UNIVERSIDAD AUTONOMA DE MADRID

Práctica 2

Redes Convolucionales TensorFlow

Master Big Data y Data Science

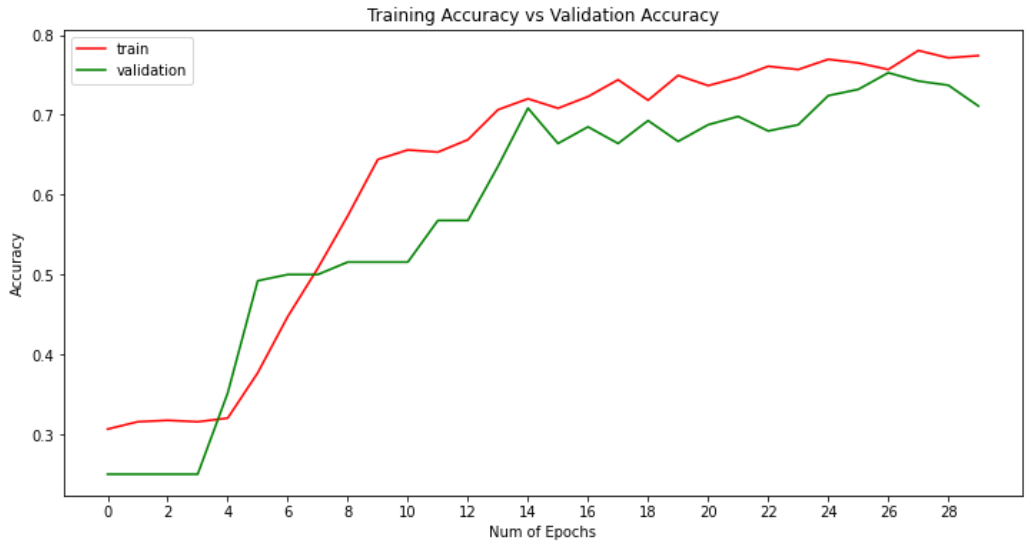
Antonio Sanz Corbalán

28 Marzo de 2020

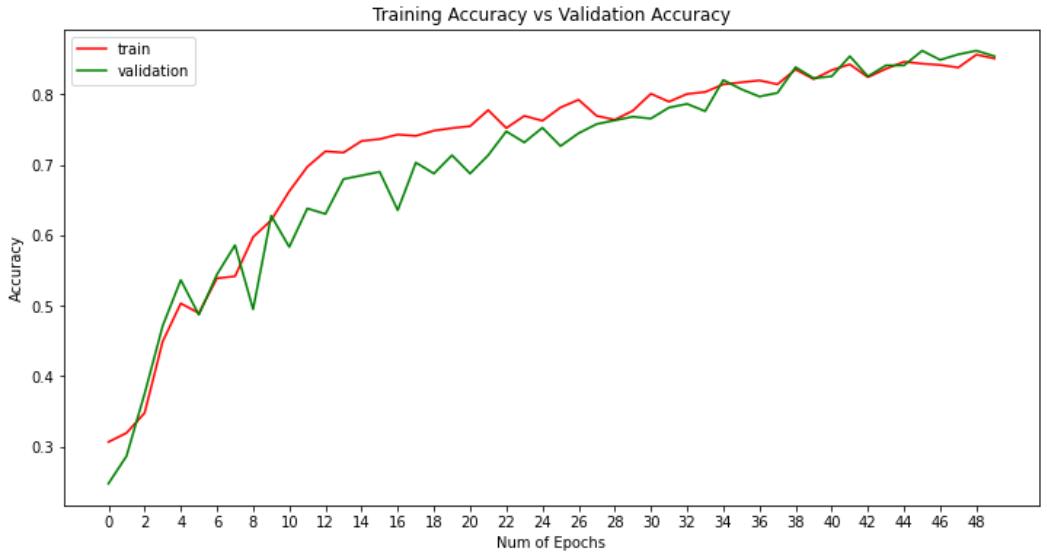
Epochs

Una época es un hiperparámetro definido antes de entrenar nuestro modelo. Una época transcurre cuando nuestro conjunto de datos entero ha pasado hacia delante y hacia atrás por nuestra red neuronal. A continuación muestro el resultado para 30 y 50 épocas. Con un mayor número de épocas permitimos a nuestro modelo calibrar o ajustar mejor los pesos. Si utilizamos una función callback que detenga el entrenamiento cuando ya no estemos mejorando los resultados, podemos determinar de forma más precisa el número de épocas necesario.

****30****



****50****

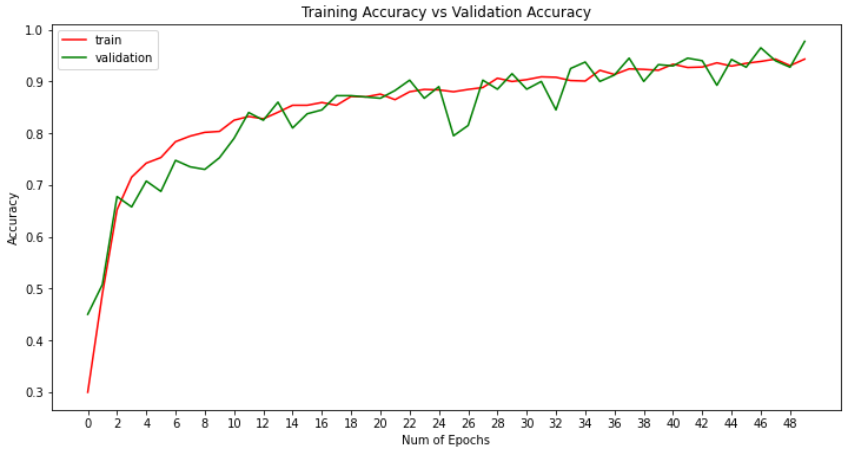


Batch_size y Steps_per_epoch

El tamaño del batch determina el número total de muestras de entrenamiento presentes en un batch. No podemos hacer pasar todo el conjunto de datos al mismo tiempo a través de la red, es por eso que definimos un tamaño de batch.
El parámetro steps_per_epoch determina el número de iteraciones batch realizadas durante cada época.
En las gráficas que muestro a continuación se aprecia que para un batch size de 16 los resultados son mejores. Hay que tener en cuenta que he definido el número de steps por época como el número de muestras de entrenamiento entre el batch size. De modo que en este apartado también estamos valorando de alguna forma la importancia del hiperparámetro steps_per_epoch. En el primer caso tenemos 70 y en el segundo 35 steps_per_epoch.

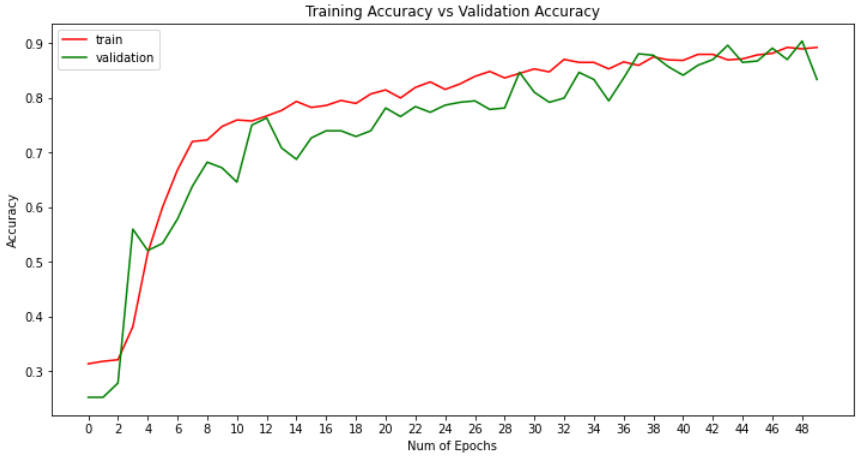
Batch size 16 y Steps per epoch 70

```
70/70 [==============================] - 12s 171ms/step - loss: 0.1427 - accuracy: 0.9432 - val_loss: 0.1180 - val_accuracy: 0.9775
```



Batch size 32 y Steps per epoch 35

```
35/35 [==============================] - 12s 337ms/step - loss: 0.3057 - accuracy: 0.8920 - val_loss: 0.3830 - val_accuracy: 0.8333
```



ImageDataGenerator

La clase ImageDataGenerator nos permite aumentar nuestro dataset de entrenamiento mediante modificaciones como por ejemplo rotaciones, variación del brillo, variación de contraste, zoom y muchas otras. De esta forma logramos que el entrenamiento de nuestro modelo sea más robusto frente a los datos de validación. [1]

```
train_datagen = ImageDataGenerator(
   rescale=1./255,
   shear_range = 0.1,
   zoom_range = [0.2,0.5],
   rotation_range = 5,
   horizontal_flip = True,
   brightness_range = [0.2,1.0])
```

En mi caso, después de hacer varias pruebas he seleccionado aumentar el dataset de entrenamiento haciendo variaciones en el zoom de la imagen original, ligeras rotaciones de 5 grados, giros de imagen con respecto a su horizontal y variaciones en el brillo de la imagen.

Learning Rate

Para evitar que los pesos de nuestro modelo de red convolucional cambien demasiado rápido y no logremos alcanzar los valores óptimos de nuestros pesos, entonces utilizamos la variable **learning_rate** o tasa de aprendizaje. Esta variable suele inicializarse con un valor muy pequeño (0.001), de esta forma nos aseguramos que cualquier cambio en los pesos de nuestro modelo no sean muy bruscos y podemos converger en pesos óptimos.[3]

```
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
factor=0.1, patience=4, verbose=1, min_delta=1e-4)
```

Callbacks

Los callbacks son una serie de funciones que aplicamos en un momento determinado en la etapa de entrenamiento. [2]
En mi caso he utilizado tres funciones callbacks:

La función **early_stop** definida comprueba continuamente el valor **'val_loss'** y comprueba continuamente si este valor ha disminuido al menos 0,0001 durante 4 épocas.

```
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
patience=4, verbose=1, min_delta=1e-4)
```

La función **reduce_lr** es similar a la anterior pero relacionada con la tasa de aprendizaje. De la misma forma que antes, en caso de mejorar la precisión y reducir la función de pérdidas en tres épocas entonces multiplicamos la tasa de aprendizaje por un factor 0,1. (new_lr = lr*factor).

```
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
factor=0.1, patience=3, verbose=1, min_delta=1e-4)
```

Por último hemos definido una clase **MinAccuracy** con un método que controle el valor de precisión accuracy y cuando superemos el valor definido **ACCURACCY_THRESHOLD** parar el entrenamiento.

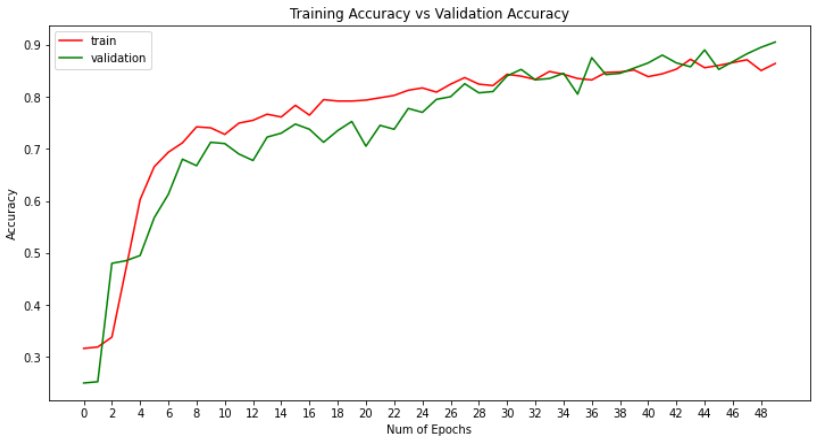
```
class MinAccuracy(tf.keras.callbacks.Callback):
 def on_epoch_end(self, epoch, logs={}):
   if(logs.get('accuracy') > ACCURACY_THRESHOLD):
     print("\nSe ha alcanzado un precisión del %2.2f%% , paramos
entrenamiento!!" %(ACCURACY_THRESHOLD*100))
     self.model.stop_training = True
```

Optimizadores

Durante el proceso de entrenamiento modificamos continuamente los pesos de nuestro modelo tratando de reducir al máximo nuestra función de pérdidas. Los optimizadores se encargan de moldear nuestra red convolucional de la forma precisa modificando los pesos y ajustando en base a la función de pérdidas. Hay muchos tipos de optimizadores, la mayoría de ellos basados en descendiente por gradiente. Yo he utilizado 4 de ellos: Stochastic gradient descent, Adam, Adamax y RMSprop. He obtenido mejores resultados con Adam y RMS ya que con ellos he logrado alcanzar el 94% de precisión entrenamiento, sin embargo con Adam lo alcanzo más temprano, en la época 36. [4]

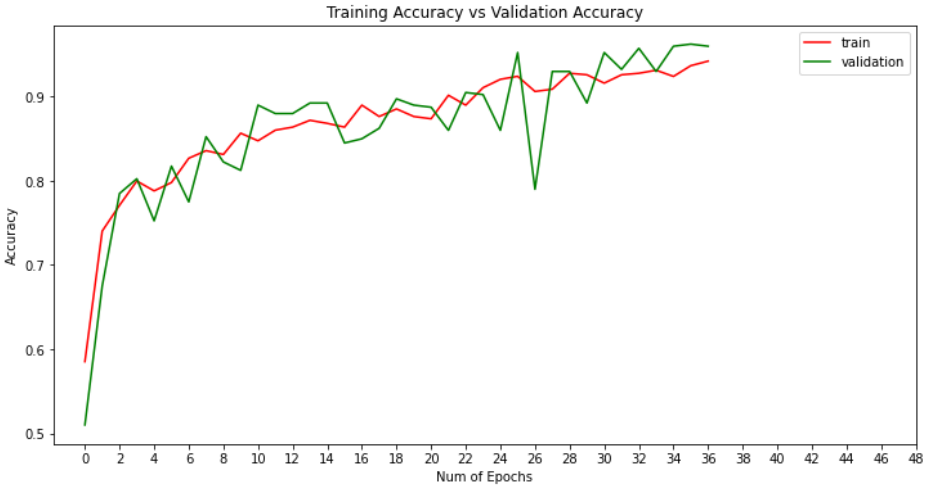
SGD

```
70/70 [==============================] - 24s 343ms/step - loss: 0.3561 - accuracy: 0.8638 -
val_loss: 0.3219 - val_accuracy: 0.9050
```



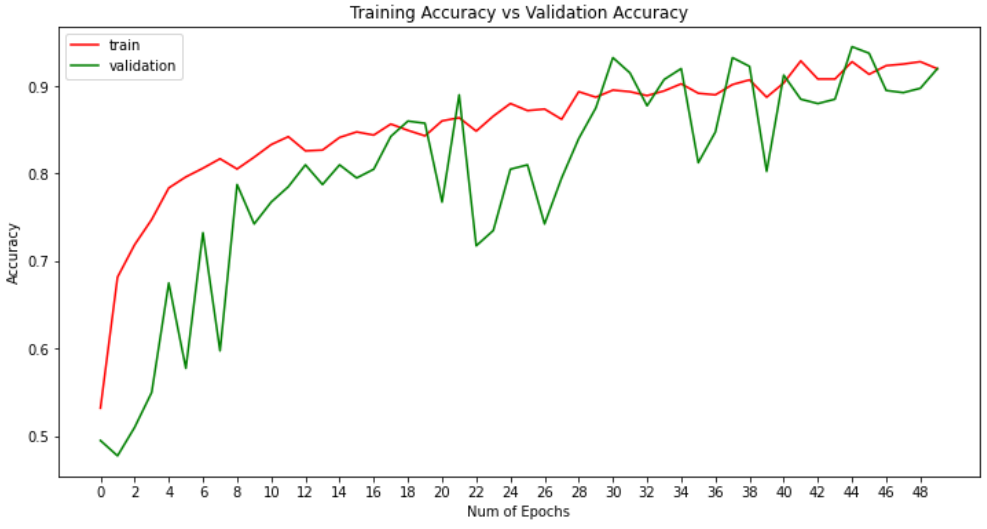
Adam

```
Se ha alcanzado un precisión del 94.00% , paramos entrenamiento!!
70/70 [==============================] - 24s 338ms/step - loss: 0.1582 - accuracy: 0.9423 -
val_loss: 0.1067 - val_accuracy: 0.9600
```



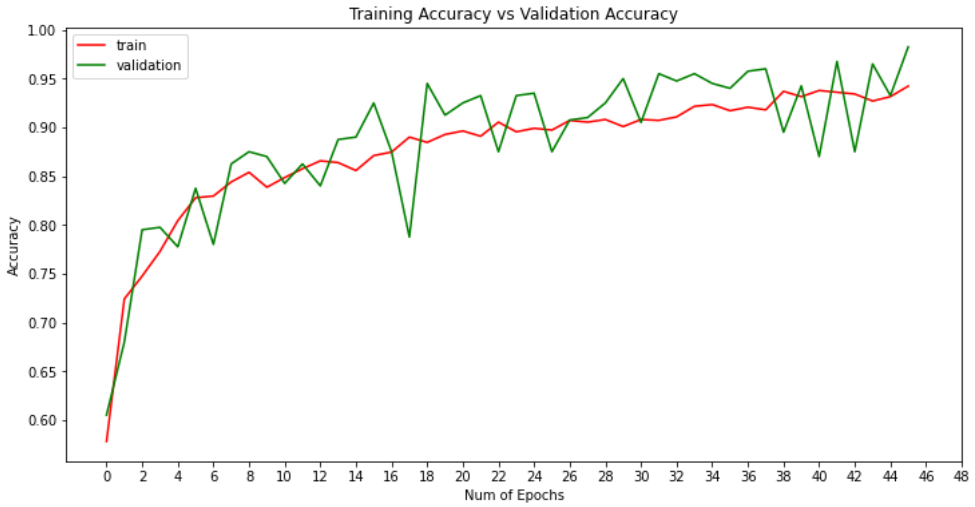
Adamax

```
70/70 [==============================] - 24s 336ms/step - loss: 0.2235 - accuracy: 0.9197 -
val_loss: 0.2365 - val_accuracy: 0.9200
```



RMS

```
Se ha alcanzado un precisión del 94.00% , paramos entrenamiento!!
70/70 [==============================] - 24s 340ms/step - loss: 0.1487 - accuracy: 0.9423 -
val_loss: 0.0623 - val_accuracy: 0.9825
```



Arquitecturas red

Después de realizar varias modificaciones sobre la arquitectura de nuestra red he decidido utilizar una red con tres capas convolucionales. Las primeras dos capas contienen 32 filtros con kernel de 3x3 y la última capa contiene 64 filtros de tamaño 3x3 también. A continuación de cada una de las capas convolucionales he introducido capas MaxPooling para minimizar el número de parámetros, el coste computacional y la posibilidad de que haya overfitting con los datos de overfitting. Otra manera de impedir que haya overfitting es utilizando capas Dropout que fijen a cero un porcentaje (20% en mi caso) de las entradas durante el entrenamiento. A la salida de la red utilizamos Flatten para vectorizar los datos y pasarlos a las capas Dense. Por último clasificamos con una función softmax la clase con mayor probabilidad: sunny, rain, shine o cloudy.

```
modelo = tf.keras.Sequential()
modelo.add(tf.keras.layers.Convolution2D(32, 3, 3,
input_shape=(IMAGE_RESIZE, IMAGE_RESIZE, 3), padding='same',
activation='relu'))
modelo.add(tf.keras.layers.Dropout(0.2))
modelo.add(tf.keras.layers.Convolution2D(32, 3, 3, activation='relu'))
modelo.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
modelo.add(tf.keras.layers.Convolution2D(64, 3, 3, activation='relu'))
modelo.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
modelo.add(tf.keras.layers.Flatten())
modelo.add(tf.keras.layers.Dense(512, activation='relu'))
modelo.add(tf.keras.layers.Dense(4, activation='softmax'))
```

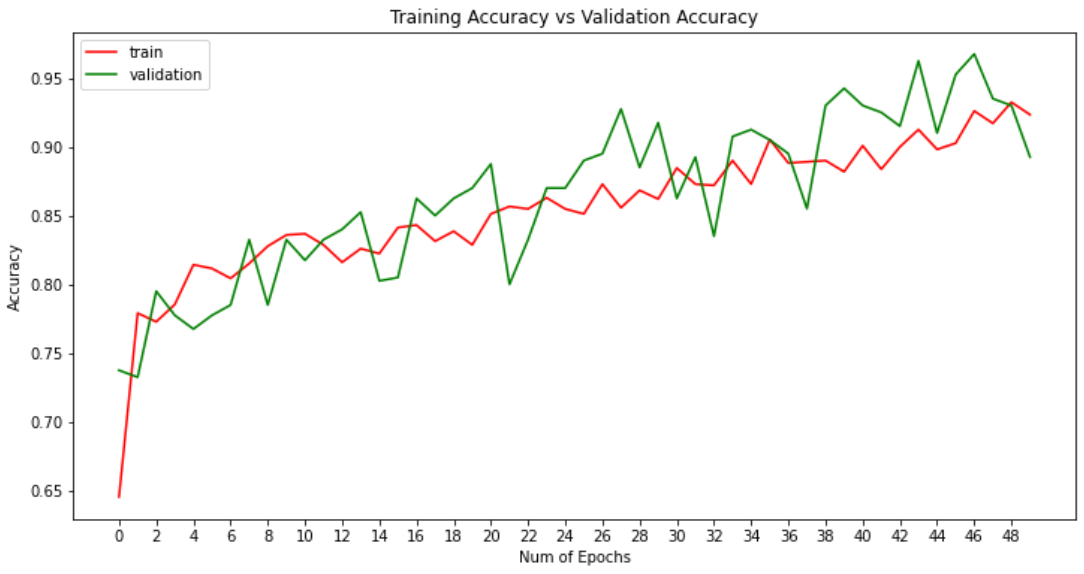
Resumen CNN

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 75, 75, 32)        896
_________________________________________________________________
dropout (Dropout)            (None, 75, 75, 32)        0
```

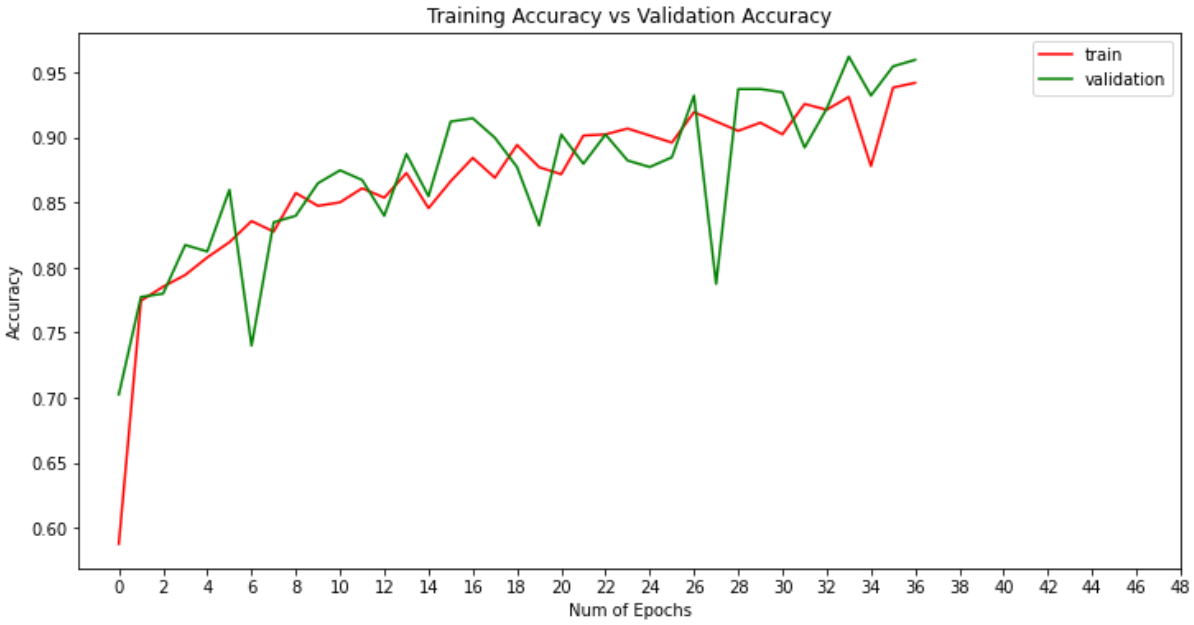
```
_________________________________________________________________


conv2d_1 (Conv2D)            (None, 25, 25, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 512)               131584
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 2052
=================================================================
Total params: 162,276
Trainable params: 162,276
Non-trainable params: 0
```

He probado también con la arquitectura de red CIFAR-10 clasificando 4 clases pero los resultados obtenidos han sido algo peores comparadas con la red anterior. En validación alcanza porcentajes de precisión altos pero en entrenamiento solo llega al 92%. [5]



En Resumen después de realizar todo el estudio probando con distintos valores para los hiperparámetros, generadores de imágenes, optimizadores, callbacks y arquitecturas de red. Por tanto concluyó que el mejor resultado lo obtengo utilizando un optimizador adam, generando imágenes mediante técnicas de horizontal flip, zoom y brightness, con tamaño de batch igual a 16, inicializando la tasa de aprendizaje en 0,001 y redimensionando las imágenes del dataset a un tamaño de 224x224 pixeles. Además he inicializado el número de épocas en 50 a pesar de que 37 son suficientes para alcanzar el 94% de precisión en entrenamiento, el número de pasos de validación (validation_steps) lo he fijado en 25 y el número de pasos por época en 70 (steps_per_epoch).



Documentación

[1] https://keras.io/preprocessing/image/
[2] https://keras.io/callbacks/
[3] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler
[4] https://keras.io/optimizers/
[5] https://keras.io/examples/cifar10_resnet/