algo como en el caso de la función de pérdida, ya que parece que en las últimas épocas este valor se va quedando un poco estancado, aunque no dista mucho del valor obtenido en el conjunto de entrenamiento.

Sin embargo, aunque el modelo no padezca de *overfit*, sí que lo hace de *underfit*: la evolución de los valores de la función de pérdida y de la *accuracy*, aunque en un principio parecen buenos ya que el error va disminuyendo y la precisión aumentando, no es del todo satisfactoria. Podemos ver claramente como el error, aún en el conjunto de entrenamiento, sigue siendo bastante alto (en las últimas épocas se queda en torno a 1.8, valor bastante alto), y en el caso de la precisión se queda en torno a 0.45. En el caso del conjunto de validación, aunque al principio los valores sean más o menos parejos con el conjunto de entrenamiento en ambas gráficas, podemos ver como al cabo de aproximadamente unas 15-20 épocas los valores empiezan a ser dispares. En el caso de la función de pérdida, al final, los valores que se obtienen están en torno a 2, mientras que en la precisión los valores obtenidos no superan el 0.4, quedándose por debajo de los obtenidos en entrenamiento.

En líneas generales, estos resultados son demasiado pobres: lo ideal hubiese sido alcanzar un error cercano a 1 o más bajo y una precisión superior a 0.5 en el

conjunto de entrenamiento, y que los valores obtenidos en el conjunto de validación hubiesen seguido casi perfectamente a los de entrenamiento. Por tanto, de aquí podemos extraer que todavía existe mucho margen de mejora.

Es importante destacar, antes de continuar, que todos los resultados que se obtienen dependen de la ejecución. Es decir, que para dos ejecuciones puede que los resultados no sean exactamente los mismos; sin embargo, podemos decir que estarán bastante cerca, en general, ya que los datos son los mismos.

Para tener una idea de cómo de bien funciona nuestro modelo base con el conjunto de test, y para tener un valor de *accuracy* que podemos utilizar para comparar este modelo base con las mejoras futuras, vamos a hacer que prediga las etiquetas del conjunto *x_test* (las imágenes de test), y compararemos dichos valores con los reales, los cuáles están en la variable *y_test*. Para hacer dicha predición, podemos hacerla de la siguiente forma:

```
# Predecir los datos
prediction = model.predict(
    x_test,
    batch_size=batch_size,
    verbose=1
)
```

De esta forma, especificamos que el modelo prediga las etiquetas asociadas al conjunto *x_test* y se le especifica un tamaño de *batch*, que será el número de elementos máximos que se predigan de golpe; es decir, no se va mandar a CPU/GPU un conjunto de datos de mayor tamaño que el especificado. El parámetro *verbose* es, de nuevo, para mostrar el proceso.

El valor de *accuracy* comparando los valores predichos con los reales gira en torno a 0.4 tras realizar algunas pruebas. Dicho valor, a pesar de no ser del todo horrible para un modelo tan simple, es bastante bajo, y creemos que tiene cierto margen de mejora. ya que posiblemente, con realizar algunas modificaciones, podamos llegar una *accuracy* igual o superior a 0.5. Por tanto, vamos a intentar mejorar nuestro modelo en la próxma sección, para ver hasta dónde somos capaces de llegar.

2. Mejora del modelo

En esta sección, vamos a ir proponiendo una serie de mejoras que podemos hacer sobre el modelo. Estas mejoras son acumulativas, es decir, que se van realizando una sobre la otra, siempre y cuando ofrezcan unos buenos resultados.

Para cada caso, vamos a discutir brevemente qué es lo que se va a mejorar,

qué parámetros se van a utilizar y cuáles son los resultados obtenidos. Para cada experimento mostraremos gráficas, iguál que las que se pueden ver en la figura 2.

Una vez que hayamos encontrado un modelo bueno (es decir, uno que no sufre ni de *overfit* ni de *underfit*, y ofrece unos valores de error y precisión razonables en el conjunto de validación), utilizaremos el conjunto de test para ver cómo de bien lo hace, y es entonces cuando podremos comparar dichos resultados con el modelo base, para poder ver hasta dónde hemos llegado. En ningún otro caso utilizaremos dicho conjunto, ya que no es buena idea dejarnos llevar por los resultados obtenidos en test para decir que un modelo es mejor que otro; para eso tenemos el conjunto de validación.

2.1. Normalización de los datos

La primera mejora que vamos a introducir es la normalización de los datos de entrada, haciendo que éstos tengan media $\mu = 0$ y desviación típica $\sigma = 1$. Se introduce esta mejora porque se sabe que gracias a la normalización se pueden obtener unos mejores resultados, además de que el entrenamiento de la red se puede llegar a acelerar.

La manera más fácil de normalizar los datos es utilizar un generador de la clase $ImageDataGenerator$. Para crearlo, podemos utilizar el siguiente fragmento de código:

```
datagen_train = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    validation_split=0.1
)
```

De esta forma, creamos un generador para los datos de entrenamiento, el cuál hará que la media sea 0 (con el parámetro $featurewise\_center = True$), normalizará la desviación típica (con el parámetro $featurewise\_std\_normalization = True$) y creará una partición de validación con el 10 % de los datos de entrenamiento (parámetro $validation\_split = 0.1$).

Como el generador utiliza normalización, hace falta entrenarlo con los datos de entrenamiento. Esto lo podemos hacer de la siguiente forma:

```
# Entrenar generadores
datagen_train.fit(x_train)
```

Con el generador ya entrenado, podemos obtener los iteradores que se van a utilizar a la hora de entrenar el modelo. Habrá un iterador para el conjunto de

entrenamiento y otro para el conjunto de validación. Obtener estos iteradores se puede hacer de la siguiente forma:

```
# Crear flow de entrenamiento y validacion
train_iter = datagen_train.flow(
    x_train,
    y_train,
    batch_size=batch_size,
    subset='training'
)

validation_iter = datagen_train.flow(
    x_train,
    y_train,
    batch_size=batch_size,
    subset='validation'
)
```

En el primer caso, creamos un iterador de entrenamiento, utilizando para ello los datos de x_train e y_train. Es aquí donde especificamos el tamaño del *batch* (recordemos que se ha establecido que sea 32), y se indica que los datos pertenecen al subconjunto de *training* (esto es porque se ha especificado en el generador que se utilice $validation\_split = 0.1$). Para el conjunto de validación el proceso es casi igual, solo que el subconjunto del que se extraerán los datos es *validation*.

Es importane destacar que a la hora de crear los iteradores (al llamar a los métodos $flow$), los datos son barajados (por defecto el parámetro $shuffle$ está puesto a **True**). Esto es importante, ya que si no, Keras solo cogería el primer 10 % de los datos como validación, lo cuál puede hacer que el conjunto de validación no represente para nada al conjunto de entrenamiento, y por tanto, los resultados obtenidos en validación sean pésimos.

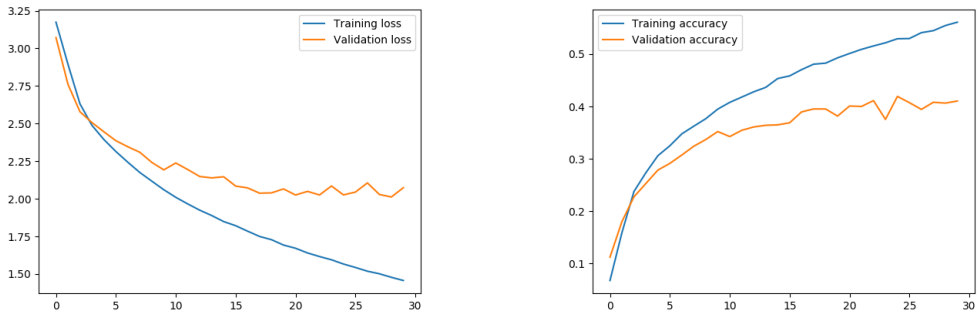
Una vez hecho esto. ya podemos entrenar el modelo. Como en este caso utilizamos generadores, no podemos utilizar el método $fit()$ tal y como hicimos anteriormente, si no que tendremos que utilizar el método $fit\_generator()$. El entrenamiento que se ha realizado se puede ver en el siguiente fragmento de código:

```
history = model.fit_generator(
    train_iter,
    steps_per_epoch=len(x_train)*0.9/batch_size,
    epochs=epochs,
    validation_data=validation_iter,
    validation_steps=len(x_train)*0.1/batch_size
)
```

Especificamos que para entrenar se va a utilizar el iterador $train\_iter$ creado anteriormente. Por cada época se van a realizar $len(x\_train) * 0.9/batch\_size$ pasos (esto es, del tamaño del conjunto de entrenamiento original se cogerá el 90 % de dicho tamaño, que representa el porcentaje de datos que se utilizarán para

entrenar, y este número de elementos se dividirá entre el tamaño del *batch*, que es 32; de esta forma se sabe cuántos pasos hay que dar para el tamaño de *batch* especificado). Luego se indica cuántas épocas se quieren entrenar (de momento, viendo los resultados que se han obtenido en el apartado anterior, vamos a conservar su valor anterior, que es 35, ya que no se produce un *overfit* que nos indique que haga falta rebajar dicho número). Se indica que como datos de validación se va a utilizar el *validation_iter* creado anteriormente y se indica el número de pasos que se van a hacer a la hora de validar, lo cuál se hace de forma similar a cómo se hizo con el conjunto de entrenamiento, solo que se utilizará el 10 % del tamaño del conjunto de entrenamiento.

Con el entrenamiento ya hecho, vamos a estudiar las gráficas para ver qué tal ha ido:



(a) Evolución de la función de pérdida.

(b) Evolución de la *accuracy*.

Figura 3: Historia del modelo *BaseNet* con normalización.

Como podemos ver, comparando los resultados con los que se pueden ver en la figura 2 los valores de pérdida son menores en el conjunto de entrenamiento que los que teníamos anteriormente. Además, el valor de *accuracy* es más alto que el que habíamos obtenido en el caso anterior para el el conjunto de entrenamiento. Sin embargo, si estudiamos los resultados obtenidos en validación, vemos que no ha habido ninguna mejora significativa, ya que los resultados son bastante parecidos a los que teníamos anteriormente.

Además de eso, si estudiamos los valores de entrenamiento y de validación de forma conjunta, podemos ver que, a diferencia de lo que sucedía en la figura 2, aquí los resultados en validación se quedan mucho más cortos cuando llegamos a las últimas épocas. Al principio, los valores van bastante pegados, pero a medida que aumentan el número de épocas, éstos se van despegando, hasta llegar al resultado final que podemos ver en las gráfias de la figura 3.

En líneas generales podemos ver que los valores de la función de pérdida en validación se quedan bastante por encima de los de entrenamiento, y la *accuracy* se queda bastante por debajo de la de entrenamiento. Aparte, en ambos parece que se estanca en las últimas épocas, mientras que los valores obtenidos en entrenamiento siguen mejorando. Por tanto, podemos detectar cierto *overfit*, ya que aunque se mejore en entrenamiento, no hay mejoras reales en validación, con lo cuál parece que el modelo está comenzando a memorizar los datos de entrenamiento. Posiblemente, de haber entrenado algunas épocas más, los valores de validación hubiesen empezado a empeorar.

Tras este breve análisis podemos ver que, a pesar de que normalizar ha permitido mejorar los resultados obtenidos en entrenamiento, los de validación aún se quedan muy cortos. Por tanto, ha habido cierta mejora, pero no significativa. Esto puede deberse a que solo se normaliza la entrada, y a media que se van haciendo convoluciones, dicha normalización se pierde. Con lo cuál, parece lógico que se tengan que introducir capas de normalización en la red para que los datos siempre estén normalizados, aunque esto lo haremos más adelante. De momento, conservaremos la normalización, ya que es una mejora que siempre ayuda y, si conseguimos combinarla con alguna otra mejora, puede que los resultados sean bastante mejores.

2.2. Aumento de datos

La siguiente mejora que podemos probar es el aumento de datos. De esta forma, a partir de los datos que tenemos para entrenar el modelo, podemos generar nuevos datos aplicándoles transformaciones, como por ejemplo rotaciones, zoom, espejo, etc. Para estudiar si existe mejora al aplicar esta técnica, vamos a probar una serie de transformaciones que utilizaremos en combinación con la normalización, ya que siempre es útil normalizar los datos de entrada. Las transformaciones a probar son las siguientes:

- ***Flip* horizontal**. Este aumento consiste en voltear la imagen en el eje horizontal. Puede ser una mejora interesante debido a que va a invertir la imagen, y por ende los elementos de la imagen, con lo cuál podemos llegar a tener la imagen normal en alguna de las épocas de entrenamiento, y la imagen volteada en otra, la cuál será tratada como una nueva muestra de entrenamiento. Por tanto, es posible que ayude a generalizar mejor.
- **Zoom**. Este aumento consiste en realizar un zoom sobre la imagen, tanto para alejarse de ella como para acercarse. Podría ser interesante aplicar este aumento ya que nos podría ofrecer información sobre una misma imagen a distintas escalas, desde más cerca o más lejos por ejemplo.

- **Rotación**. Este aumento rota la imagen en α grados en cualquiera de los dos sentidos. Es un aumento que puede ser útil cuando la imagen no ha sido tomada en condiciones óptimas (por ejemplo, con algún tipo de rotación, haciendo que los objetos estén de lado). De esta forma, se puede aprender más mirando un mismo objeto con distintas orientaciones.
- ***Flip* horizontal + zoom**. Esta es una mejora que combina tanto el *flip* com el zoom, con el objetivo de ver si al invertir la imagen y hacer zoom se puede aprender más, y por tanto, generalizar mejor, ya que se tienen nuevos elementos en el conjunto de entrenamiento a distintas esclas.

Ya que en este apartado vamos a generar nuevas imágenes, podemos probar a aumentar el número de épocas de 35 a 50, ya que con pocas épocas no se va a notar nada el aumento. Esto se debe a que los aumentos se generan sobre la marcha, y tienen una probabilidad de aparecer o no. Con lo cuál, podría darse la mala suerte de que no salgan muchos aumentos en pocas épocas, y entonces es como si estuviésemos entrenando con los datos normales. Además, así podremos ver si al insertar nuevos datos y aumentar el número de épocas, podemos disminuir un poco el *overfit* que se producía, tal y como se puede ver en la figura 3.

Antes de ver los resultados que ofrece cada uno de estos aumentos, hace falta conocer cómo se hacen los aumentos. Gracias a la clase $ImageDataGeneratos$ podemos hacerlo de forma bastante sencilla, ya que podemos especificar qué aumentos queremos hacer (recordemos que también vamos a especificar que se normalicen los datos y que vamos a crear una partición de validación).

Para especificar que queremos hacer un ***flip* horizontal**, además de aplicar normalización y crear un conjunto de validación, podemos hacerlos de la siguiente forma:

```
# Datagen con flip horizontal
datagen_train_flip = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    validation_split=0.1,
    horizontal_flip=True
)
```

Lo único diferente a lo que hacíamos anteriormente es especificar el parámetro $horizontal\_flip$ poniéndolo a **True**.

Podemos declarar un generador con **zoom** de la siguiente forma:

```
# Datagen con zoom de 0.2
datagen_train_zoom = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
```

```
    validation_split=0.1,
    zoom_range=0.2
)
```

El parámetro *zoom_range* controla la escala del zoom, tanto para alejarse como para acercarse de la imagen. En este caso, se ha especificado que dicho valor sea 0.2, un valor no muy grande y que es comunmente.

Para declarar un generador que utilice **rotación**, podemos utilizar algo como lo que se ve a continuación:

```
# Datagen con rotacion de 25
datagen_train_rot = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    validation_split=0.1,
    rotation_range=25
)
```

La rotación se especifica con el parámetro *rotation_range*. En este caso, se ha especificado que se rote como máximo 25º en cualquiera de los dos sentidos. Se ha escogido este valor porque se cree que es razonable que se realicen pequeñas rotaciones sobre la imagen en vez de rotaciones muy abruptas.

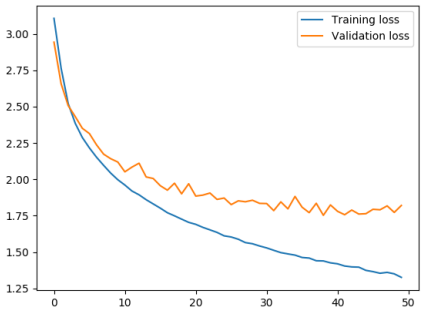
Finalmente, podemos crear un generador que combine ***flip*** **horizontal** junto con **zoom** de la siguiente forma:

```
# Datagen con flip horizontal y zoom de 0.2
datagen_train_fz = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    validation_split=0.1,
    horizontal_flip=True,
    zoom_range=0.2
)
```

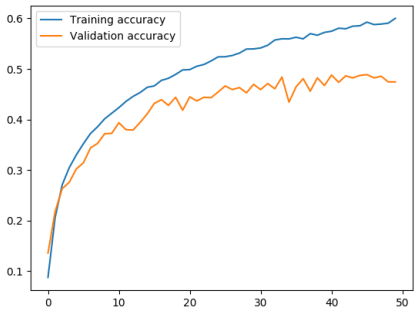
Aquí no hay nada nuevo que comentar, ya que es una combinación de dos de los generadores vistos anteriormente.

Para utilizar los generadores, de nuevo tenemos que utilizar el método $fit()$ tal y como hicimos antes, además de crear los iteradores correspondientes. Para entrenar el modelo, de nuevo podemos utilizar el método $fit\_generator()$.

Una vez comentados los aspectos generales, vamos a analizar los resultados obtenidos para cada tipo de aumento de datos, comparándolos con lo que teníamos anteriormente y entre ellos, y decidiendo cuál es el mejor para este problema.

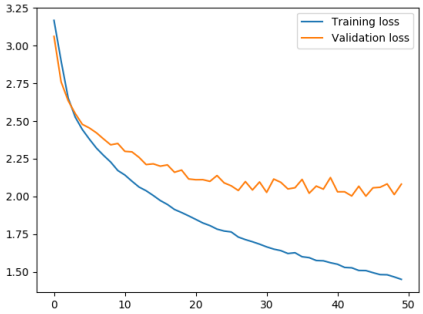


(a) Evolución de la función de pérdida.

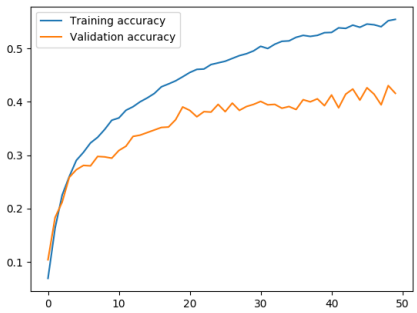


(b) Evolución de la *accuracy*.

Figura 4: Historia del aumento de datos con *flip* horizontal.

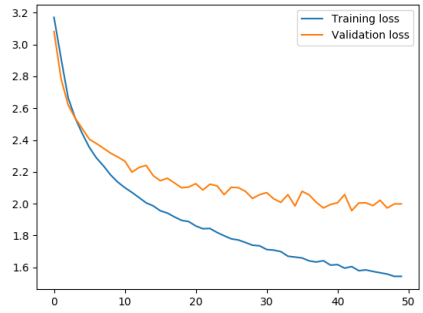


(a) Evolución de la función de pérdida.

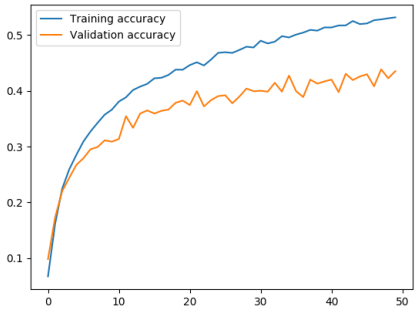


(b) Evolución de la *accuracy*.

Figura 5: Historia del aumento de datos con zoom.

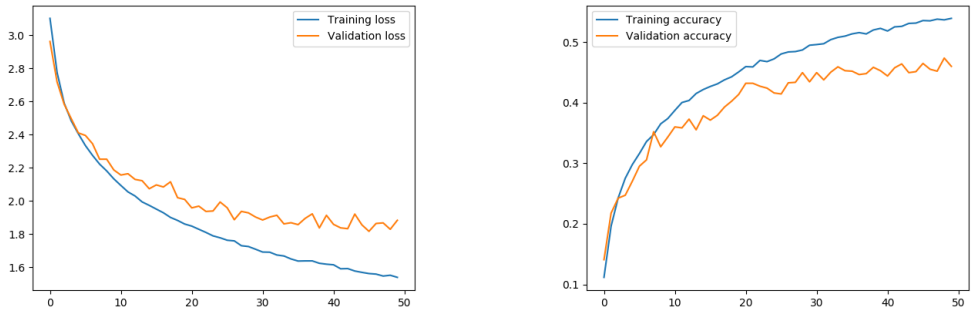


(a) Evolución de la función de pérdida.



(b) Evolución de la *accuracy*.

Figura 6: Historia del aumento de datos con rotación.



(a) Evolución de la función de pérdida. (b) Evolución de la *accuracy*.

Figura 7: Historia del aumento de datos con *flip* horizontal y zoom.

En general, si miramos los resultados, podemos ver que hay aumentos de datos, como por ejemplo el zoom y la rotación que no aportan casi ninguna mejora, ya que los resultados obtenidos para entrenamiento son bastante buenos, mientras que en validación se queda bastante corto, tanto en las gráficas de error como en la de *accuracy*. El error en validación se queda, en ambos casos, muy cerca de 2, mientras que el de entrenamiento baja hasta 1.5-1.6. La *accuracy* en validación se queda en torno a 0.4 en validación, mientras que en entrenamiento supera los 0.5 en ambos casos. Estos resultados recuerdan mucho a los que se pueden ver en la figura 3, y por tanto, no representan una mejora suficiente como para volver a utilizarlos en el futuro.

En cambio, si comparamos los resultados obtenidos para los casos en los que se utiliza *flip* horizontal, vemos que sí que se produce cierta mejora respecto a los modelos anteriores, y a los otros modelos con aumentos de zoom y rotación. En los dos modelos que se usa el *flip* horizontal el error en validación se sitúa por debajo de 2, mejorando por tanto los resultados obtenidos por el modelo base y por el modelo con solo la normalización. La *accuracy* en validación también ha sufrido un ligero incremento, situándose en ambos casos por encima de 0.4. Por tanto, el modelo mejora ciertamente al utilizar *flip* horizontal.

Ahora bien, si comparamos ambos modelos, vemos que en el caso en el que solo se hace el *flip*, los valores en validación son un poco mejores que los obtenidos al aumentar los datos con *flip* y zoom. Sin embargo, en este segundo caso, los resultados de validación se quedan más cerca de los de entrenamiento. Por tanto, en el segundo caso, se podría decir que se está generalizando mejor, a pesar de que los resultados son, en general, un poco peores. A pesar de eso, parece que en ambos modelos las curvas de validación se van a estancar en unas pocas épocas más, mientras que las de entrenamiento continuarán mejorando. Por tanto, ninguno de los dos modelos está libre de potencial *overfit*.

Si nos basamos en como de bien se puede generalizar, la opción clara, de momento, sería escoger como mejor aumento de datos el que realiza un *flip* junto con un zoom. Sin embargo, si nos atendemos a los resultados obtenidos en validación, el mejor es el que hace solo el *flip*. Como el modelo puede ser regularizado para reducir el *overfit*, vamos a escoger como **mejor aumento** el que **solo realiza un *flip* horizontal** por habernos ofrecido unos mejores resultados en validación.

2.3. Aumento de la profundidad de la red

La siguiente mejora que nos podemos plantear es aumentar la profundidad de la red. Para ello, se van a proponer dos arquitecturas. Ambas se compararán, y se elegirá la mejor. En un principio, ambas se compararán solo con la normalización, y después se mirará como son afectadas por el aumento de datos con *flip* horizontal (el aumento que mejores resultados ha proporcionado).

Ambos modelos están compuestos por dos módulos convolucionales, cada uno con dos capas convolucionales, y una capa de *max pooling* al final. Además, en ambos modelos se ha pasado a tener 3 capas totalmente conectadas, formando una red de $128 \times 50 \times 25$ neuronas que se encargarán de clasificar según la información extraída por la red convolucional, y en los dos modelos el número de canales va creciendo a medida que se van haciendo convoluciones. Al tener una arquitectura como esta, no se reduce el tamaño de las imágenes tan rápidamente como pasaba antes, ya que antes de cada *max pool* hay dos convoluciones en vez de una.

A pesar de las similitudes que tienen los dos modelos, existen dos diferencia importante entre ellos: el tamaño del *kernel* de las convoluciones y el número de canales de salida:

- En el caso de la primera red, el primer módulo convolucional está compuesto por dos convoluciones 5×5, con 8 y 16 canales de salida, respectivamente. El segundo módulo está compuesto por dos capas convolucionales con *kernel* de tamaño 3×3, y con 32 y 64 canales de salida.
- En la segunda red los dos módulos utilizan convoluciones de tamaño 3×3, y el número de canales es 16 y 32 para el primer módulo convolucional y de 64 y 64 para el segundo módulo.

En el primer modelo, las convoluciones del primer módulo se han hecho así para imitar las que hace *BaseNet*, solo que se ha cambiado el número de canales de salida de la primera convolución. Las del segundo módulo tienen un tamaño de 3×3 para no reducir demasiado la imagen, además de que las convoluciones de este tamaño son más rápidas que las 5×5. El número de canales de salida es siempre

una potencia de 2. No hay ningún motivo para que sea un número potencia de 2, ya que la red no va a obtener mejores resultados por tener canales de salida de tamaño potencia de 2.

En el segundo modelo, todas las convoluciones son de 3×3 porque son menos costosas desde el punto de vista computacional. Todos los canales de salida de las capas convolucionales son, de nuevo, potencias de 2. Sin embargo, a diferencia del modelo anterior, el número de canales de la primera capa convoluciona es de 16, y se incrementa hasta un máximo de 64, ya que se cree que no tiene sentido hacer canales más grandes debido a que éstos pueden llevar a un *overfit* excesivo sin regularizar.

Una vez que hemos comentado los aspectos generales de ambas arquitecturas, vamos a ver cómo sería la implementación de ellas. El código correspondiente al primer modelo es el siguiente:

```
# Definicion del nuevo modelo
model_v2 = Sequential()
model_v2.add(Conv2D(8, kernel_size=(5, 5), padding='valid',
    input_shape=input_shape))
model_v2.add(Activation('relu'))
model_v2.add(Conv2D(16, kernel_size=(5, 5), padding='valid'))
model_v2.add(Activation('relu'))
model_v2.add(MaxPooling2D(pool_size=(2, 2)))

model_v2.add(Conv2D(32, kernel_size=(3, 3), padding='valid'))
model_v2.add(Activation('relu'))
model_v2.add(Conv2D(64, kernel_size=(3, 3), padding='valid'))
model_v2.add(Activation('relu'))
model_v2.add(MaxPooling2D(pool_size=(2, 2)))

model_v2.add(Flatten())
model_v2.add(Dense(units=128))
model_v2.add(Activation('relu'))
model_v2.add(Dense(units=50))
model_v2.add(Activation('relu'))
model_v2.add(Dense(units=25))
model_v2.add(Activation('softmax'))
```

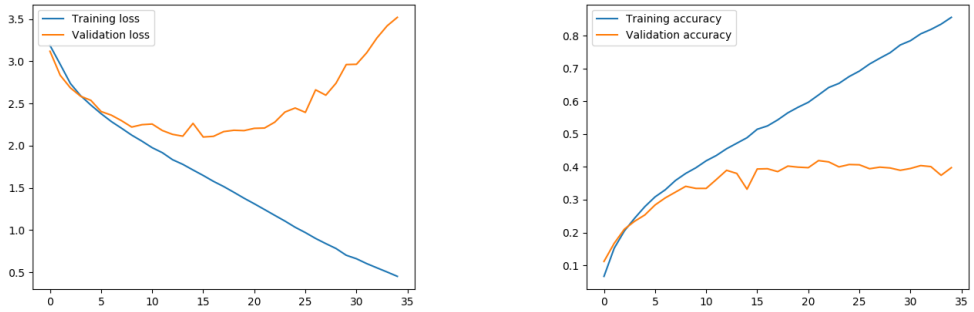
El segundo modelo se ha declarado de la siguiente forma:

```
# Definicion del nuevo modelo
model_v3 = Sequential()
model_v3.add(Conv2D(16, kernel_size=(3, 3), padding='valid',
    input_shape=input_shape))
model_v3.add(Activation('relu'))
model_v3.add(Conv2D(32, kernel_size=(3, 3), padding='valid'))
model_v3.add(Activation('relu'))
model_v3.add(MaxPooling2D(pool_size=(2, 2)))

model_v3.add(Conv2D(64, kernel_size=(3, 3), padding='valid'))
```

```
model_v3.add(Activation('relu'))
model_v3.add(Conv2D(64, kernel_size=(3, 3), padding='valid'))
model_v3.add(Activation('relu'))
model_v3.add(MaxPooling2D(pool_size=(2, 2)))

model_v3.add(Flatten())
model_v3.add(Dense(units=128))
model_v3.add(Activation('relu'))
model_v3.add(Dense(units=50))
model_v3.add(Activation('relu'))
model_v3.add(Dense(units=25))
model_v3.add(Activation('softmax'))
```

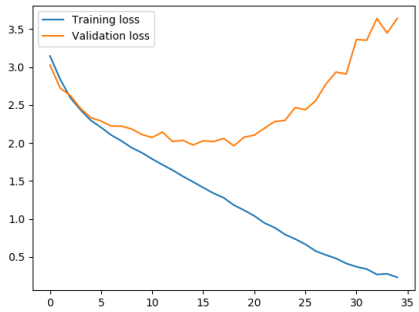
Una vez compilados los modelos, vamos a entrenarlos y a ver los resultados que obtenemos. Es importante destacar que, para entrenar estos modelos, el número de épocas ha sido reducido 35. De esta forma, si los modelos tardan más en entrenarse, no se realizan tantas épocas, y por tanto el tiempo de entrenamiento es menor. Además, como vimos en la sección anterior, al haber entrenado 50 épocas, los modelos estaban muy cerca de padecer de sobreajuste, con lo cuál parece sensato reducir el número de épocas para intentar evitar que esto suceda para estos nuevos modelos.

Una vez comentado este detalle, vamos a pasar a estudiar los resultados obtenidos.

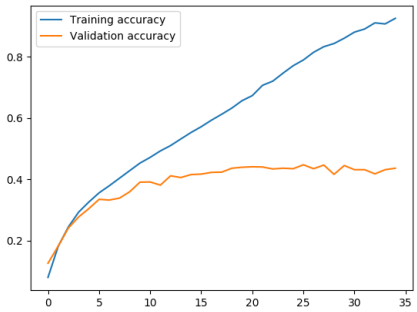


(a) Evolución de la función de pérdida. (b) Evolución de la *accuracy*.

Figura 8: Historia del primer modelo profundo.



(a) Evolución de la función de pérdida.



(b) Evolución de la *accuracy*.

Figura 9: Historia del segundo modelo profundo.

Se puede observar a simple vista que los resultados obtenidos son muy malos. En ambos casos, el *overfit* es evidente, ya que los valores del error en validación empiezan a subir a partir de las 15 épocas, mientras que los valores del error en entrenamiento sigue constante. Esto también se refleja en las gráficas de *accuracy*, donde podemos ver que la precisión en el conjunto de entrenamiento va mejorando, mientras que la del conjunto de validación se queda estancada y no mejora nada. Toda esta situación llevará a que la red no sea capaz de generalizar correctamente, ya que al existir sobreajuste, la red acaba memorizando los datos de entrenamiento, y obtendrá unos resultados pésimos con nuevos datos que nunca antes ha visto.

Este problema se debe a que hemos aumentado la profundidad de la red, y con eso, el número de parámetros de la red. Estos parámetros pueden tomar cualquier valor, ya que no hay ninguna restricción sobre ellos. Cuando sucede esto, los parámetros se pueden "descontrolar", tomando valores que le permiten ajustarse más a los datos de entrenamiento, y por tanto, disminuir el error en el conjunto de entrenamiento, ya que es lo que se está intentando minimizar. Esto nos llevará a unos resultados como los que podemos ver en las figuras 8 y 9, donde los resultados en el conjunto de entrenamiento son muy buenos mientras que los de validación son muy malos.

Este problema puede ser difícil de solventar, ya que no hay una única manera de intentar reducir el *overfit*. Una idea simple sería parar el entrenamiento antes de que empiece a sobreajustar. En este caso, sería parar el entrenamiento alrededor de las 10-15 épocas. Sin embargo, si hacemos eso, el modelo que obtendremos será bastante malo, ya que no habrá aprendido lo suficiente debido a que los valores del error y la precisión serán bastante malos en general. Por tanto, para solventar este problema podemos probar a utilizar una técnica de regularización como **Dropout**, la cuál procederemos a ver a continuación.

Referencias

[1] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.