Reconocimiento de Imágenes con Redes Neuronales

0. Índice

1. Cifar-10

1.1 Información sobre el dataset cifar-10

Este conjunto de datos está orientado al reconocimiento de imágenes. Contiene 60.000 imágenes de 32x32 píxeles y hay 10 clases diferentes. Hay 6000 imágenes por cada clase y el 20% de todos los datos está destinado a test.

```
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
                'frog', 'horse', 'ship', 'truck']
```

A continuación un ejemplo del tipo de imágenes que podemos encontrar en este conjunto de datos:

Este dataset ya viene limpio y preparado para su uso, aún así, requiere un par de pasos de preprocesamiento, que son redimensionar y normalizar las imágenes.

```
# RESHAPE
x_train_full = x_train_full.reshape(x_train_full.shape[0], 32, 32, 3)
x_test = x_test.reshape(x_test.shape[0], 32, 32, 3)
# NORMALIZATION
x_train_full = x_train_full / 255.0
x_test = x_test / 255.0
```

Además de esto, aunque no es necesario, es útil convertir la columna de clase a categórica para poder hacer uso de técnicas de regularización que mencionaremos más adelante.

```
y_train_full = keras.utils.to_categorical(y_train_full, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

La importación del dataset la he realizado directamente desde la librería de keras. He hecho una división de los datos en entrenamiento y test. Además, los datos de entrenamiento están divididos a su vez en entrenamiento (90%) y validación(10%).

```
(x_train_full, y_train_full), (x_test, y_test) =
keras.datasets.cifar10.load_data()

x_train, x_validation, y_train, y_validation =
train_test_split(x_train_full, y_train_full, test_size=0.1,
random_state=5432)
```

1.2 Red neuronal

Para entrenar el modelo, he empleado una red neuronal convolucional de 8 capas (6 ocultas). Esta red neuronal cuenta con un total de 1.343.018 parámetros. En cuanto a **regularización**, he utilizado diversas técnicas, que son:

- **Dropout**
- **Batch Normalization**
- **L2 regularizer**

He probado con varios optimizadores diferentes, pero el que acabó dando mejores resultados fue el optimizador Adam. La función de pérdida usada es **CategoricalCrossentropy.**

```
# OPTIMIZADORES
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
adam = Adam(learning_rate=0.002, decay=0, beta_1=0.9, beta_2=0.999,
epsilon=1e-08)
rms = RMSprop(learning_rate=0.003)
model.compile(optimizer=adam, loss='categorical_crossentropy',
metrics=['accuracy', 'AUC'])
```

El modelo fue entrenado con un batch size de 128.

Resumen de la Red Neuronal:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896

batch_normalization (BatchN  (None, 32, 32, 32)        128
ormalization)

conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248

batch_normalization_1 (Batc  (None, 32, 32, 32)        128
hNormalization)

max_pooling2d (MaxPooling2D  (None, 16, 16, 32)        0
)

dropout (Dropout)            (None, 16, 16, 32)        0

conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496

batch_normalization_2 (Batc  (None, 16, 16, 64)        256
hNormalization)

conv2d_3 (Conv2D)            (None, 16, 16, 64)        36928

batch_normalization_3 (Batc  (None, 16, 16, 64)        256
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)          0
2D)

dropout_1 (Dropout)          (None, 8, 8, 64)          0

conv2d_4 (Conv2D)            (None, 8, 8, 128)         73856

batch_normalization_4 (Batc  (None, 8, 8, 128)         512
hNormalization)

conv2d_5 (Conv2D)            (None, 8, 8, 128)         147584

batch_normalization_5 (Batc  (None, 8, 8, 128)         512
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)         0
2D)

dropout_2 (Dropout)          (None, 4, 4, 128)         0

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 512)               1049088

dropout_3 (Dropout)          (None, 512)               0

dense_1 (Dense)              (None, 10)                5130

=================================================================
Total params: 1,343,018
Trainable params: 1,342,122
Non-trainable params: 896
_________________________________________________________________
```

Otra técnica fundamental para poder conseguir un buen resultado y evitar el overfitting es usar ***Data Augmentation***. Esta técnica consiste en aumentar de manera artificial la cantidad de datos de entrenamiento (rotando, desplazando, invirtiendo, etc) las imágenes ya existentes para crear 'nuevas'.

```
# Configuration for creating new images
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
    rotation_range=20,
    horizontal_flip=True,
)
```

Para evitar el sobreentrenamiento, también he empleado dos callbacks, el primero es el ***EarlyStopping*** (Detener el entrenamiento en el momento en que se deja de mejorar el resultado para los datos de validación) y ***ReduceLROnPlateau*** (Ir reduciendo el learning rate si el modela se empieza a sobreentrenar).

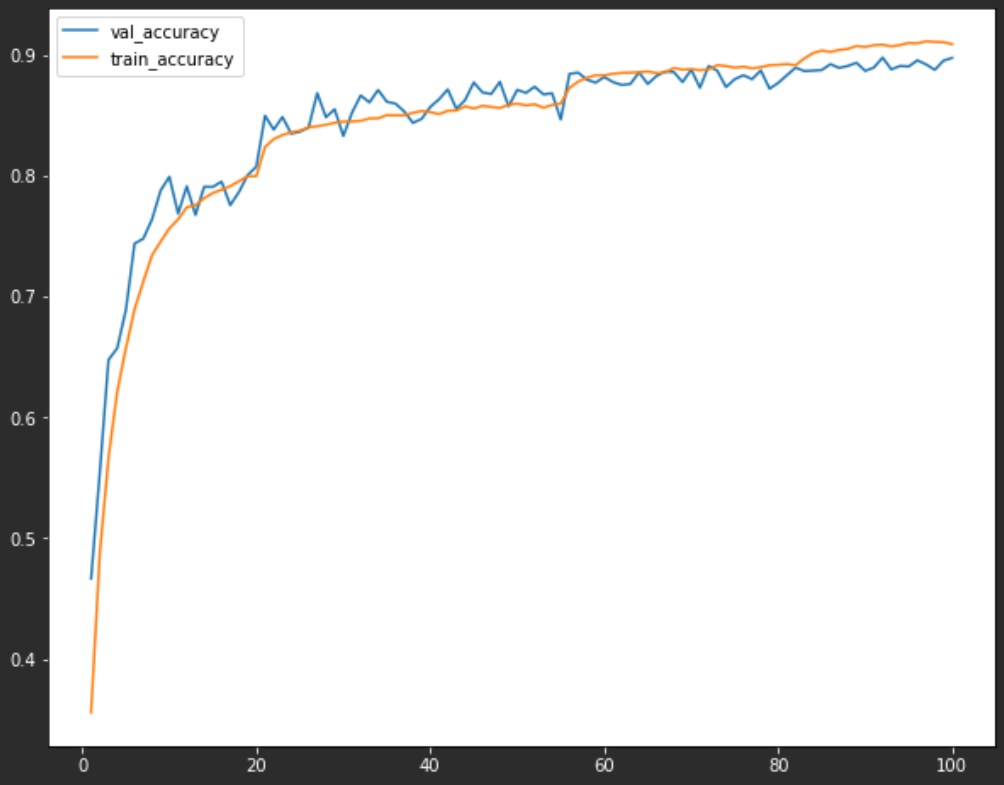
```
#early stopping to monitor the validation loss and avoid overfitting
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
mode='min', verbose=1, patience=25, restore_best_weights=True)
#reducing learning rate on plateau
rlrop = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
mode='auto', patience=10, factor= 0.5, min_lr= 1e-6, verbose=1)
#training
history = model.fit(train_datagen.flow(x_train, y_train,
batch_size=128), epochs=100, steps_per_epoch=len(x_train)/128,
validation_data=(x_test, y_test), callbacks=[early_stop, rlrop])
```

1.3 Métricas y rendimiento del modelo

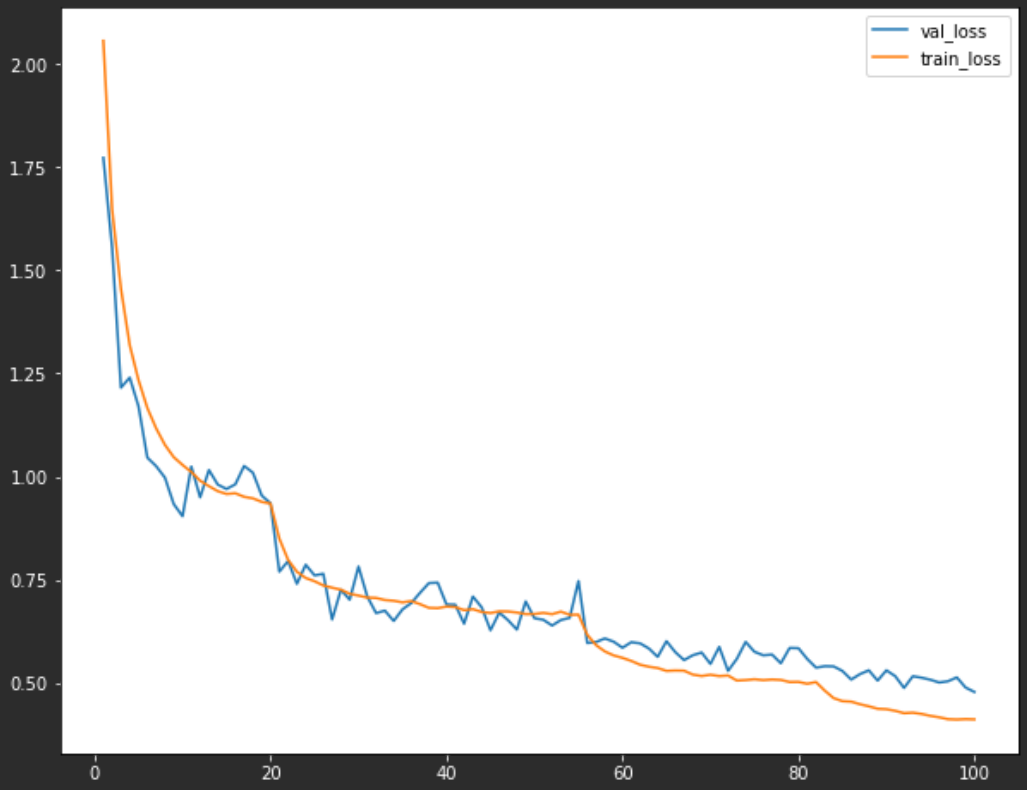
Este modelo alcanza en los datos de test una tasa de aciertos del **90%**, un loss de **0.4778** y un AUC de **0.9899**. Ha sido entrenado durante **100 épocas** y el entrenamiento ha tardado en total unas **5-6 horas**.

A continuación, las gráficas que representan el rendimiento del modelo conforme se iba entrenando durante las 100 épocas.

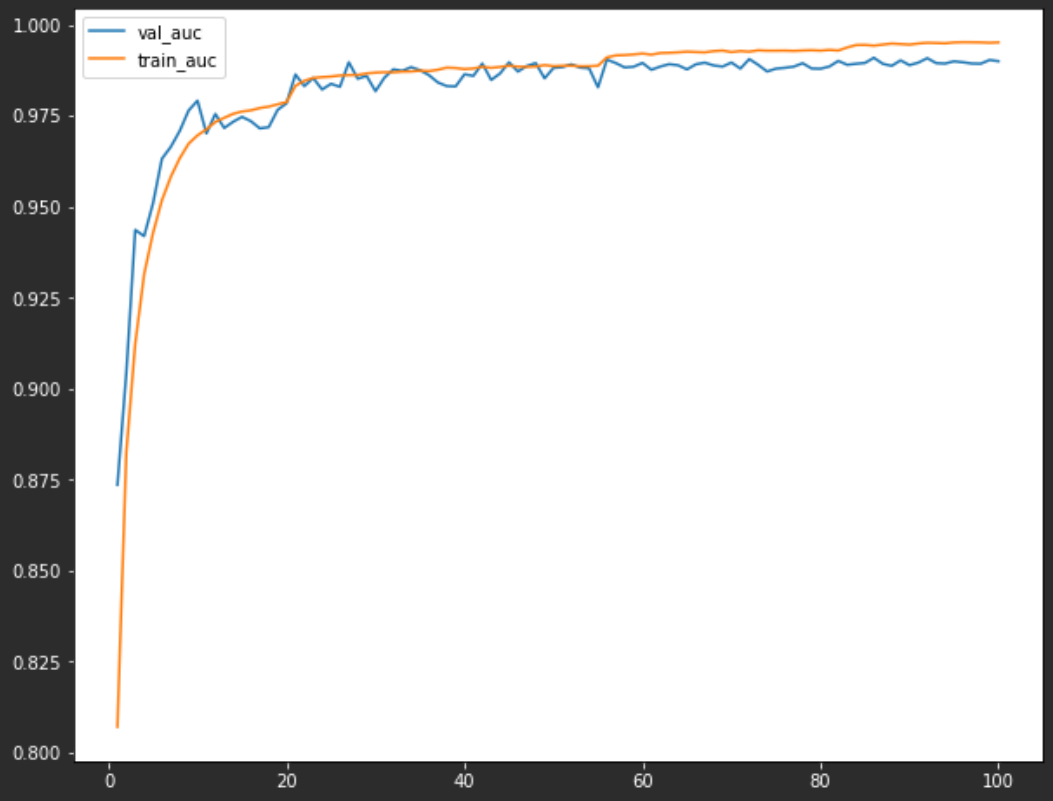
Gráfica del porcentaje de aciertos en entrenamiento frente al de validación



Gráfica de la función loss en entrenamiento frente a la de validación



Área bajo la curva ROC de entrenamiento frente a validación



1.4 Prueba del modelo

Para probar el modelo, tenemos que hacer una llamada al servidor de Flask local que se arranca en el **puerto 5000** al ejecutar el archivo **app.py.**

```
@application.route('/cifar10', methods=['POST'])
```

Debemos pasarle un objeto JSON, con un atributo llamado ‘key’, cuyo valor será el nombre de la imagen que queramos clasificar, ubicada en “/CIFAR_PROJECT/img”.

```
{
    "key": "truck.jpg"
}
```

La imagen que vamos a probar es de un camión, que, lógicamente, no se encuentra en el dataset de entrenamiento. De hecho esta imagen está sacada directamente de internet y redimensionada a 32x32.

Si hacemos la prueba, podemos ver como el modelo está un 99.9% seguro de que la foto es de un camión.

```
Predicción 1:  truck (99.9038 %)
Predicción 2:  automobile (0.0953 %)
Predicción 3:  ship (0.0008 %)
```

```
{
    "key": "dog.jpg"
}
```

Vamos a realizar una segunda prueba con esta otra foto, en este caso de un perro, también sacada de internet y redimensionada manualmente.

En este caso, nuestro modelo está prácticamente 100% seguro de que esta imagen pertenece a la clase *dog*.

```
Predicción 1:  dog (99.9999 %)
Predicción 2:  horse (0.0 %)
Predicción 3:  bird (0.0 %)
```

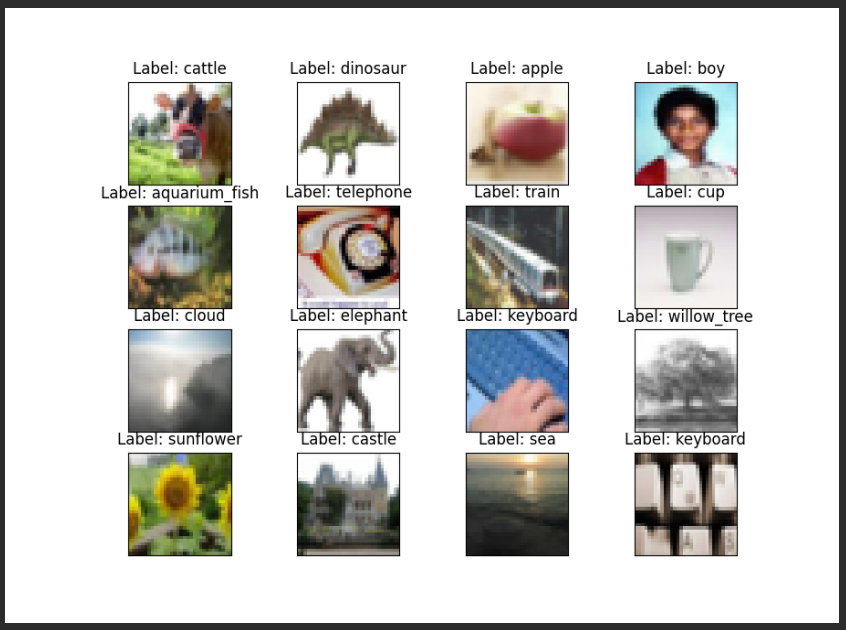
2. Cifar-100

2.1 Información sobre el dataset cifar-10

Este dataset es exactamente igual al anterior, solo que en este caso contamos con 100 clases y 600 imágenes por cada clase.

```
labels = ['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed',
   'bee', 'beetle', 'bicycle', 'bottle', 'bowl', 'boy', 'bridge',
    'bus', 'butterfly', 'camel', 'can', 'castle', 'caterpillar',
  'cattle', 'chair', 'chimpanzee', 'clock', 'cloud', 'cockroach',
    'couch', 'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin',
'elephant', 'flatfish', 'forest', 'fox', 'girl', 'hamster', 'house',
  'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain',
   'mouse', 'mushroom', 'oak_tree', 'orange', 'orchid', 'otter',
'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate',
'poppy', 'porcupine', 'possum', 'rabbit', 'raccoon', 'ray', 'road',
    'rocket', 'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk',
 'skyscraper', 'snail', 'snake', 'spider', 'squirrel', 'streetcar',
    'sunflower', 'sweet_pepper', 'table', 'tank', 'telephone',
    'television', 'tiger', 'tractor', 'train', 'trout', 'tulip',
  'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman',
                                'worm']
```

A continuación un ejemplo del tipo de imágenes que podemos encontrar en este conjunto de datos:



En este caso el dataset no lo hemos importado directamente desde keras, sino que hemos descargado los ficheros binarios y los abrimos con pickle. Las imágenes son redimensionadas y normalizadas como con CIFAR-10.

```
# File that contains data
train_file = r'datasets\cifar-100-python\train'
# Open the file
train_images = unpickle(train_file)
# Extract the data
X_train_images = train_images['data']
# Reshape the whole image data
X_train_images = X_train_images.reshape(len(X_train_images),3,32,32)
# Transpose the whole data
X_train_images = X_train_images.transpose(0,2,3,1)
# NORMALIZATION
X_train_images = X_train_images / 255.0
```

También está dividido el dataset en **20%** para **test** y **80%** para **entrenamiento**, este último dividido también dejando un **10%** para **validación**.

```
# Convertimos la columna de clase a categórica para poder usar data
augmentation
X_train_labels = keras.utils.to_categorical(X_train_labels, 100)
X_test_labels = keras.utils.to_categorical(X_test_labels, 100)

# VALIDATION SPLIT
X_train, X_validation, y_train, y_validation =
train_test_split(X_train_images, X_train_labels, test_size=0.1,
random_state=5432)
```

2.2 Red neuronal

La red neuronal usada para este conjunto es prácticamente igual que la usada para CIFAR-10. La única diferencia es que son 9 capas en vez de 8 y el número total de parámetros es de 2.965.124.

En cuanto a técnicas de regularización y optimización, he empleado exactamente las mismas para este dataset.

Resumen de la Red Neuronal:

```
Layer (type)                 Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 batch_normalization (BatchN  (None, 32, 32, 32)       128
 ormalization)

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 batch_normalization_1 (Batc  (None, 32, 32, 32)       128
 hNormalization)

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 dropout (Dropout)           (None, 16, 16, 32)        0

 conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_2 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

 batch_normalization_3 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 8, 8, 64)          0

 conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

 batch_normalization_4 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

 batch_normalization_5 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 dropout_2 (Dropout)         (None, 4, 4, 128)         0

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 1024)              2098176

 dropout_3 (Dropout)         (None, 1024)              0

 dense_1 (Dense)             (None, 512)               524800

 batch_normalization_6 (Batc  (None, 512)              2048
 hNormalization)

 dropout_4 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 100)               51300

=================================================================
Total params: 2,965,124
Trainable params: 2,963,204
Non-trainable params: 1,920
_________________________________________________________________
```

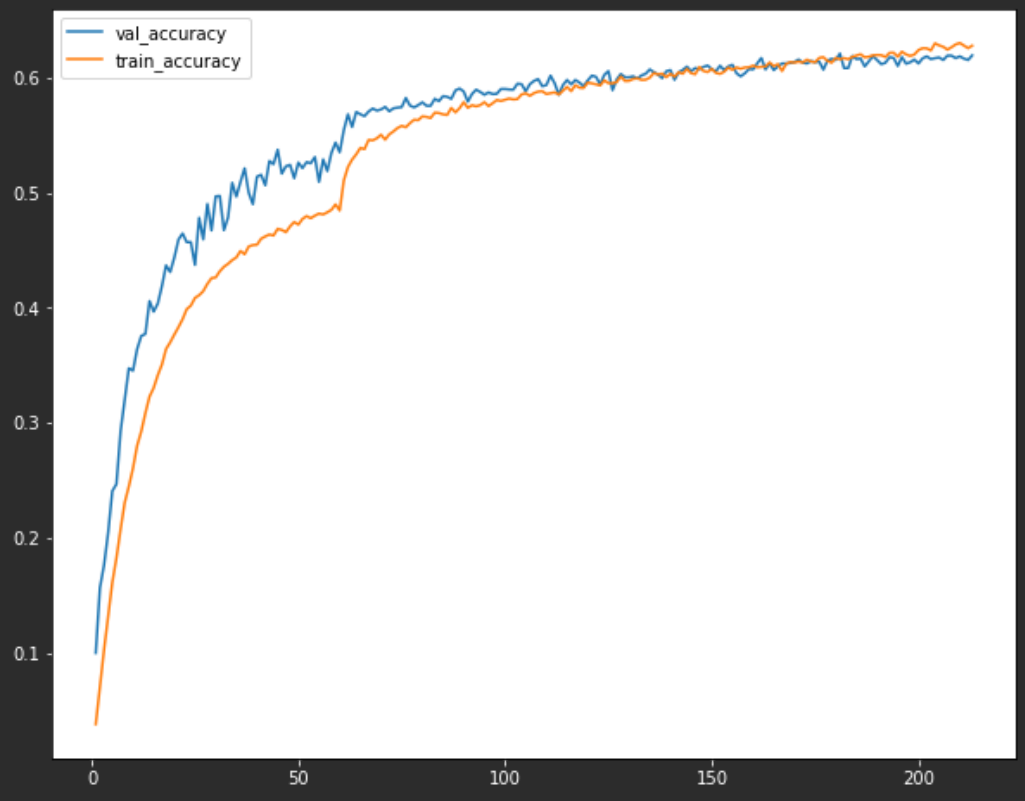
2.3 Métricas y rendimiento del modelo

Este modelo alcanza en los datos de test una tasa de aciertos del **62%**, un loss de **1.7465** y un AUC de **0.9598**.

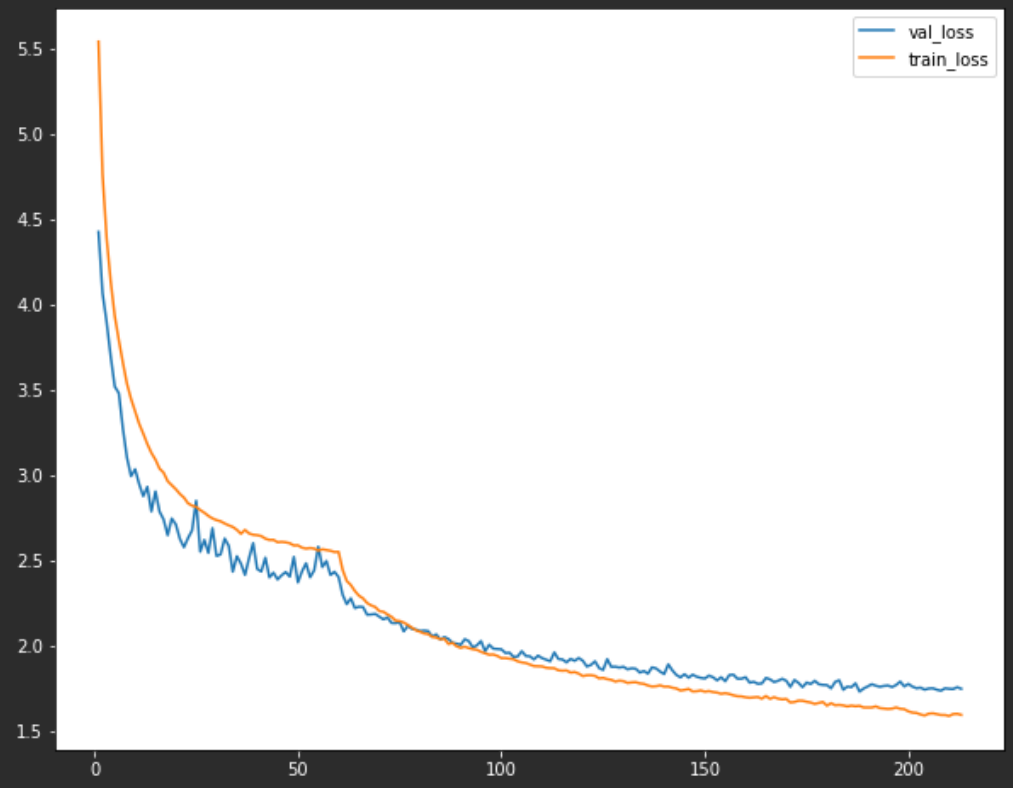
Ha sido entrenado durante **250 épocas,** pero gracias al *EarlyStopping*, el entrenamiento paró a las 215 épocas, tardando en total unas **11-12 horas**.

A continuación, las gráficas que representan el rendimiento del modelo conforme se iba entrenando durante las 100 épocas.

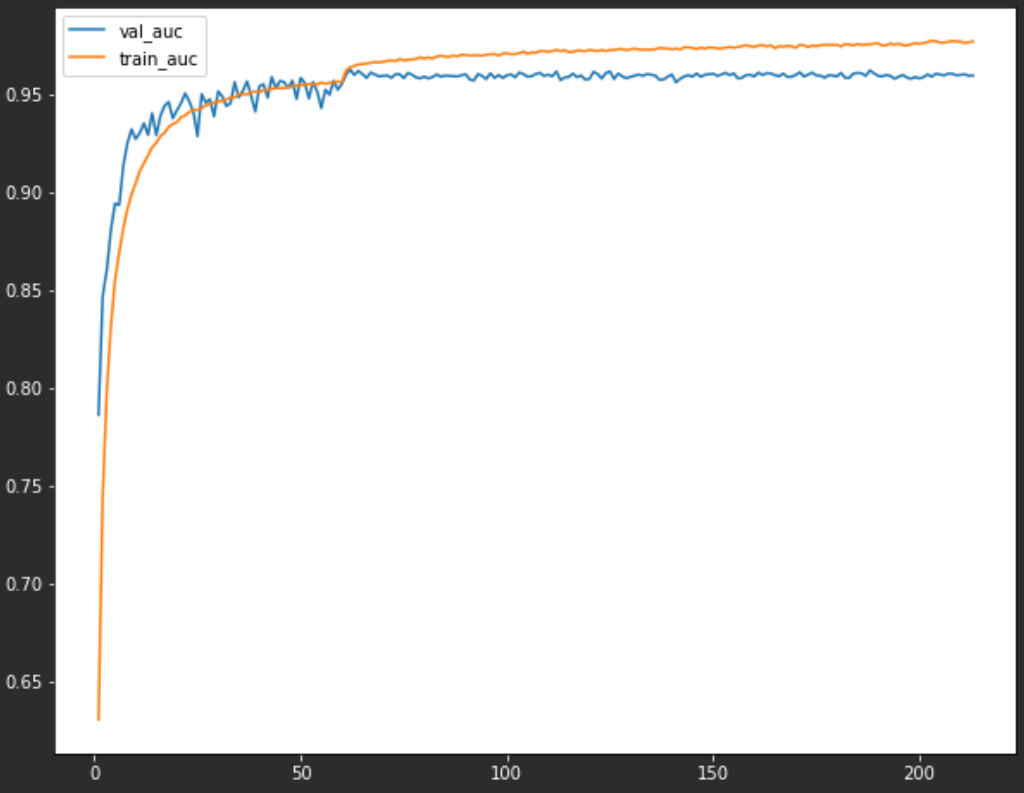
Gráfica del porcentaje de aciertos en entrenamiento frente al de validación



Gráfica de la función loss en entrenamiento frente a la de validación



Gráfica de AUC en entrenamiento frente al de validación



2.4 Prueba del modelo

Para probar el modelo, debemos seguir los mismos pasos del ejemplo anterior. Solo hay que cambiar el endpoint al siguiente:

```
@application.route('/cifar100', methods=['POST'])
```

Le pasamos un objeto JSON con el nombre del fichero de la imagen a probar.

```
{
    "key": "camel.jpg"
}
```

El primer ejemplo lo haremos con esta foto sacada de internet de un camello.

Si analizamos el resultado, podemos ver como nuestro modelo está un 88% seguro de que esta imagen pertenece a la clase *camel*.

```
Predicción 1:  camel (88.5898 %)
Predicción 2:  otter (2.0892 %)
Predicción 3:  snail (1.5274 %)
```

```
{
    "key": "cloud.jpg"
}
```

Para nuestra segunda prueba, vamos a probar con esta imagen correspondiente a una nube.

En este caso, nuestro modelo sólo está seguro un 29% de que esto es una nube.

```
Predicción 1:  cloud (29.7515 %)
Predicción 2:  mountain (23.6181 %)
Predicción 3:  keyboard (19.81 %)
```