DEEP LEARNING AND AI

Proyecto IV – Visión por computador

Alumna: **Marina Ramiro Pareta**

Resumen Proyectos:

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- Proyecto 5: 85.94 (0.426) – Data Augmentation
- Proyecto 6: 81.80 (0.559) – VGG16 (weights=None)
- Proyecto 7: 60.94 (1.118) – TransferLearning-VGG16 (weights='imagenet', Frozen)
- Proyecto 8: 86.88 (0.411) – TransferLearning-VGG16 (weights='imagenet', Trainable)
- Proyecto 9: 87.800 (0.388) – TransferLearning-miniVGG16 (weights='imagenet', Trainable)
- **Proyecto 10: 91.66 (0.279) – CIFAR-10 (64x64) + TransferLearning-VGG16 (weights='imagenet', Trainable)**

0. Modelo base: *230630_base_cnn-cifar10.ipynb*

```
model = ks.Sequential()

# Primera capa convolucional
model.add(ks.layers.Conv2D(32, (3, 3), strides=1, activation='relu',
                           padding='same', input_shape=(32,32,3)))
model.add(ks.layers.MaxPooling2D((2, 2)))

# Capa de aplanamiento
model.add(ks.layers.Flatten())

# Capa completamente conectada
model.add(ks.layers.Dense(32, activation='relu'))

# Capa de salida
model.add(ks.layers.Dense(10, activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

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

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 32)                262176

 dense_1 (Dense)             (None, 10)                330

=================================================================
Total params: 263,402
Trainable params: 263,402
Non-trainable params: 0
```

0.1. Añadimos el **callback de EarlyStopping** al modelo base. Si no se produce una mejora de la métrica de pérdida o en el accuracy en el conjunto de validación durante 5 epochs consecutivos, el entrenamiento se detendrá y se restaurarán los pesos del modelo con los mejores resultados.

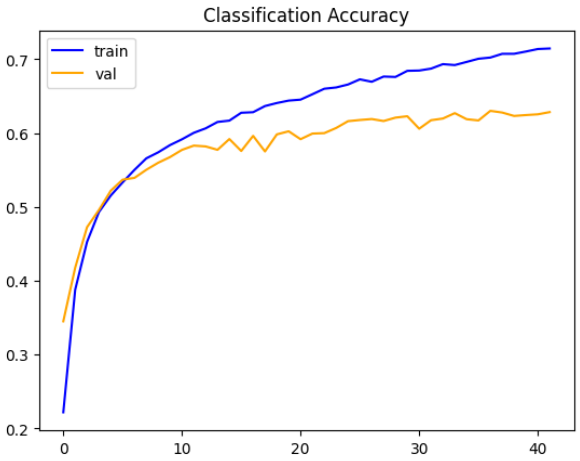
```
# Definir el callback de EarlyStopping
early_stopping_val_loss = EarlyStopping(monitor = 'val_loss',
                                        patience = 5,
                                        restore_best_weights = True)
early_stopping_val_accuracy = EarlyStopping(monitor = "val_accuracy",
                                            patience = 5,
                                            restore_best_weights = True)
```

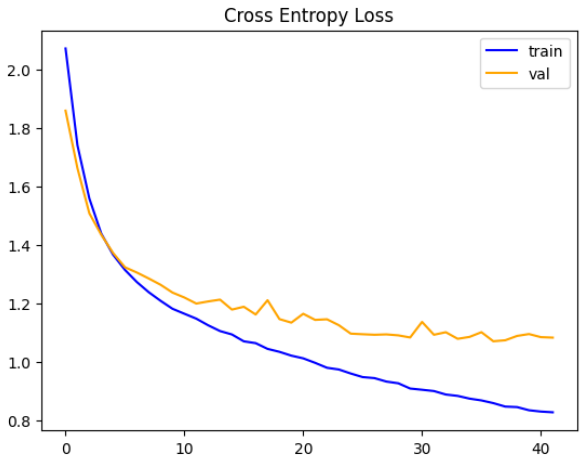
0.2. Cambiamos los **epochs a 100**.

```
# Define the hyperparameters
BATCH_SIZE = 512
EPOCHS = 100
```

```
history = model.fit(x_train_scaled,
                    y_train,
                    epochs = EPOCHS,
                    batch_size = BATCH_SIZE,
                    validation_data = (x_val_scaled, y_val),
                    callbacks = [early_stopping_val_loss, early_stopping_val_accuracy])
```

```
Tiempo de entrenamiento: 0:00:47.380263
```






```
Accuracy (train): 71.342
Accuracy (test): 62.560
```

```
Cross Entropy Loss (train): 0.840
Cross Entropy Loss (test): 1.070
```

- **Proyecto base: 62.56 (1.070)**

1. Proyecto 1: *230630_1_cnn-cifar10.ipynb*

1.1. Como primera aproximación vamos a **aumentar la profundidad de la red** agregando más capas de convolución, de pooling, y de neuronas densas en las capas completamente conectadas. Al aumentar el número de neuronas, el modelo tiene más capacidad para capturar patrones y relaciones complejas en los datos.

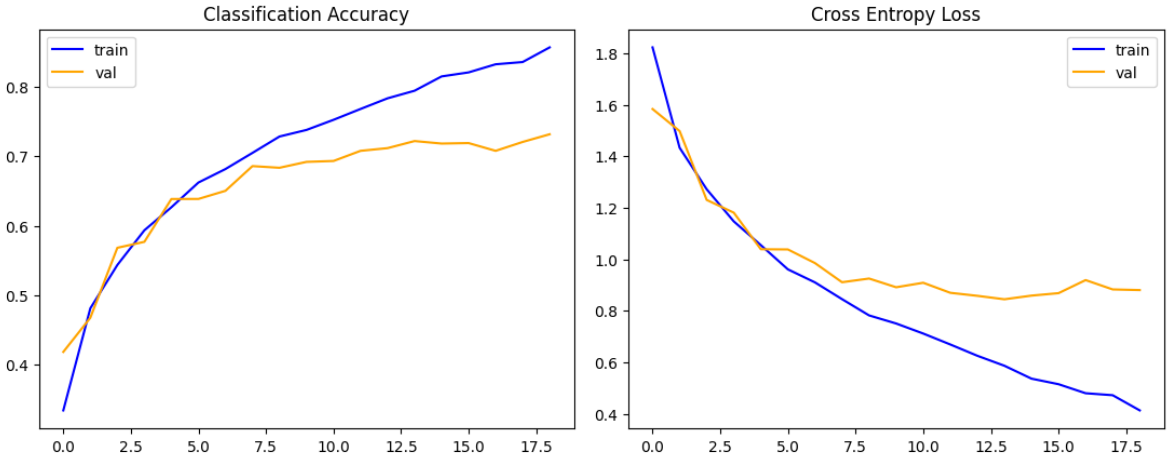
```
model = ks.Sequential()

# Primera capa convolucional
model.add(Conv2D(32, (3, 3), strides=1, activation='relu',
                             padding='same', input_shape=(32,32,3)))
model.add(MaxPooling2D((2, 2)))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(MaxPooling2D((2, 2)))

# Tercera capa convolucional
model.add(Conv2D(128, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(MaxPooling2D((2, 2)))

# Capa de aplanamiento
model.add(Flatten())

# Capa completamente conectada
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))

# Capa de salida
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

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

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496

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

 conv2d_2 (Conv2D)           (None, 8, 8, 128)         73856

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

 flatten (Flatten)           (None, 2048)              0
```

```
dense (Dense)                (None, 256)               524544

dense_1 (Dense)              (None, 128)               32896

dense_2 (Dense)              (None, 10)                1290

=================================================================
Total params: 651,978
Trainable params: 651,978
Non-trainable params: 0
```

```
Tiempo de entrenamiento: 0:00:45.206875
```




```
Accuracy (train): 87.643
Accuracy (test): 72.300
```

```
Cross Entropy Loss (train): 0.360
Cross Entropy Loss (test): 0.902
```

- Hemos mejorado ambas métricas del modelo, pero observamos un sobreajuste a los datos de entrenamiento.

- Proyecto base: 62.56 (1.070)
- **Proyecto 1: 72.30 (0.902)**

2. Proyecto 2: *230630_2_cnn-cifar10_dropout.ipynb*

2.1. Vamos a **añadir capas de Dropout**, una técnica de regularización que ayuda a prevenir ese overfitting al apagar aleatoriamente un porcentaje de las neuronas durante el entrenamiento. Esto evita que el modelo se vuelva demasiado dependiente de ciertas características y promueve la generalización.

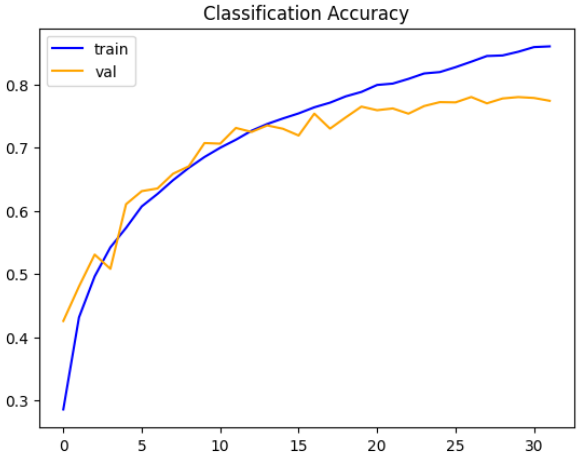
```
model = ks.Sequential()

# Primera capa convolucional
model.add(Conv2D(32, (3, 3), strides=1, activation='relu',
                 padding='same', input_shape=(32,32,3)))
model.add(MaxPooling2D((2, 2)))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), strides=1, activation='relu',
                 padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Tercera capa convolucional
model.add(Conv2D(128, (3, 3), strides=1, activation='relu',
                 padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Capa de aplanamiento
model.add(Flatten())

# Capa completamente conectada
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))

# Capa de salida
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

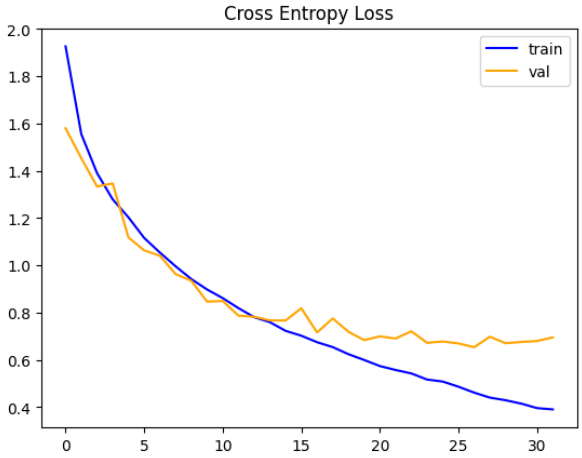
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

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

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496
```

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

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

conv2d_2 (Conv2D)            (None, 8, 8, 128)        73856

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

dropout_1 (Dropout)          (None, 4, 4, 128)        0

flatten (Flatten)            (None, 2048)             0

dense (Dense)                (None, 256)              524544

dense_1 (Dense)              (None, 128)              32896

dropout_2 (Dropout)          (None, 128)              0

dense_2 (Dense)              (None, 10)               1290

=================================================================
Total params: 651,978
Trainable params: 651,978
Non-trainable params: 0
```

```
Tiempo de entrenamiento: 0:01:23.477872
```




```
Accuracy (train): 92.065
Accuracy (test): 77.300
```




```
Cross Entropy Loss (train): 0.268
Cross Entropy Loss (test): 0.673
```

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- **Proyecto 3: 77.30 (0.673) – Dropout**

3. Proyecto 3: *230630_3_cnn-cifar10.ipynb*

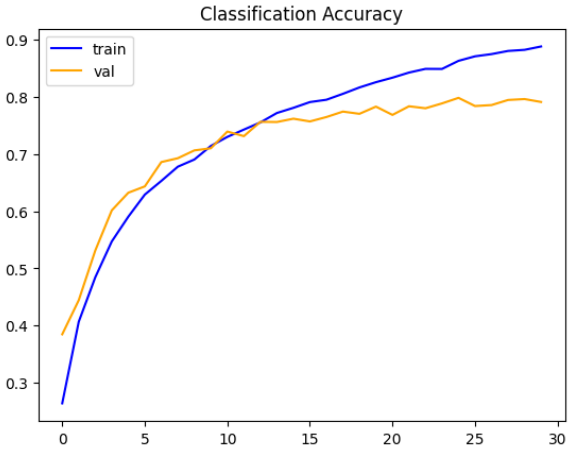
3.1. Vamos a probar de **generar una arquitectura más profunda**, duplicando las capas convolucionales de cada bloque:

```
model = ks.Sequential()

# Primera capa convolucional
model.add(Conv2D(32, (3, 3), strides=1, activation='relu',
                             padding='same', input_shape=(32,32,3)))
model.add(Conv2D(32, (3, 3), strides=1, activation='relu',
                             padding='same', input_shape=(32,32,3)))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(Conv2D(64, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Tercera capa convolucional
model.add(Conv2D(128, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(Conv2D(128, (3, 3), strides=1, activation='relu',
                             padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Capa de aplanamiento
model.add(Flatten())

# Capa completamente conectada
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))

# Capa de salida
model.add(ks.layers.Dense(10, activation='softmax'))
```

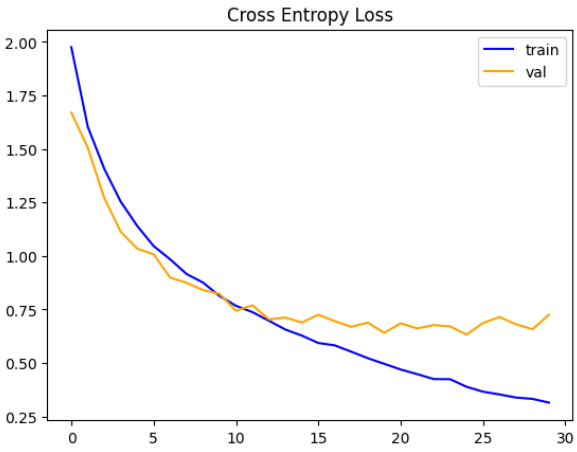
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

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

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

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

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

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

  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

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

  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: 1,341,226
 Trainable params: 1,341,226
 Non-trainable params: 0
 _________________________________________________________________
```

```
Tiempo de entrenamiento: 0:02:16.842777
```




```
Accuracy (train): 94.010
Accuracy (test): 78.410
```




```
Cross Entropy Loss (train): 0.196
Cross Entropy Loss (test): 0.672
```

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- **Proyecto 3: 78.41 (0.672)**

4. Proyecto 4: *230630_4_cnn-cifar10_BatchNorm.ipynb*

4.1. Vamos a probar de agregar **capas de Batch Normalization después de cada capa convolucional**. La normalización de lotes ayuda a normalizar la activación de las capas anteriores, lo que puede mejorar la rapidez con la que el modelo alcanza un estado óptimo o cerca del óptimo durante el proceso de entrenamiento.

```
model = ks.Sequential()

# Primera capa convolucional
model.add(Conv2D(32, (3, 3), input_shape=(32,32,3),activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Tercera capa convolucional
model.add(Conv2D(128, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Capa de aplanamiento
model.add(Flatten())

# Capa completamente conectada
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

# Capa de salida
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

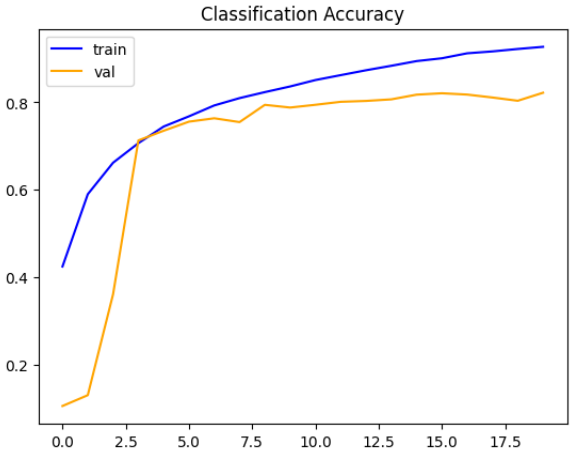
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 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, 256)               524544

 batch_normalization_6 (Batc  (None, 256)              1024
 hNormalization)

 dense_1 (Dense)             (None, 128)               32896

 batch_normalization_7 (Batc  (None, 128)              512
 hNormalization)

 dropout_3 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290
```

```
=================================================================
Total params: 849,066
Trainable params: 847,402
Non-trainable params: 1,664
_________________________________________________________________
```

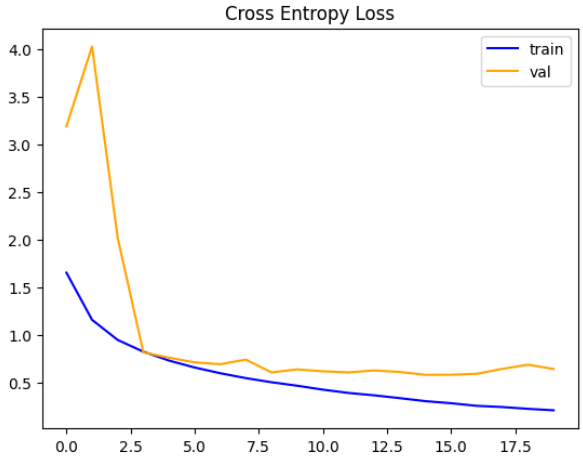
4.2. Además, también **vamos a disminuir el tamaño de batch (**516 → **256) durante el entrenamiento**. Al entrenar con un tamaño de lote más pequeño, se introduce más variabilidad en los datos de cada iteración. Esto puede ayudar al modelo a generalizar mejor y evitar el sobreajuste, ya que se expone a diferentes ejemplos en cada paso de actualización de los pesos.

```
# Define the hyperparameters
BATCH_SIZE = 256
EPOCHS = 100
```

```
Tiempo de entrenamiento: 0:02:28.758790
```




```
Accuracy (train): 97.728
Accuracy (test): 81.730
```




```
Cross Entropy Loss (train): 0.074
Cross Entropy Loss (test): 0.660
```

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- **Proyecto 4: 81.73 (0.660) – Batch Normalization**

5. Proyecto 5: *230701_5_cnn-cifar10_DataAugmentation.ipynb*

5.1. Mantenemos la arquitectura preestablecida de nuestra red neuronal:

```
model = ks.Sequential()

# Primera capa convolucional
model.add(Conv2D(32, (3, 3), input_shape=(32,32,3),activation='relu',
                              padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Tercera capa convolucional
model.add(Conv2D(128, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu',
                             padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

# Capa de aplanamiento
model.add(Flatten())

# Capa completamente conectada
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

# Capa de salida
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 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, 256)               524544

batch_normalization_6 (Batc  (None, 256)               1024
hNormalization)

dense_1 (Dense)              (None, 128)               32896

batch_normalization_7 (Batc  (None, 128)               512
hNormalization)

dropout_3 (Dropout)          (None, 128)               0

dense_2 (Dense)              (None, 10)                1290
```

```
=================================================================
Total params: 849,066
Trainable params: 847,402
Non-trainable params: 1,664
_________________________________________________________________
```

5.2. Vamos a **augmentar la paciencia de nuestro sistema de EarlyStopping (**5→**10)**, para dar más margen de mejora al entrenamiento:

```
# Definir el callback de EarlyStopping
early_stopping_val_loss = EarlyStopping(monitor = 'val_loss',
                                        patience = 10,
                                        restore_best_weights = True)
early_stopping_val_accuracy = EarlyStopping(monitor = "val_accuracy",
                                            patience = 10,
                                            restore_best_weights = True)
```

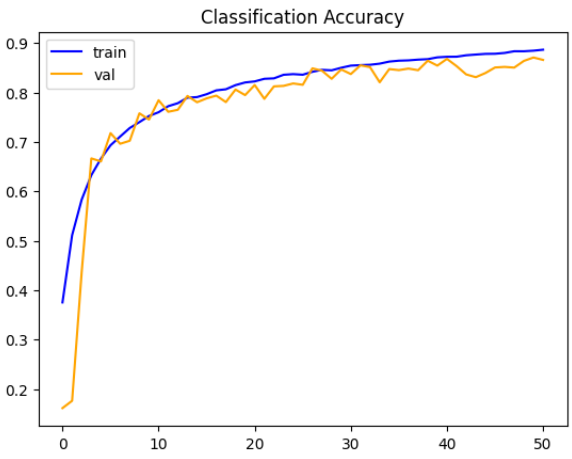
5.3. Vamos a probar la técnica del **Data Augmentation**, para ver si mejoramos el resultado de nuestro modelo. A partir de ahora vamos a usar esta técnica para todos los modelos que siguen.

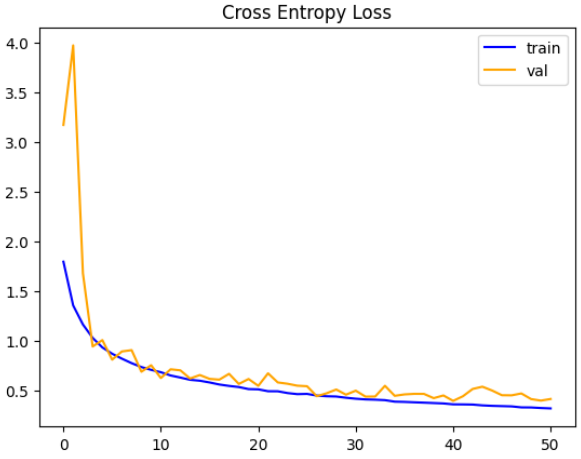
```
# Definir el generador de aumento de datos para el conjunto de entrenamiento
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest')

# Ajustar el generador de aumento de datos a los datos de entrenamiento
train_datagen.fit(x_train)
```

```
# Ajustar el generador de aumento de datos a los datos de validación
validation_datagen = ImageDataGenerator(rescale=1./255)
```

```
# Entrenamiento con Data Augmentation:
history = model.fit(train_datagen.flow(x_train,
                                       y_train,
                                       batch_size = BATCH_SIZE),
                    epochs = EPOCHS,
                    callbacks = [early_stopping_val_loss, early_stopping_val_accuracy],
                    steps_per_epoch = math.ceil(len(x_train)/BATCH_SIZE),
                    validation_data = validation_datagen.flow(x_val,
                                                              y_val,
                                                              batch_size = BATCH_SIZE),
                    validation_steps = math.ceil(len(x_val)/BATCH_SIZE),
                    shuffle=True)
```

```
Tiempo de entrenamiento: 0:20:25.159844
```






```
Accuracy (train): 92.820
Accuracy (test): 85.940
```

```
Cross Entropy Loss (train): 0.206
Cross Entropy Loss (test): 0.426
```

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- **Proyecto 5: 85.94 (0.426) – Data Augmentation**

6. Proyecto 6: *230703_6.1_cnn-cifar10_vgg16_weights=None.ipynb*

6.1. En este experimento aplicaremos la estrategia de **Transfer Learning utilizando la arquitectura de VGG16 sin cargar los pesos del modelo (weights = None)**, y añadiendo la capa de clasificación que previamente hemos diseñado:

```
# Creemos una red que será extracción de features basada en VGG16
model_vgg16 = vgg16.VGG16(include_top = False,
                          weights = None,
                          input_shape = (32,32,3))
```

```
# Añadir un Flatten en la última capa
output = model_vgg16.layers[-1].output
new_output_layer = ks.layers.Flatten()(output)
model_pre_vgg16 = Model(model_vgg16.input, new_output_layer)
model_pre_vgg16.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 block1_conv1 (Conv2D)       (None, 32, 32, 64)        1792

 block1_conv2 (Conv2D)       (None, 32, 32, 64)        36928

 block1_pool (MaxPooling2D)  (None, 16, 16, 64)        0

 block2_conv1 (Conv2D)       (None, 16, 16, 128)       73856

 block2_conv2 (Conv2D)       (None, 16, 16, 128)       147584

 block2_pool (MaxPooling2D)  (None, 8, 8, 128)         0

 block3_conv1 (Conv2D)       (None, 8, 8, 256)         295168

 block3_conv2 (Conv2D)       (None, 8, 8, 256)         590080

 block3_conv3 (Conv2D)       (None, 8, 8, 256)         590080

 block3_pool (MaxPooling2D)  (None, 4, 4, 256)         0

 block4_conv1 (Conv2D)       (None, 4, 4, 512)         1180160

 block4_conv2 (Conv2D)       (None, 4, 4, 512)         2359808

 block4_conv3 (Conv2D)       (None, 4, 4, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 2, 2, 512)         0

 block5_conv1 (Conv2D)       (None, 2, 2, 512)         2359808

 block5_conv2 (Conv2D)       (None, 2, 2, 512)         2359808
```

```
 block5_conv3 (Conv2D)       (None, 2, 2, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 1, 1, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

```
model_post_vgg16 = ks.Sequential()

model_post_vgg16.add(model_pre_vgg16)

# Capa completamente conectada
model_post_vgg16.add(Dense(256, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dense(128, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dropout(0.25))

# Capa de salida
model_post_vgg16.add(Dense(10, activation='softmax'))
```

```
model_post_vgg16.summary()
```

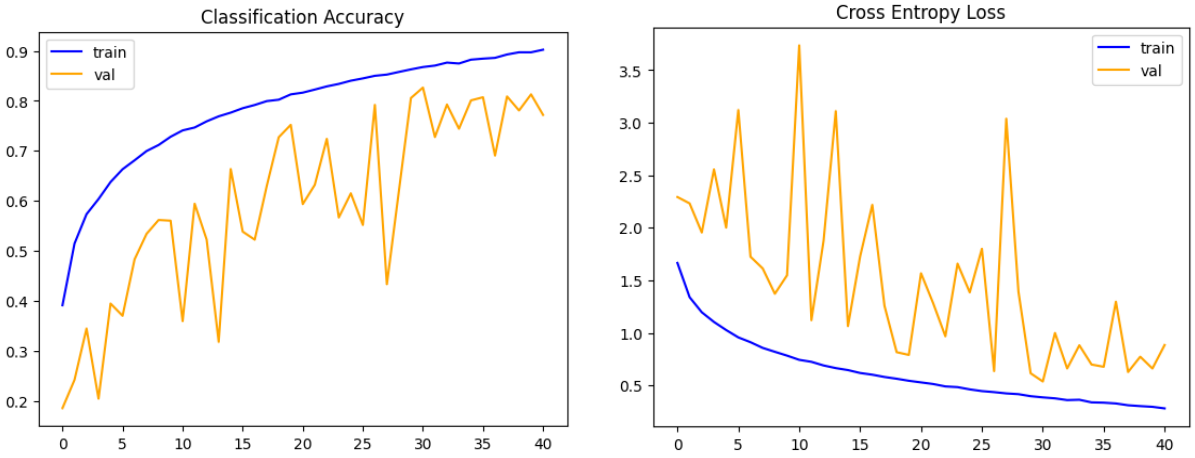
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 model_2 (Functional)        (None, 512)               14714688

 dense_3 (Dense)             (None, 256)               131328

 batch_normalization_2 (Batc  (None, 256)              1024
 hNormalization)

 dense_4 (Dense)             (None, 128)               32896

 batch_normalization_3 (Batc  (None, 128)              512
 hNormalization)

 dropout_1 (Dropout)         (None, 128)               0

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 14,881,738
Trainable params: 14,880,970
Non-trainable params: 768
_________________________________________________________________
```

6.2. Vamos a definir una tasa de aprendizaje (*learning rate*) más baja para el optimizador Adam **(**0.001→**1e-4)**, que puede hacer que modelos más complejos, como p.e.: VGG16, converjan de manera más estable pero más lenta. También vamos a limitar el gradiente (*clipnorm*) para evitar que las gradientes del modelo sean demasiado grandes y prevenir

problemas de explosión del gradiente durante el entrenamiento. A partir de ahora vamos a usar esta técnica para todos los modelos que siguen.

```
# Crear una instancia del optimizador Adam
optimizer = Adam(learning_rate=1e-4,
                 clipnorm=1.0)

# Compilación del modelo
model_post_vgg16.compile(optimizer=optimizer,
                         loss='sparse_categorical_crossentropy',
                         metrics=['accuracy'])
```

```
Tiempo de entrenamiento: 0:15:48.826757
```




```
Accuracy (train): 88.450
Accuracy (test): 81.800
```

```
Cross Entropy Loss (train): 0.328
Cross Entropy Loss (test): 0.559
```

- Aunque la arquitectura VGG16 sea mucho más compleja, observamos que para la clasificación de CIFAR-10 no da tan buenos resultados como arquitecturas más simples.
- Observamos muchas fluctuaciones en la precisión y pérdida durante el entrenamiento. Estas fluctuaciones pueden deberse a varios factores, p.e.: arquitecturas subóptimas, una tasa de aprendizaje inadecuada, problemas de regularización insuficientes o hiperparámetros mal ajustados.

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- **Proyecto 5: 85.94 (0.426) – Data Augmentation**
- Proyecto 6: 81.80 (0.559) – VGG16(weights=None)

7. Proyecto 7: *230703_6.3_cnn-cifar10_vgg16 _weights='imagenet'_frozen.ipynb*

7.1. En este experimento, vamos a aplicar la estrategia de **Transfer Learning utilizando la arquitectura de VGG16**. Para ello, **cargaremos los pesos del modelo previamente entrenado con las imágenes de ImageNet utilizando el parámetro "weights='imagenet'".** Esto nos permitirá aprovechar el conocimiento y las características aprendidas por VGG16 en la tarea de clasificación de imágenes generales.

```
# Creemos una red que será extracción de features basada en VGG16
model_vgg16 = vgg16.VGG16(include_top = False,
                          weights = 'imagenet',
                          input_shape = (32,32,3))
```

Misma arquitectura de VGG16 que el proyecto 6.

7.2. Además, **vamos a congelar esos pesos**, lo que significa que no los actualizaremos durante el entrenamiento de nuestro modelo. Esto garantiza que **solo se entrenará la capa de clasificación que hemos diseñado previamente.** Al congelar los pesos de la capa base, evitamos que se pierda la información aprendida y nos enfocamos en ajustar los parámetros de la capa de clasificación para que se adapten a nuestra tarea específica.

```
trainable = False
for layer in model_pre_vgg16.layers:
    layer.trainable = trainable

pd.set_option("display.max_colwidth", True)
layers = [(layer, layer.name, layer.trainable) for layer in model_pre_vgg16.layers]
pd.DataFrame(layers, columns=("Layer","Name","Is Trainable?"))
```

	Layer	Name	Is Trainable?
0	<keras.engine.input_layer.InputLayer object at 0x7fd98911fe20>	input_1	False
1	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd992e69900>	block1_conv1	False
2	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9890fef80>	block1_conv2	False
3	<keras.layers.pooling.max_pooling2d.MaxPooling2D object at 0x7fd9857ed120>	block1_pool	False
4	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9857ee0b0>	block2_conv1	False
5	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9857eeda0>	block2_conv2	False
6	<keras.layers.pooling.max_pooling2d.MaxPooling2D object at 0x7fd9857ef3a0>	block2_pool	False
7	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9857ee620>	block3_conv1	False
8	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9857ef7f0>	block3_conv2	False
9	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9858742e0>	block3_conv3	False
10	<keras.layers.pooling.max_pooling2d.MaxPooling2D object at 0x7fd9858763b0>	block3_pool	False
11	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd985877040>	block4_conv1	False
12	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd985875b70>	block4_conv2	False
13	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd985877f70>	block4_conv3	False
14	<keras.layers.pooling.max_pooling2d.MaxPooling2D object at 0x7fd985876710>	block4_pool	False

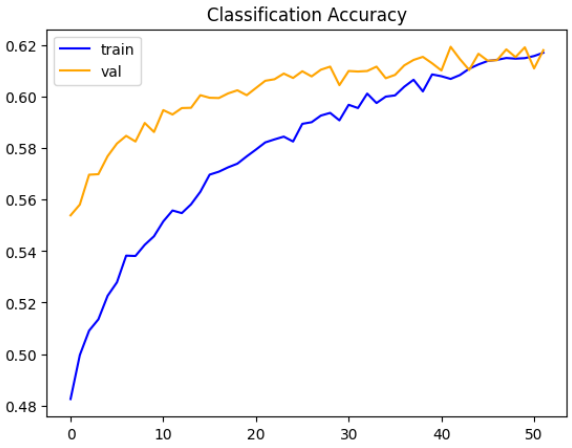
15	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd985877e50>	block5_conv1	False
16	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9280d1d80>	block5_conv2	False
17	<keras.layers.convolutional.conv2d.Conv2D object at 0x7fd9280d2da0>	block5_conv3	False
18	<keras.layers.pooling.max_pooling2d.MaxPooling2D object at 0x7fd9280d3e50>	block5_pool	False
19	<keras.layers.reshaping.flatten.Flatten object at 0x7fd98911ffd0>	flatten	False

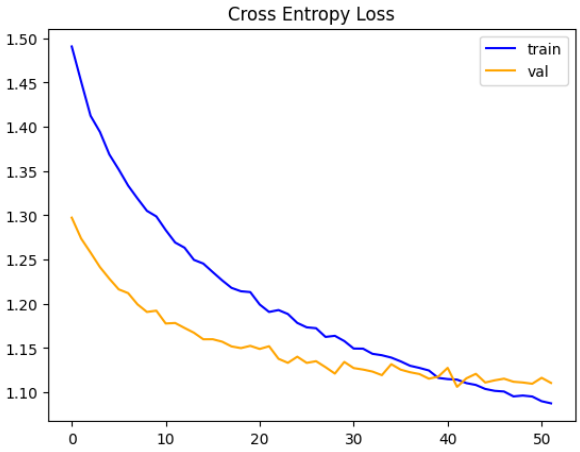
```
model_post_vgg16 = ks.Sequential()

model_post_vgg16.add(model_pre_vgg16)

# Capa completamente conectada
model_post_vgg16.add(Dense(256, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dense(128, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dropout(0.25))

# Capa de salida
model_post_vgg16.add(Dense(10, activation='softmax'))
```

```
model_post_vgg16.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 model_1 (Functional)        (None, 512)               14714688

 dense_3 (Dense)             (None, 256)               131328

 batch_normalization_2 (Batc  (None, 256)              1024
 hNormalization)

 dense_4 (Dense)             (None, 128)               32896

 batch_normalization_3 (Batc  (None, 128)              512
 hNormalization)

 dropout_1 (Dropout)         (None, 128)               0

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 14,881,738
Trainable params: 166,282
Non-trainable params: 14,715,456
_________________________________________________________________
```

```
Tiempo de entrenamiento: 0:22:47.992402
```






```
Accuracy (train): 66.653
Accuracy (test): 60.940
```

```
Cross Entropy Loss (train): 0.950
Cross Entropy Loss (test): 1.118
```

- Esta configuración no es adecuada para clasificar imágenes del conjunto de datos CIFAR-10. Es posible que esto se deba a las diferencias entre las características y las clases de imágenes en CIFAR-10 y las imágenes de ImageNet. La arquitectura VGG16 puede haber aprendido características que no son tan relevantes para la tarea de clasificación de CIFAR-10.

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- **Proyecto 5: 85.94 (0.426) – Data Augmentation**
- Proyecto 6: 81.80 (0.559) – VGG16 (weights=None)
- Proyecto 7: 60.94 (1.118) – TransferLearning-VGG16 (weights='imagenet', Frozen)

8. Proyecto 8: *230703_6.2_cnn-cifar10_vgg16 _weights='imagenet'_trainable.ipynb*

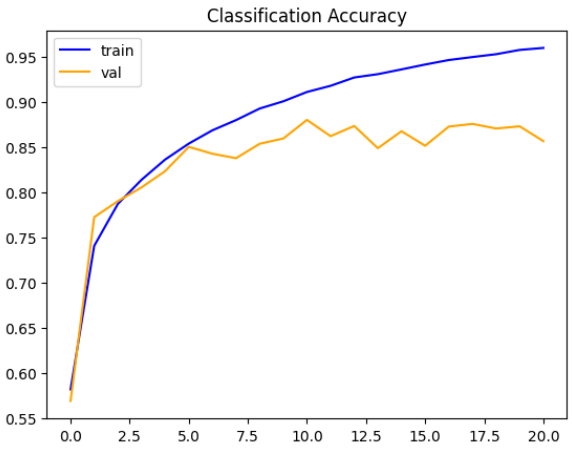
8.1. En este experimento, vamos a aplicar la misma estrategia de **Transfer Learning utilizando la arquitectura de VGG16**. **Cargaremos los pesos del modelo previamente entrenado con las imágenes de ImageNet utilizando el parámetro "weights='imagenet'".** Esto nos permitirá aprovechar el conocimiento y las características aprendidas por VGG16 en la tarea de clasificación de imágenes generales.

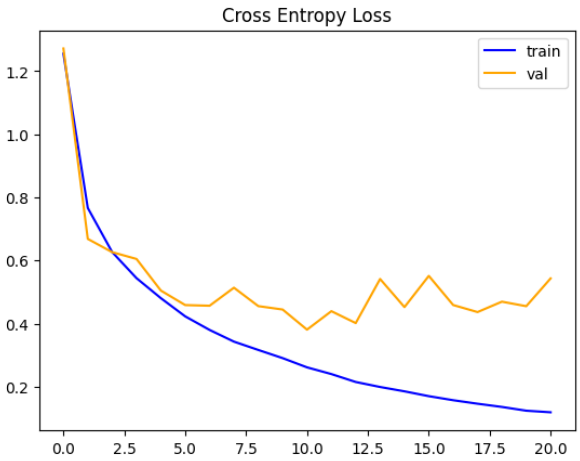
```
# Creemos una red que será extracción de features basada en VGG16
model_vgg16 = vgg16.VGG16(include_top = False,
                          weights = 'imagenet',
                          input_shape = (32,32,3))
```

Misma arquitectura de VGG16 que el proyecto 6 y 7.

8.2. En este caso, se ha decidido **no congelar los pesos de la arquitectura VGG16**. Esto significa que, durante el entrenamiento, los pesos de todas las capas del modelo, incluyendo las capas pre-entrenadas de VGG16, serán actualizados y ajustados en función de los datos específicos de la tarea de clasificación que se está abordando.

```
Tiempo de entrenamiento: 0:09:42.279129
```






```
Accuracy (train): 95.335
Accuracy (test): 86.880
```

```
Cross Entropy Loss (train): 0.140
Cross Entropy Loss (test): 0.411
```

- Observamos como esta estrategia es la más efectiva hasta el momento.
- Sin embargo, obtiene resultados muy similares a los obtenidos con el Proyecto 5, que tenía una arquitectura más simple y no estaba pre-entrenada.

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization

- Proyecto 5: 85.94 (0.426) – Data Augmentation
- Proyecto 6: 81.80 (0.559) – VGG16(weights=None)
- Proyecto 7: 60.94 (1.118) – TransferLearning-VGG16 (weights='imagenet', Frozen)
- **Proyecto 8: 86.88 (0.411) – TransferLearning-VGG16 (weights='imagenet', Trainable)**

9. Proyecto 9: *230705_8.1_cnn-cifar10_mini-vgg16_weights='imagenet'_trainable.ipynb*

9.1. Aplicaremos la estrategia de **Transfer Learning utilizando la arquitectura de VGG16. Cargaremos los pesos** del modelo previamente entrenado **con las imágenes de ImageNet** utilizando el parámetro "weights='imagenet'".

```
# Creemos una red que será extracción de features basada en VGG16
model_vgg16 = vgg16.VGG16(include_top = False,
                          weights = 'imagenet',
                          input_shape = (32,32,3))
```

9.2. Sin embargo, vamos a probar una arquitectura más simple para la clasificación de CIFAR-10. **Vamos a usar una versión reducida de VGG16 (mini-VGG16), usando solo los 3 primeros bloques convolucionales.**

```
# Añadir un Flatten en la última capa
output = model_vgg16.get_layer('block3_pool').output
new_output_layer = ks.layers.Flatten()(output)
model_pre_vgg16 = Model(model_vgg16.input, new_output_layer)
model_pre_vgg16.summary()
```

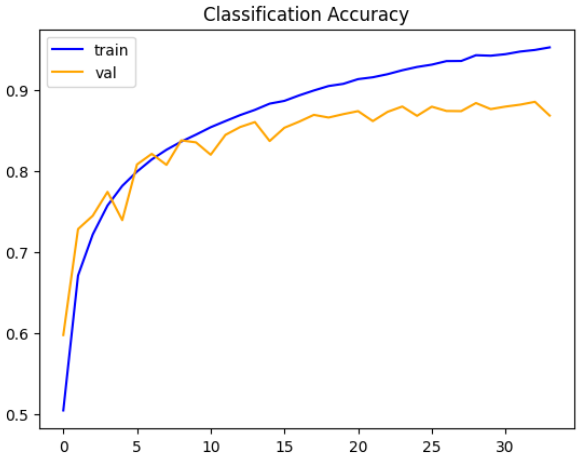
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 block1_conv1 (Conv2D)       (None, 32, 32, 64)        1792

 block1_conv2 (Conv2D)       (None, 32, 32, 64)        36928

 block1_pool (MaxPooling2D)  (None, 16, 16, 64)        0

 block2_conv1 (Conv2D)       (None, 16, 16, 128)       73856

 block2_conv2 (Conv2D)       (None, 16, 16, 128)       147584

 block2_pool (MaxPooling2D)  (None, 8, 8, 128)         0

 block3_conv1 (Conv2D)       (None, 8, 8, 256)         295168

 block3_conv2 (Conv2D)       (None, 8, 8, 256)         590080

 block3_conv3 (Conv2D)       (None, 8, 8, 256)         590080

 block3_pool (MaxPooling2D)  (None, 4, 4, 256)         0

 flatten (Flatten)           (None, 4096)              0

=================================================================
Total params: 1,735,488
Trainable params: 1,735,488
Non-trainable params: 0
_________________________________________________________________
```

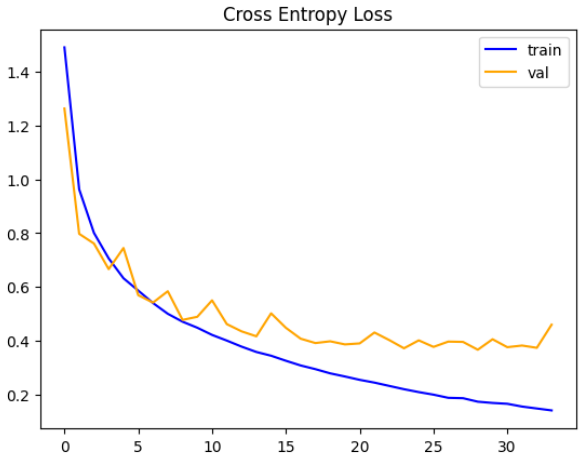
```
model_post_vgg16 = ks.Sequential()

model_post_vgg16.add(model_pre_vgg16)

# Capa completamente conectada
model_post_vgg16.add(Dense(256, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dense(128, activation='relu'))
model_post_vgg16.add(BatchNormalization())
model_post_vgg16.add(Dropout(0.25))

# Capa de salida
model_post_vgg16.add(Dense(10, activation='softmax'))
```

```
model_post_vgg16.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 model (Functional)          (None, 4096)              1735488

 dense (Dense)               (None, 256)               1048832

 batch_normalization (BatchN  (None, 256)              1024
 ormalization)

 dense_1 (Dense)             (None, 128)               32896

 batch_normalization_1 (Batc  (None, 128)              512
 hNormalization)

 dropout (Dropout)           (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

=================================================================
Total params: 2,820,042
Trainable params: 2,819,274
Non-trainable params: 768
_________________________________________________________________
```

```
Tiempo de entrenamiento: 0:13:09.168614
```






```
Accuracy (train): 97.303
Accuracy (test): 87.800
```

```
Cross Entropy Loss (train): 0.082
Cross Entropy Loss (test): 0.388
```

- Observamos resultados muy similares a los obtenidos usando toda la arquitectura de VGG16, a pesar de a ver reducido la profundidad del modelo. Seguramente porque no necesitamos una arquitectura tan compleja para la clasificación de CIFAR-10.

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- Proyecto 5: 85.94 (0.426) – Data Augmentation
- Proyecto 6: 81.80 (0.559) – VGG16 (weights=None)
- Proyecto 7: 60.94 (1.118) – TransferLearning-VGG16 (weights='imagenet', Frozen)
- Proyecto 8: 86.88 (0.411) – TransferLearning-VGG16 (weights='imagenet', Trainable)
- **Proyecto 9: 87.800 (0.388) – TransferLearning-miniVGG16 (weights='imagenet', Trainable)**

10. Proyecto 10: *230705_6.5_cnn-cifar10_vgg16 _weights='imagenet'_trainable_64x64.ipynb*

10.1. Para aprovechar el Transfer Learning de VGG16, cuyo entrenamiento se realizó con imágenes más grandes de ImageNet (244x244), **vamos a redimensionar el tamaño de las imágenes de CIFAR-10 (32x32) a 64x64 píxeles**. Aunque quisiéramos probar con dimensiones más grandes, no podemos por limitaciones de RAM en Google Colab.

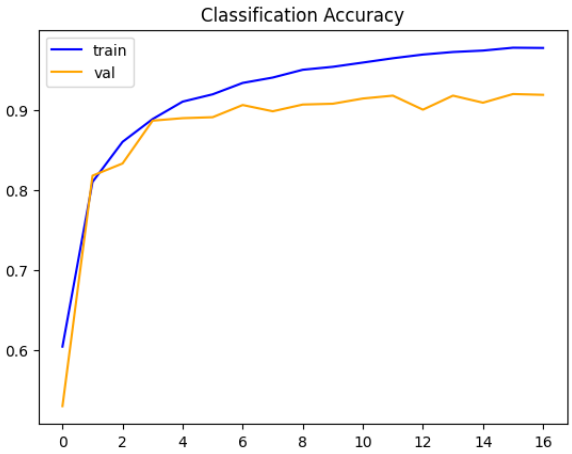
```
def load_cifar10_data(img_rows, img_cols):
    # Cargar los datos de CIFAR-10
    cifar10 = ks.datasets.cifar10
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    # Cambiar el tamaño de las imágenes de CIFAR-10
    if image_data_format() == 'channels_first':
        x_train = np.array([cv2.resize(img.transpose(1, 2, 0),
         (img_rows, img_cols)).transpose(2, 0, 1) for img in x_train])
        x_test = np.array([cv2.resize(img.transpose(1, 2, 0),
         (img_rows, img_cols)).transpose(2, 0, 1) for img in x_test])
    else:
        x_train = np.array([cv2.resize(img,
         (img_rows, img_cols)) for img in x_train])
        x_test = np.array([cv2.resize(img,
         (img_rows, img_cols)) for img in x_test])

    # Convertir los datos en arrays de una dimension (vectores)
    y_train = y_train.ravel()
    y_test = y_test.ravel()

    return x_train, y_train, x_test, y_test
```

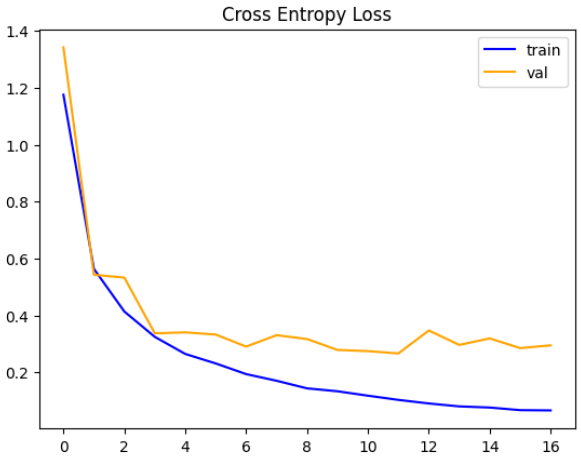
```
x_train, y_train, x_test, y_test = load_cifar10_data(64, 64)
```

10.2. En este experimento, vamos a aplicar la misma estrategia de **Transfer Learning utilizando la arquitectura de VGG16**. **Cargaremos los pesos del modelo previamente entrenado con las imágenes de ImageNet utilizando el parámetro "weights='imagenet'", usando una estructura para imágenes de 64x64.**

```
# Creemos una red que será extracción de features basada en VGG16
model_vgg16 = vgg16.VGG16(include_top = False,
                          weights = 'imagenet',
                          input_shape = (64,64,3))
```

```
Tiempo de entrenamiento: 0:15:48.318542
```






```
Accuracy (train): 98.153
Accuracy (test): 91.660
```

```
Cross Entropy Loss (train): 0.056
Cross Entropy Loss (test): 0.279
```

- A más tamaño de imagen, más se pueden explotar las funcionalidades de la red VGG16.

- Proyecto base: 62.56 (1.070)
- Proyecto 1: 72.30 (0.902)
- Proyecto 2: 77.30 (0.673) – Dropout
- Proyecto 3: 78.41 (0.672)
- Proyecto 4: 81.73 (0.660) – Batch Normalization
- Proyecto 5: 85.94 (0.426) – Data Augmentation
- Proyecto 6: 81.80 (0.559) – VGG16 (weights=None)
- Proyecto 7: 60.94 (1.118) – TransferLearning-VGG16 (weights='imagenet', Frozen)
- Proyecto 8: 86.88 (0.411) – TransferLearning-VGG16 (weights='imagenet', Trainable)
- Proyecto 9: 87.800 (0.388) – TransferLearning-miniVGG16 (weights='imagenet', Trainable)
- **Proyecto 10: 91.66 (0.279) – CIFAR-10 (64x64) + TransferLearning-VGG16 (weights='imagenet', Trainable)**