UNIVERSIDAD
DE GRANADA

Práctica 2.
Redes neuronales convolucionales

Lukas Häring García

November 30, 2019

Tabla de contenidos

1 Primer Apartado

Este primer apartado consiste en crear una red convolucional basada en una tabla ofrecida en la práctica, utilizando *keras*. Para ello, vamos a definir nuestro modelo de la siguiente forma:

```
def BaseNet():
    return Sequential([
    # 5 Kernel = 4 + 1 ( central ) = 2 (left/right, top/bottom)
    # 32 - 2 (left/bottom) - 2 (right/top) = 28
    # Input Image 32 x 32 x 3
    Conv2D(6, kernel_size = (5, 5), input_shape = (32, 32, 3)),

    Activation("relu"),

    # 28 / 2 = 14
    MaxPooling2D(pool_size = (2, 2), strides = (2, 2)),

    # 14 - 2 (left/bottom) - 2 (right/top) = 10
    Conv2D(16, kernel_size = (5, 5)),

    Activation("relu"),

    # 10 / 2 = 5
    MaxPooling2D(pool_size = (2, 2), strides = (2, 2)),

    # Aplanar antes de pasarlo por las capas dense
    Flatten(),

    # Input 400, output 50
    Dense(units = 50, input_shape = (400,)),

    Activation("relu"),

    Dense(units = 25, input_shape = (50,)),

    Activation("softmax")
  ])
```

Un comentario importante que no aparece en la tabla es que la última capa utilizada, será una de activación de tipo **"softmax"**, esto es debido a que la última capa es la encargada de dar resultados que podemos valorar como humanos (por ejemplo, probabilidades). **Softmax** es una función de probabilidad con "N" salidas.

Para **entrenar nuestro modelo**, vamos a utilizar una función creada, llamada *train* (Véase el pié de página *"Función de entrenamiento"*). Además se han creado unos enumerados (Véase el pié de página *"Enumerados"*) para agilizar el uso de este método, que se utilizará para llamar diferentes funciones, por ejemplo, distintos generadores de imágenes; diferente conjunto de datos "**dataset**".

En nuestro caso, vamos a utilizar el conjunto de datos **"CIFAR"**, que se trata de una base de datos con **25 clases** (Elementos distintos, pero podemos tener diferente grados de detalle, por ejemplo, animales o razas).

Además, aunque no se va a especificar porque está dentro de la definición de train, **el generador de imágenes es simple**, definido también por otro enumerado. Esto quiere decir que sólo tendrá un diéz por ciento de datos de entrenamiento para la parte de evaluación (ya que no necesitamos tanta información para validar que nuestro modelo funciona) y no será utilizado en el entrenamineto.

Como es un modelo simple, bastaría con utilizar **30 épocas** (esto es similar a hablar de iteraciones de entrenamiento). Aunque podemos decir que, si entrenamos demasiado nuestro modelo, podemos **sobreajustarlo** *"overfitting"*, esto quiere decir que nuestro modelo ya no mejora.

```
history = train(
  BaseNet(),
  info = TrainInfo.HALF,
  dataset = ILoader.CIFAR,
  epochs = 30,
  #early_stopping = True
)

show_evolution(history)
```

1.1 Resultados

Fig. 1: Este gráfico se corresponde a un **"accuracy"** 0.3896

Como vemos, el modelo presentado es capaz de identificar un 38% de imágenes correctamente, lo que nos dice que no es muy bueno (Con una media $\bar{x} = 0.3906$) de accuracy, realizando 5 veces el mismo entrenamiento con diferentes elementos del dataset utilizado).

Vemos además en la gráfica que hay un sobreajuste aproximadamente en la época 5, esto quiere decir que nuestro modelo ha ido empeorando cada vez más ya que se ha centrado cada vez más en el dataset de entrenamiento proporcionado, por lo que no está generalizando mucho.

2 Segundo Apartado

En este apartado consiste optimizar (conseguir mejor porcentaje de acierto) el modelo anterior, para ello vamos a utilizar diferentes estrategias:

Incrementar el número de capas convolucionales con mayor número de características, empezando con 16 y acabamos con profundidad de 256 (podríamos añadir más). Además el kernel es de 3 x 3, Como se ha especificado en la práctica, se ha añadido BatchNormalization después de cada capa de Convolución.

Además, se han hecho pruebas probando añadir más capas full connected, pero esto no va a conseguir optimizar el modelo ya que lo que se consigue es realizar más combinaciones entre las características, pero en muy pocos casos, vamos a ver una optimización **BaseNetBatchOptimized**.

Se han creado además dos modelos a partir del modelo inicial que intercambian el "**BatchNormalization**" para comprobar cuando es mejor, antes "*BaseNetBatchBefore*" o después "*BaseNetBatchAfter*" de la capa de activación.

```
def BaseNetBatchOptimized():
  return Sequential([
    # 5 Kernel = 4 + 1 ( central ) = 2 (left/right, top/bottom)
    # Input Image 32 x 32 x 3
    # 32 - 3 (left/bottom) - 3 (right/top) = 26
    Conv2D(16, kernel_size = (3, 3), input_shape = (32, 32, 3)),
    Activation("relu"),
    BatchNormalization(),

    # 26 - 2 (left/bottom) - 2 (right/top) = 22
    Conv2D(32, kernel_size = (3, 3)),
    Activation("relu"),
    BatchNormalization(),

    # 26 / 2 = 11
    MaxPooling2D(pool_size = (2, 2), strides = (2, 2)),

    # 11 - 2 (left/bottom) - 1 (right/top) = 9
    Conv2D(64, kernel_size = (3, 3)),
    Activation("relu"),
    BatchNormalization(),

    # 13 - 1 (left/bottom) - 1 (right/top) = 11
    Conv2D(128, kernel_size = (3, 3)),
    Activation("relu"),
```

```
    BatchNormalization(),

    # 11 - 1 (left/bottom) - 1 (right/top) = 9
    Conv2D(256, kernel_size = (3, 3)),
    Activation("relu"),
    BatchNormalization(),

    # 11 / 2 = 5.5
    MaxPooling2D(pool_size = (2, 2), strides = (2, 2)),

    # Aplanar antes de pasarlo por las capas dense
    Flatten(),

    # Input 400, output 50
    Dense(units = 50, input_shape = (400,)),
    Activation("relu"),

    Dense(units = 25, input_shape = (50,)),
    Activation("softmax")
])
```

2.1 Resultados

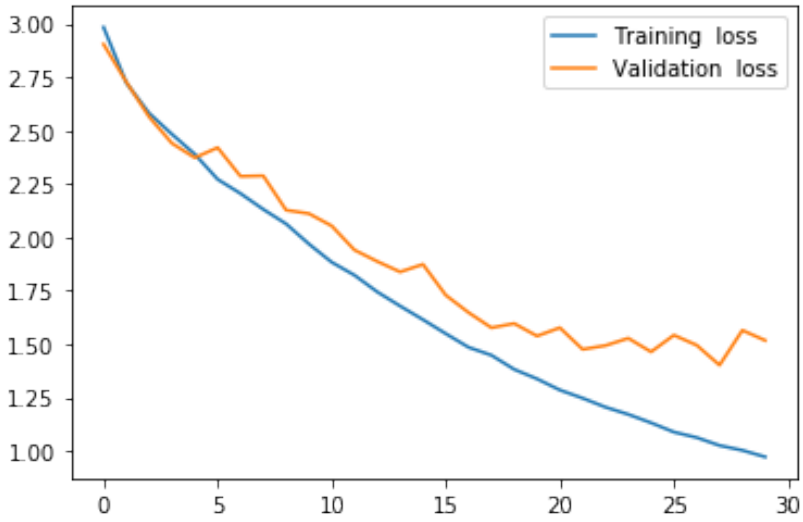

Fig. 2: Modelo optimizado con **"accuracy"** 0.6172

Este modelo ha sido entrenado utilizando un generador de datos que incrementa los elementos volteándolos, escalándolos y normalizándolos, por lo que tendremos mayor cantidad de imágenes con las que entrenar, incrementando así el tiempo de ejecución.

Este modelo acierta un 62% de las veces, casi el doble que nuestro modelo inicial, además vemos que este deja de mejorar aproximadamente a partir de las 25 épocas, esto es una buena indicación ya que aunque tarde más en mejorar por época (ya que su entrenamiento es 10 veces más lento), podemos asegurar un mejor modelo.

Podemos decir que extraer características a partir de incrementar la produndidad de los canales, nos dice que nuestro modelo está aprendiendo más características y cada vez más minucionsas pero importantes.

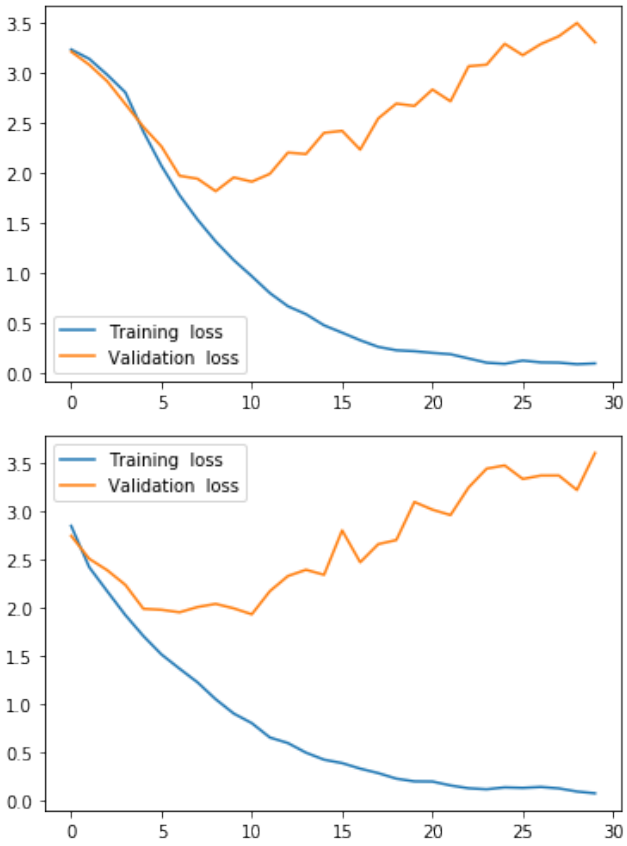

Fig. 3:
Arriba, Modelo "*BaseNetBatchOptimized*" con Simple **"accuracy"** 0.5344.
Abajo, Modelo "*BaseNetBatchOptimized*" con Whitening **"accuracy"** 0.5112

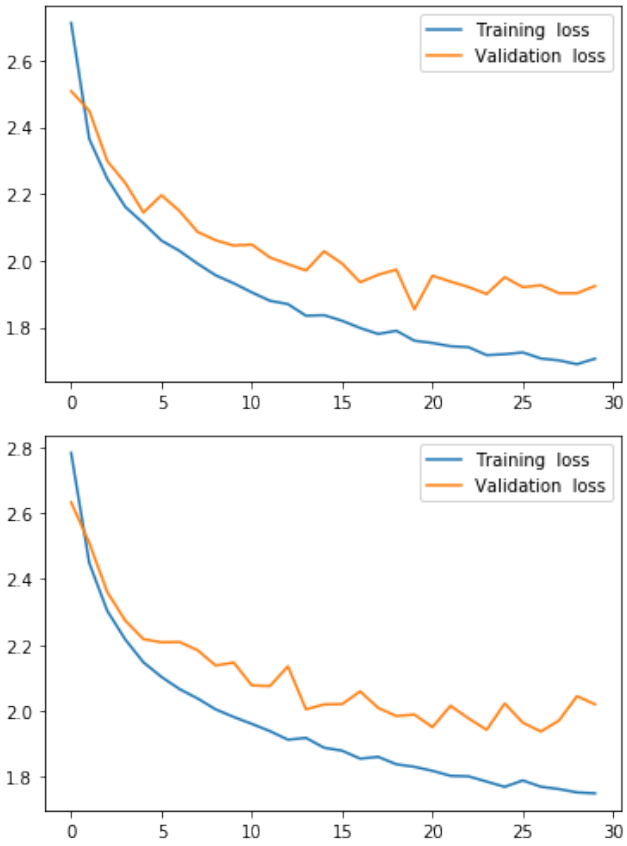

Fig. 4:
Arriba, Modelo "*BaseNetBatchAfter*" con **"accuracy"** 0.4504.
Abajo, Modelo "*BaseNetBatchBefore*" con **"accuracy"** 0.4228

Vemos que no hay una gran diferencia entre utilizar el "*BatchNormalization*" antes ($\bar{x}_n$) o después ($\bar{x}_m$) de la capa de activación, además, se ha realizado la diferencia entre la media de ambas y obtenemos que $|\bar{x}_n - \bar{x}_m| < 0.02$, esto quiere decir que existe un modelo que nos ofrece un 2% más que el otro en este ejemplo (pero no se puede generalizar). Esto es un tema de debate abierto[3].

En mi opinión, tiene más sentido poner la capa de "*BatchNormalization*" antes que la de activación ("*ReLu*") ya que se encarga de normalizar la distribución de características, el cual son todos valores positivos, que si fuera de la otra forma, sesgaría los valores por la aparición de números negativos.

3 Tercer Apartado

3.1 Tercer Apartado

Vamos a utilizar un modelo conocido por "**ResNet50**", pre-entrenado con la base de datos "*imagenet*"[4] (que contiene 1.2 millones de imágenes con 1000 categorías), para extraer características y para utilizarlo como una nueva red neuronal para clasificar nuestras imágenes de nuestra base de datos de pájaros "*CUB-200*" con 200 clases (200 especies distintas de pájaros).

3.2 Extractor de características

Extraer características consiste en ver en qué se fija el modelo pre-entrenado (con distinto dataset) de nuestro dataset y ver que es capaz de predecir. Podemos realizar este método de dos formas, crear un modelo y utilizar la salida para entrenar nuestro nuevo modelo o crear un modelo completo y congelar las capas del modelo que vamos a utilizar (pre-entrenado) y entrenarlo por completo. Para la salida deberemos quitarle la última capa y añadir un transformador (dos full connected) con la salida del número de clases que tiene nuestro "data set".

```
def ResNetExtractor():
  # Create ResNet
  Res50 = ResNet50(
    weights = "imagenet", # Set Weights of imagenet
    include_top = False, # Remove Last Layer
    pooling = "avg", # Add Average Polling last layer
    input_shape = (224, 224, 3) # Input shape and channels
  )

  # Set all layers to no trainable because
  # we only want the characteristics
  for layer in Res50.layers:
    layer.trainable = False

  # Create Net Characteristics Extractor
  return Sequential([
    Res50,

    # Dense
    Dense(500, input_shape = (2048,)),
    Activation("relu"),

    # Dense of 200 characteristics
    Dense(200, input_shape = (500,)),
    Activation("softmax")
  ])
```

3.3 Resultados

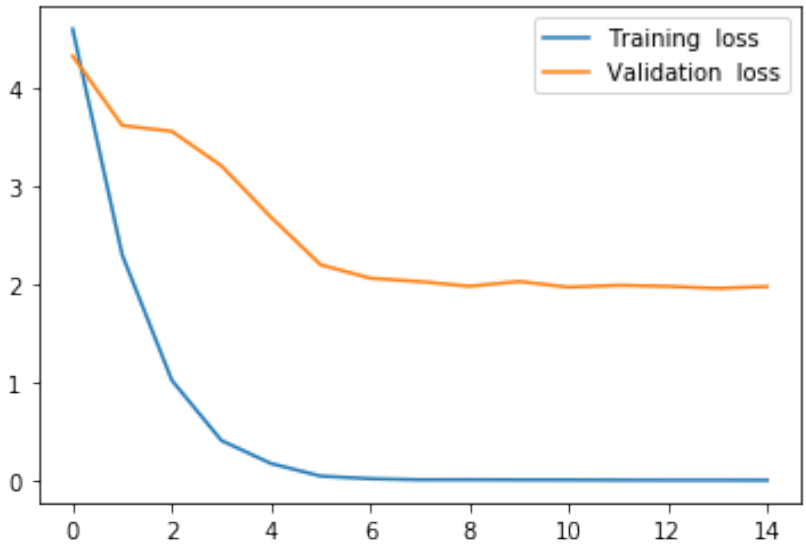

Fig. 5: Feature Extractor con **"accuracy"** 0.3415

Vemos que este modelo es capaz de predecir alrededor del 35% de nuestros datos correctamente, lo que nos indica que no está nada mal para un modelo que ha sido entrenado con imágenes más genéricas sobre un data set de pájaros.

Podemos observar además que nuestro modelo se entrena muy rápido, bastaría con 2 a 6 épocas para tener especializado nuestro extractor de características.

3.4 Fine Tuning

Este proceso consiste en untilizar un modelo pre-entrenado con un número de clases distinto y re-entrenarlo añadiendo algunas capas para utilizarlo en nuestro dataset. Para ello, vamos a utilizar el mismo esquema que el apartado anterior, pero ahora sin congelar las capas del modelo utilizado, es decir, entrenar el modelo conjunto completo.

```
# Create Net Fine Tuning ( Difference: All Layers are trainable)
def ResNetFineTuning():
  return Sequential([
    ResNet50(
      weights = "imagenet", # Set Weights of imagenet
      include_top = False, # Remove Last Layer
      pooling = "avg", # Add Average Polling last layer
      input_shape = (224, 224, 3) # Input shape and channels
    ),

    # Dense
    Dense(500, input_shape = (2048,)),
    Activation("relu"),

    # Dense of 200 characteristics
    Dense(200, input_shape = (500,)),
    Activation("softmax")
  ])
```

3.5 Resultados

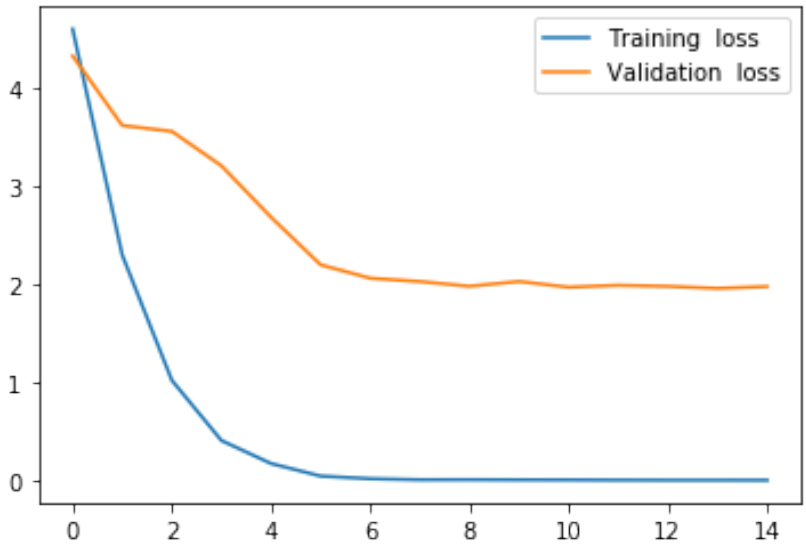

Fig. 6: Fine Tuning **"accuracy"** 0.440

Entrenando el modelo, llegamos a obtener un resultado de 44%, que ha diferencia del apartado anterior, hemos obtenido un incremento del 10%. Aunque parece poco, para un dataset con 200 clases, es mucho, estamos hablando de casi un 50% de precisión.

Además, ha bastado re-entrenar nuestro modelo de 6 a 8 épocas (aproximadamente diéz minutos de entrenamiento) para obtener el resultado anterior.

Aún así, una hipótesis de este resultado puede ser que el modelo al tener los pesos del modelo "ImageNet", puede que muchas neuronas se hayan especializado en clases que no tienen nada que ver con pájaros.

4 Bonus

Para atacar este problema, he estado interesándome en modelos híbridos, modelos que adaptan las cualidades de diferentes modelos, en mi caso, un modelo híbrido de ResNet con Inception[5]. Un modelo presentado por Google, que utiliza capas como **Dropout** que bloquea el entrenamiento de forma aleatoria de neuronas.

Además, utiliza un modelo no lineal de una red muy profunda, **véase la figura 7**. Combina una gran cantidad de capas convolucionales (para extraer características). Pero como ellos indican, necesita el doble de memoria y cómputo.

Antes de optar por esta versión, he probado el modelo **Inception V3** sin la hibridez, los resultados han sido muy malos, incluso peores que el modelo propuesto "**ResNet**"[6], se trata de un paper de identificación de 10 razas de perros. De ahí, mi decisión de probar un modelo más complejo, que sea capaz de diferenciar muchas más clases.

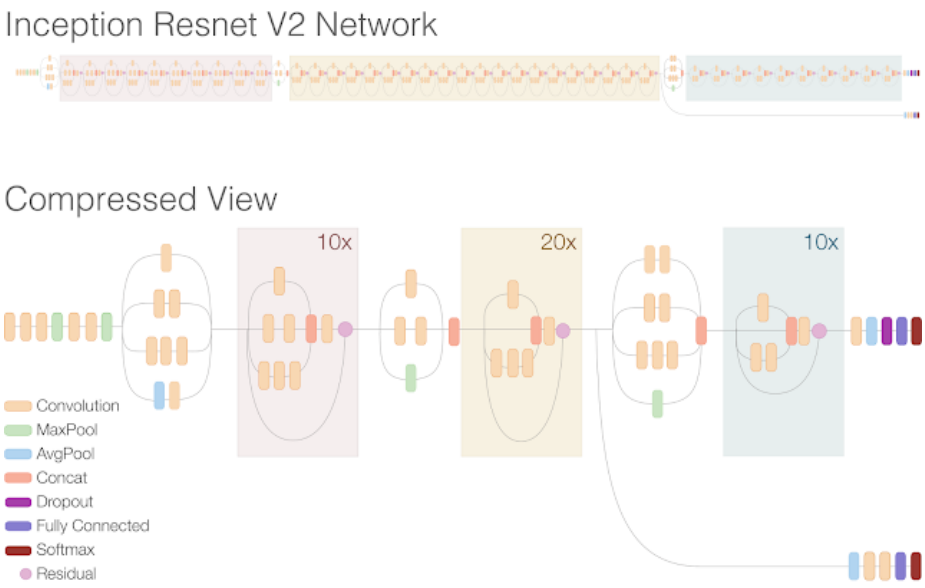

Fig. 7: Diagrama Esquemático de Inception-ResNet-v2

```
# Create
def Inception():
  return Sequential([
    InceptionResNetV2(
      weights = "imagenet", # Set Weights of imagenet
      include_top = False, # Remove Last Layer
      pooling = "avg", # Add Average Polling last layer
      input_shape = (224, 224, 3) # Input shape and channels
    ),
    # Dense
    Dense(1024, input_shape = (2048,)),
    Activation("relu"),
    # Dense of 200 characteristics
    Dense(200, input_shape = (1024,)),
    Activation("softmax")
  ])
```

Recordar que el generador de imágenes pre-procesa las imágenes para que el nuevo modelo sea capaz de entenderlo, para ello utiliza el enumerado "*INCEPTION*".

4.1 Resultados

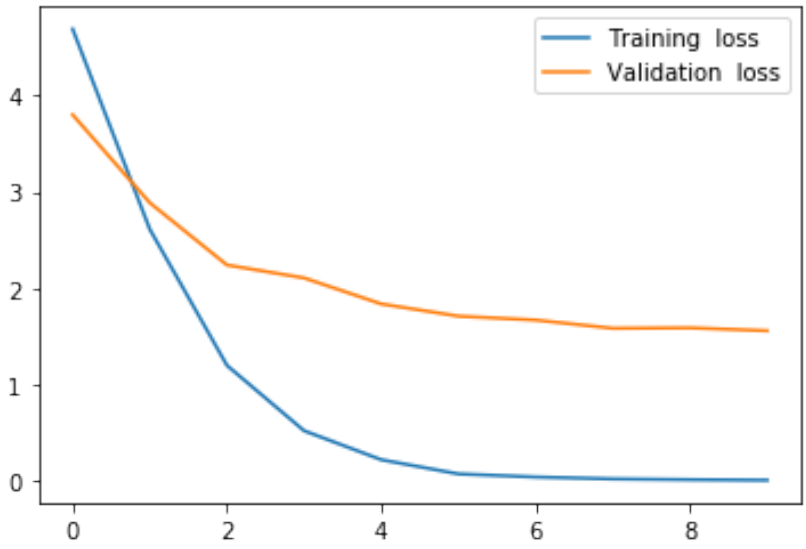

Fig. 8: Bonus Hybrid **"accuracy"** 0.5535

Este modelo ha obtenido un **accuracy del 55%**, mucho mejor que el modelo anterior, aún así, se le podrían haber pasado algunas épocas más y haber conseguido probablemente un mayor porcentaje de acierto, ya que si vemos la gráfica, no observamos en ningún momento **overfitting**.

Este modelo de google aprovecha muchas propiedades que se sabe que funciona, una mayor profundidad en el modelo convolucional y un restablecimiento aleatorio de neuronas (dropout), que en este caso, keras deja de entrenar aleatoriamente un porcentaje de neuronas.

5 Pié de página

5.1 Función de Entrenamiento

```
def train(model, batch_size = 32, epochs = 80,
          dataset = ILoader.CIFAR, info = TrainInfo.FULL,
          compile_type = Compile.SGD,
          igenerator_type = IGenerator.SIMPLE,
          early_stopping = False):
  # Detail of the Information
  verbose = int(info.value)

  # Callbacks
  callbacks = []

  # Early Stopping Callback
  if early_stopping:
    callbacks.append(
      EarlyStopping(
        # Monitorize Val Loss
        monitor  = 'val_loss',
        # Patience (Minimum number of epochs at least trains)
        patience = 10,
        # Information Quantity
        verbose  = verbose,
      )
    )

  # Load Dataset
  x_train, y_train, x_test, y_test = load_dataset(dataset)

  # Debug
  if DEBUG:
    debug(model, compile_type, igenerator_type, x_train, y_train)

  print("--- Training Model ---")
  # Compile Model
  compiler(model, compile_type)

  # Obtain Generator, pass images and standarize (data augmentation)
  generator_train = igenerator(
    igenerator_type,
    split = 0.1
  )

  # Asign to ram a size of x_train
```

```
generator_train.fit(x_train)

# Fits the model on batches
history = model.fit_generator(
  # Training Data Batch
  generator = generator_train.flow(
    x_train,
    y_train,
    batch_size = batch_size,
    subset = "training"
  ),
  # Number of epochs (Times all batches will pass)
  epochs = epochs,
  # Number of batches for the training
  steps_per_epoch = len(x_train) * 0.9 / batch_size,

  # Validation Data Batch
  validation_data = generator_train.flow(
    x_train,
    y_train,
    batch_size = batch_size,
    subset = "validation"
  ),
  # Number of batches for the testing
  validation_steps = len(x_train) * 0.1 / batch_size,
  # Appearance the info will show per bach
  verbose = verbose,
  # Callbacks (Early Stopping, etc.)
  callbacks = callbacks
)

# Evaluate
evaluate(model, x_test, y_test, info, igenerator_type)

# Return Training History
return history
```

5.2 Función de Validación

```
def evaluate(model, x_test, y_test,
             info = TrainInfo.FULL,
             igenerator_type = IGenerator.SIMPLE):
  # Detail of the Information
  verbose = int(info.value)

  print("--- Evaluate Results ---")

  # Test Generator
  generator_test = igenerator(
    igenerator_type
  )

  # Asign to ram  a size of x_test
  generator_test.fit(x_test)

  # Calc Accuracy
  scores = model.evaluate_generator(
    generator = generator_test.flow(
      x_test,
      y_test,
      batch_size = 1,
      shuffle = False
    ),
    # Appearance the info will show
    steps = len(x_test),
    verbose = verbose,
  )

  print("Accuracy: ", str(scores[1]))
```

5.3 Enumerados

5.3.1 Enumerado de cantidad de información

Muestra la cantidad de información que queremos que el entrenamiento muestre,

1. *NONE* no devolverá información,
2. *HALF*, mostrará información parcial,
3. FULL mostrará toda la información posible.

```
class TrainInfo(enum.Enum):
    NONE = 0
    FULL = 1
    HALF = 2
```

5.3.2 Enumerado de generador de datos

Para añadir mayor cantidad de elementos a nuestro conjunto de datos, utilizaremos el generador de keras para: normalizar, escalar, desplazar, etc.

Se han creado enumerados que se encargarán de delegar la función (utilizar un generador) según el enumerado elegido:

1. *SIMPLE* genera un 10% de datos para la evaluación.
2. *COMPLEX* genera un 10% de datos para la evaluación además aleatoriamente añadirá un espejo horizontal / verticalmente de la imagen y un zoom aleatorio desde hasta un 20%. Además, normalizará los datos (*featurewise_center* y *featurewise_std_normalization*).
3. *WHITENING* genera un 10% de datos para la evaluación, normalizará los datos y el "whitening" que se encarga de decorrelar las carácterísticas de las imágenes.
4. *RESNET* genera un 10% de datos para la evaluación y preprocesa utilizando un método del modelo RESNET.
5. *INCEPTION* genera un 10% de datos para la evaluación y preprocesa utilizando un método del modelo INCEPTION-RESNET-2.

```
class IGenerator(enum.Enum):
    SIMPLE    = 0
    COMPLEX   = 1
    WHITENING = 2
    RESNET    = 3
    INCEPTION = 4
```

```
def igenerator(igenerator_type = IGenerator.SIMPLE, split = 0.0):

  generator = ""

  value = int(igenerator_type.value)
  if    value == int(IGenerator.SIMPLE.value)     :
    generator = IGenerator_simple(split)
  elif value == int(IGenerator.COMPLEX.value)    :
    generator = IGenerator_complex(split)
  elif value == int(IGenerator.WHITENING.value) :
    generator = IGenerator_whitening(split)
  elif value == int(IGenerator.RESNET.value)     :
    generator = IGenerator_resnet(split)
  elif value == int(IGenerator.INCEPTION.value) :
    generator = IGenerator_inception(split)

  return generator
```

5.3.3 Enumerado de compilación

Se proponen dos métodos de compilación (búsqueda en espacios n-dimensionales), pero solo se utilizará **SGD**.

1. *ADAM*
2. *SGD*

```
class Compile(enum.Enum):
   SGD = 0
   ADAM = 1

def compiler(model, compile_type = Compile.SGD):
  optimizer = ""

  value = int(compile_type.value)
  if    value == int(Compile.SGD.value) : optimizer = sgd_optimizer()
  elif value == int(Compile.ADAM.value): optimizer = adam_optimizer()

  model.compile(
    optimizer,
    loss = "categorical_crossentropy",
    metrics = ["accuracy"]
  )
  return model
```

Referencias

[1] Softmax
https://es.wikipedia.org/wiki/Funci\%C3\%B3n_SoftMax

[2] Optimizadores
https://keras.io/optimizers/

[3] BatchNormalization after or before?
https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md

[4] ImageNet
https://appliedmachinelearning.blog/2019/07/29/transfer-learning-using-feature-extraction-from-trained-models-food-images-classificatio

[5] Inception Hybrid
https://ai.googleblog.com/2016/08/improving-inception-and-image.html

[6] Dog Breed Classification using Deep Learning
https://towardsdatascience.com/dog-breed-classification-hands-on-approach-b5e4f88c333e