

Prácticas de Visión por Computador

Grupo 2

Presentación de la Práctica 3

Introducción a Keras

Pablo Mesejo

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial



UNIVERSIDAD
DE GRANADA



Índice

- Normas de entrega
- Introducción a Keras
- Presentación de la práctica

Índice

- **Normas de entrega**
- Introducción a Keras
- Presentación de la práctica

Normas de la Entrega de Prácticas

- Uno o varios ficheros Python (podéis usar como plantilla e inspiración: `Ejemplo_mnist.ipynb`).
- El código debe estar comentado.
- Se entrega memoria (PDF) y código (Python) → ZIP/RAR
 - Recordad que también podéis entregar directamente un Notebook con todo integrado (memoria/código)

Normas de la Entrega de Prácticas

- Solo se entrega memoria y código fuente → **no imágenes!**
- Lectura de imágenes o cualquier fichero de entrada:
“imagenes/nombre_fichero”
- **No escribir nada en el disco!**
- La práctica deberá poder ser ejecutada de principio a fin **sin errores** y sin necesidad de **ninguna selección de opciones**.
 - Hay que fijar de inicio los parámetros que se consideren óptimos.
- Puntos de parada para mostrar imágenes, o datos por terminal.

Entrega

- Fecha límite: 22 de Diciembre
- Valoración: 15 puntos (+3 puntos)
- Lugar de entrega: PRADO
- **Se valorará mucho la memoria:** descripción de qué se ha hecho y cómo, justificación de las decisiones tomadas, discusión de los resultados obtenidos

Dudas

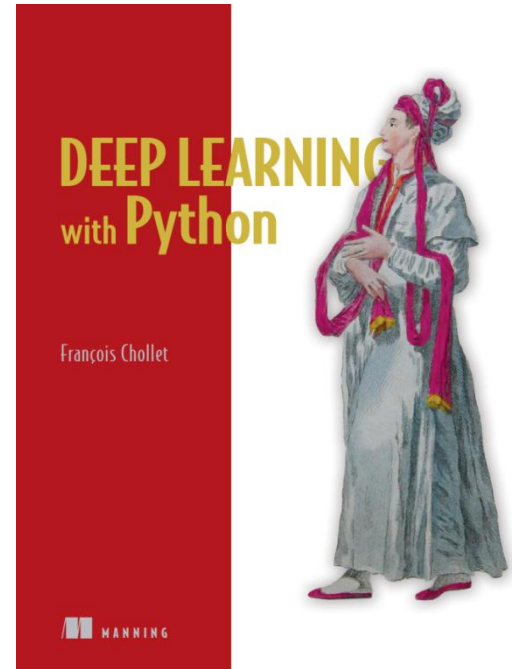
pmesejo@go.ugr.es

Índice

- Normas de entrega
- **Introducción a Keras**
- Presentación de la práctica

Keras

- Las redes profundas y, en particular, las ConvNets se pueden programar en muchos lenguajes diferentes. Nosotros utilizaremos Keras: <https://keras.io/>
- Libro de referencia:
<https://tanhiamhuat.files.wordpress.com/2018/03/deeplearningwithpython.pdf>
- Notebooks compartidos por el autor (François Chollet): <https://github.com/fchollet/deep-learning-with-python-notebooks>
- Otros recursos:
https://keras.io/getting_started/learning_resources/



Keras

- Keras es una API de alto nivel para *Deep Learning* escrita en Python.
- Como backend utiliza TensorFlow (antes también empleaba Theano, CNTK, MXNet,...)
 - De hecho, es la API de alto nivel oficial de TensorFlow.
- La última versión es la 2.7.0 (<https://github.com/keras-team/keras/releases>)
- Documentación: <https://keras.io/>
- Código (GitHub): <https://github.com/keras-team/keras>

Keras

- Acordaos de instalar Keras en vuestros equipos. En el *Anaconda Prompt* podéis hacer

```
pip install tensorflow
```

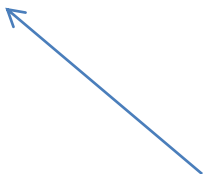
```
pip install keras
```

Nota: si ya habéis instalado TensorFlow y, a pesar de todo, obtenéis el siguiente error *ImportError: Keras requires TensorFlow 2.2 or higher. Install TensorFlow via `pip install tensorflow`*, una posible forma de resolverlo (en Windows) es instalando https://download.visualstudio.microsoft.com/download/pr/d60aa805-26e9-47df-b4e3-cd6fcc392333/7D7105C52FCD6766BEEE1AE162AA81E278686122C1E44890712326634D0B055E/VC_redist.x64.exe

Keras: lectura de imágenes

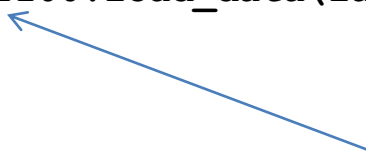
- El vector con las imágenes tendrá dimensión (x, y, z, w):
 - x es el numero de imágenes,
 - y es la altura de las imágenes,
 - z es la anchura de las imágenes,
 - w es el número de canales (1: monobanda; 3: tribanda)
- Ejemplo:

```
(x_train, y_train), (x_test, y_test) = cifar100.load_data(label_mode='fine')
```



(50000, 32, 32, 3)

“50.000 imágenes de 32x32x3”



Hay datasets que ya están incorporados
directamente en Keras <https://keras.io/api/datasets/>

Keras: fases principales

- Las fases principales para crear, entrenar y usar un modelo para clasificación son las siguientes:
 1. Definición del modelo
 2. Declaración del optimizador
 3. Compilación del modelo
 4. Entrenamiento
 5. Predicción

Revisad el script `Ejemplo_mnist.ipynb`, que os proporcionamos como ejemplo, para tener una idea más clara de algunas de estas etapas.

Keras: Definición del modelo

- En Keras hay tres formas de definir modelos de redes neuronales (<https://keras.io/api/models/>): *Sequential*, *Model* y *Model subclassing*. Nos centraremos en los dos primeros.
 - *Sequential* (https://keras.io/guides/sequential_model/) fuerza a que todas las capas de la red vayan una detrás de otra de forma secuencial, sin permitir ciclos ni saltos entre las capas.
 - *Model* o *Functional* (https://keras.io/guides/functional_api/) permite cualquier tipo de red neuronal, incluyendo ciclos y saltos entre capas.
 - *Model subclassing* (https://www.tensorflow.org/guide/keras/custom_layers_and_models) permite implementar cualquier cosa *from scratch*. Se usa si se tienen casos de uso complejos y muy particulares.

Keras: Definición del modelo

- Con *Sequential* podemos usar el método *add* directamente sobre el modelo, y la nueva capa se añadirá después de la última capa añadida.

```
model = Sequential()  
model.add(Dense(50, input_dim=4, activation='relu'))  
model.add(Dense(12, activation='relu'))  
model.add(Dense(3, activation='softmax'))
```

- Con *Model* tenemos que especificar sobre qué capa estamos añadiendo la nueva capa.

```
input1 = Input(shape=(4,))  
hidden1 = Dense(50, activation='relu')(input1)  
hidden2 = Dense(12, activation='relu')(hidden1)  
output = Dense(3, activation='softmax')(hidden2)  
model = Model(inputs=input1, outputs=output)
```

Keras: Definición del modelo

- En nuestro caso, vamos a hacer clasificación multiclase y definiremos como última capa una capa *fully connected* (*Dense* en Keras) con tantas neuronas como clases tenga el problema, y una activación *softmax* para transformar las salidas de las neuronas en la probabilidad de pertenecer a cada clase.

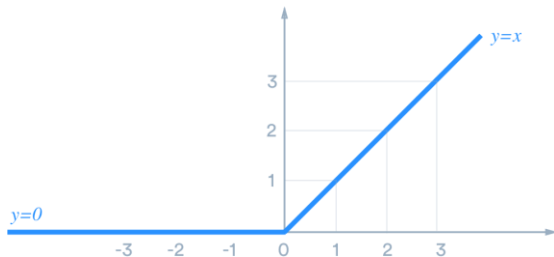
$$\text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^N \exp(z_k)}$$

donde \mathbf{z} es el vector de salida de la capa *Dense* y $\text{softmax}(\mathbf{z})$ es el vector que contiene en la componente j la probabilidad de que la imagen pertenezca a la clase j , para $j = 1, \dots, N$, con N el total de clases.

Keras: Definición del modelo

- El modo más habitual de introducir funciones de activación es detrás de cualquier capa, usando el argumento *activation* de esa capa. Lo siguiente introduciría una activación ReLU en una capa *Dense* con 128 unidades de procesamiento (neuronas):

```
model.add(Dense(128, activation='relu'))
```



Keras: Definición del modelo

En las prácticas vamos a usar algunas de las siguientes capas:

- *Fully connected*: `Dense(units, activation = None, ...)`
- *Dropout*: `Dropout(rate, noise_shape = None, seed = None)`
- *Flatten*: `Flatten()`
- *Convolución 2D*: `Conv2D(filters, kernel_size, strides = (1,1), padding = 'valid', activation = None, ...)`
- *Pooling 2D*: `MaxPooling2D(pool_size = (2,2), strides = None, ...)`. Equivalentemente, `AveragePooling2D()`, `GlobalMaxPooling()`, `GlobalAveragePooling()`,...
- *Batch Normalization*: `BatchNormalization()`

Keras: Definición del modelo

- Tened en cuenta que Keras cuenta con muchos más tipos de capas (<https://keras.io/api/layers/>):

The base Layer class

- Layer class
- weights property
- trainable_weights property
- non_trainable_weights property
- trainable property
- get_weights method
- set_weights method
- get_config method
- add_loss method
- add_metric method
- losses property
- metrics property
- dynamic property

Layer activations

- relu function
- sigmoid function
- softmax function
- softplus function
- softsign function
- tanh function
- selu function
- elu function
- exponential function

Layer weight initializers

- RandomNormal class
- RandomUniform class
- TruncatedNormal class
- Zeros class
- Ones class
- GlorotNormal class
- GlorotUniform class
- Identity class
- Orthogonal class
- Constant class
- VarianceScaling class

Layer weight regularizers

- l1 class
- l2 class
- l1_l2 function

Layer weight constraints

- MaxNorm class
- MinMaxNorm class
- NonNeg class
- UnitNorm class
- RadialConstraint class

Core layers

- Input object
- Dense layer
- Activation layer
- Embedding layer
- Masking layer
- Lambda layer

Convolution layers

- Conv1D layer
- Conv2D layer
- Conv3D layer
- SeparableConv1D layer
- SeparableConv2D layer
- DepthwiseConv2D layer
- Conv2DTranspose layer
- Conv3DTranspose layer

Pooling layers

- MaxPooling1D layer
- MaxPooling2D layer
- MaxPooling3D layer
- AveragePooling1D layer
- AveragePooling2D layer
- AveragePooling3D layer
- GlobalMaxPooling1D layer
- GlobalMaxPooling2D layer
- GlobalMaxPooling3D layer
- GlobalAveragePooling1D layer
- GlobalAveragePooling2D layer
- GlobalAveragePooling3D layer

Recurrent layers

- LSTM layer
- GRU layer
- SimpleRNN layer
- TimeDistributed layer
- Bidirectional layer
- ConvLSTM2D layer
- Base RNN layer

Preprocessing layers

- Core preprocessing layers
- Categorical data preprocessing layers
- Image preprocessing & augmentation layers

Normalization layers

- BatchNormalization layer
- LayerNormalization layer

Regularization layers

- Dropout layer
- SpatialDropout1D layer
- SpatialDropout2D layer
- SpatialDropout3D layer
- GaussianDropout layer
- GaussianNoise layer
- ActivityRegularization layer
- AlphaDropout layer

Attention layers

- MultiHeadAttention layer
- Attention layer
- AdditiveAttention layer

Reshaping layers

- Reshape layer
- Flatten layer
- RepeatVector layer
- Permute layer
- Cropping1D layer
- Cropping2D layer
- Cropping3D layer
- UpSampling1D layer
- UpSampling2D layer
- UpSampling3D layer
- ZeroPadding1D layer
- ZeroPadding2D layer
- ZeroPadding3D layer

Merging layers

- Concatenate layer
- Average layer
- Maximum layer
- Minimum layer
- Add layer
- Subtract layer
- Multiply layer
- Dot layer

Locally-connected layers

- LocallyConnected1D layer
- LocallyConnected2D layer

Activation layers

- ReLU layer
- Softmax layer
- LeakyReLU layer
- PReLU layer
- ELU layer
- ThresholdedReLU layer

Keras: Definición del modelo

- Una vez el modelo esta construido, podemos ver una descripción del mismo usando *summary* sobre el objeto creado:

`my_model.summary()`

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 4)]	0
dense_3 (Dense)	(None, 50)	250
dense_4 (Dense)	(None, 12)	612
dense_5 (Dense)	(None, 3)	39
=====		
Total params: 901		
Trainable params: 901		
Non-trainable params: 0		

```
from keras.models import Model
from keras.layers import Input, Dense

def define_model_by_functional_api():
    input1 = Input(shape=(4,))
    hidden1 = Dense(50, activation='relu')(input1)
    hidden2 = Dense(12, activation='relu')(hidden1)
    output = Dense(3, activation='softmax')(hidden2)
    model = Model(inputs=input1, outputs=output)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Keras: Declaración del optimizador

- Para poder modificar los parámetros del optimizador, es necesario declararlo previamente y crear un objeto. Por ejemplo, para usar el gradiente descendente estocástico deberíamos declararlo y así podríamos cambiar alguno de sus parámetros.

```
import tensorflow as tf
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000,
    decay_rate=0.9)
opt = tf.keras.optimizers.SGD(learning_rate=lr_schedule,
    momentum=0.9)
```

- Documentación de optimizadores: <https://keras.io/optimizers/>

Keras: Compilación del modelo

- La **función de pérdida** (*loss function*), o función objetivo que se va a usar (y que va a ser minimizada), depende del problema a resolver.
 - Clasificación binaria: `binary_crossentropy`
 - Clasificación multiclase: `categorical_crossentropy`
- Documentación sobre las funciones de pérdida disponibles: <https://keras.io/losses/>

Keras: Compilación del Modelo

- Con el argumento *metrics* se pueden especificar las **métricas** que se quieren calcular a lo largo de las épocas de entrenamiento.
 - clasificación multiclase: común usar la métrica *accuracy*, definida como el porcentaje de imágenes bien clasificadas.
- Para **compilar**, usamos el método **compile()**:

```
my_model.compile(loss=keras.losses.categorical_crossentropy,  
                  optimizer=opt, metrics=['accuracy'],...)
```

Keras: Entrenamiento

- Una vez el modelo está compilado, podemos pasar a entrenarlo. Para ello, debéis usar:
 - el método *fit()*: recibe las imágenes de entrenamiento (en un NumPy array, un tensor de TensorFlow, o un *ImageDataGenerator*, entre otros).
 - *ImageDataGenerator*: se emplea parahacer *data augmentation*, para usar alguna función de preprocesado, o para separar un conjunto de validación durante el entrenamiento.

Keras: Entrenamiento

- Cuando se entrena un modelo con *fit()*, Keras guarda el estado del modelo por donde se ha quedado entrenando.
 - Esto quiere decir que si volvemos a usar *fit()*, el entrenamiento seguirá por donde se ha quedado, y no empezará desde el principio.
 - Recomendación: si vamos a usar varias veces *fit()* sobre el mismo modelo definido previamente (con distintos argumentos en *ImageDataGenerator* para, por ejemplo, probar distintos tipos de *data augmentation*) tenemos que restablecer los pesos de la red a como estaban antes del entrenamiento.

Keras: Entrenamiento

- Esto se puede hacer guardando los pesos de la red antes del primer entrenamiento (y después de la compilación) usando:

```
weights = my_model.get_weights()
```

- Y después restablecerlos antes del siguiente entrenamiento usando

```
my_model.set_weights(weights)
```

Keras: Entrenamiento

- ¿Cómo es posible que, cada vez que entrenamos un modelo en Keras, los resultados sean distintos?
 - estamos trabajando con métodos estocásticos
 - inicialización aleatoria de pesos, eliminación aleatoria de unidades en Dropout, transformaciones aleatorias de los datos en *data augmentation*, etc.
 - la función *fit()* va a actualizar continuamente los pesos, de modo que si la llamamos varias veces con el mismo modelo actualizará progresivamente los pesos cada vez
 - Es decir, entrenará incrementalmente el modelo a partir de los pesos encontrados en el anterior entrenamiento.
 - Hay numerosas entradas donde se aborda el tema:
 - <https://stackoverflow.com/questions/62168306/keras-internal-weights-memory-why-multiple-fits-with-one-epoch-still-sees-im>
 - <https://stackoverflow.com/questions/48599464/does-calling-the-model-fit-method-again-reinitialize-the-already-trained-weights>
 - <https://datascience.stackexchange.com/questions/67411/does-keras-model-fit-remember-learning-rate-when-called-multiple-times>

Keras: Entrenamiento

- La clase *ImageDataGenerator*
 - **Nos permite normalizar los datos** (bien con media y varianza, o usando una función de preprocesado determinada), **usar *data augmentation*, o separar del conjunto de entrenamiento una parte para validación** (entre otras cosas).
 - Para usarla, tenemos que crear un objeto de esta clase y usarlo como generador de imágenes a la hora de entrenar y/o testear el modelo.
 - ***Data augmentation* solo debe usarse en el conjunto de entrenamiento.**
 - Se hace *data augmentation* online (cada imagen es transformada/aumentada 1 vez en cada época; y de hecho la imagen original no se llega a ver en ningún momento)
 - Recordad que la normalización debe hacerse en ambos conjuntos, pero **la normalización del conjunto de test debe hacerse con los parámetros de las imágenes de entrenamiento.**
 - Documentación: <https://keras.io/preprocessing/image/>

Keras: Entrenamiento

- La clase *ImageDataGenerator*
 - Se pueden usar los argumentos *featurewise_center* y/o *featurewise_std_normalization*, para normalizar con media 0 y varianza 1 los conjuntos de datos:

```
datagen = ImageDataGenerator( featurewise_center =  
    True, featurewise_std_normalization = True)  
# A continuación, se estiman los parámetros de normalización  
datagen.fit(imagenes_train)
```

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

Keras: Entrenamiento

- Separar un conjunto de validación con la clase *ImageDataGenerator*
 - Para separar un 10% del conjunto de entrenamiento para validación, usaremos la clase *ImageDataGenerator*. Con ella, definiremos un generador *datagen* que se encargará de generar las imágenes de entrenamiento:

```
datagen = ImageDataGenerator(validation_split = 0.1)
```

Keras: Entrenamiento

```
datagen = ImageDataGenerator(validation_split = 0.1)
```

– Atención a los detalles:

- `validation_split` siempre escoge el porcentaje correspondiente a los últimos ejemplos del conjunto de entrenamiento para validación.
 - Dependiendo cómo lo usemos, siempre estaríamos validando exactamente con los mismos ejemplos y, si los ejemplos están ordenados por clase, nada asegura que caigan ejemplos de distintas clases en dicho conjunto de validación (por lo que, al final, podríamos estar validando el modelo solamente con ejemplos de ciertas clases)

Keras: Entrenamiento

```
datagen = ImageDataGenerator(validation_split = 0.1)
```

– Atención a los detalles:

- Si queremos más control, podríamos hacer algo como:

```
from sklearn.model_selection import train_test_split
X_train, X_val, Y_train, Y_val = train_test_split(x_train, y_train,
    test_size=0.1, stratify=y_train)

it_train = datagen.flow(X_train , Y_train, batch_size = batch_size)
it_validation = datagen.flow(X_val , Y_val)
```

datagen.flow() toma arrays de datos y etiquetas, y genera batches de datos aumentados.

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator#flow

Keras: Entrenamiento

- Usar *fit()* con el conjunto de validación
 - *datagen.flow()* puede ser el primer argumento de la función *fit()*.
 - **Tendremos que usar también el argumento *validation_data* para especificar el conjunto de validación** y hacerlo usando el mismo generador, pero especificando en *subset* que queremos generar el conjunto de validación:


```
my_model.fit(datagen.flow(imagenes_train,
    etiquetas_train, batch_size = 32, subset =
    'training'), validation_data =
    datagen.flow(imagenes_train, etiquetas_train,
    batch_size = 32, subset = 'validation'))
```

Keras: Entrenamiento

- *fit()* tiene otros tres parámetros a tener en cuenta:
 - ***steps_per_epoch***: número total de pasos (*batches* de imágenes) que se usan antes de terminar una época del entrenamiento y pasar a la siguiente.
 - Igual al número de imágenes dividido por el tamaño del *batch* (es decir, $\text{ceil}(\text{num_samples} / \text{batch_size})$).
 - ***epochs***: número de épocas durante las que se entrena la red.
 - ***validation_steps***: igual que *steps_per_epoch* pero, en lugar de en *training*, en validación. Número de *batches* de imágenes de validación que se generan al final de cada época.
 - Igual al número de imágenes en validación entre el tamaño de cada *batch*.
 - *If 'validation_steps' is None, validation will run until the validation_data is exhausted.*

Keras: Entrenamiento

- Finalmente, una llamada para entrenar un modelo puede quedar parecido a lo siguiente:

```
my_model.fit(  
    datagen.flow(imagenes_train, etiquetas_train,  
    batch_size=32, subset='training'),  
    steps_per_epoch = None,   $\text{math.ceil}(\text{imagenes\_train.shape}[0] // \text{batch\_size})$   
    epochs = 100,  
    validation_data = datagen.flow(X_val , Y_val) ,  
    validation_steps = None)
```

Keras: Entrenamiento

- Emplearemos dos *ImageDataGenerator* (uno para entrenamiento y otro para test):
 - El **generador de *train*** tendrá los parámetros correspondientes a la normalización de los datos de entrada, el *data augmentation*, y el porcentaje que se guarde para validación.
 - El **generador de test** solo tendrá la normalización (que se hará con los parámetros obtenidos de las imágenes de *train*).

Keras: Predicción

- Se usa la función *predict()*, de forma análoga a *fit()*.
 - https://www.tensorflow.org/api_docs/python/tf/keras/Sequential#predict

```
predicciones = my_model.predict(imagenes_test)
```

Keras: Cálculo de Accuracy

- Una vez tenemos las predicciones, podemos calcular el porcentaje de ejemplos de test que el modelo clasifica bien (*accuracy*).

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```



```
def calcularAccuracy(labels, preds):
    labels = np.argmax(labels, axis = 1)
    preds = np.argmax(preds, axis = 1)
    accuracy = sum(labels == preds)/len(labels)
    return accuracy
```

Revisad Ejemplo_mnist.ipynb. Contiene muchas ideas útiles para realizar la práctica.

Keras: Redes Pre-entrenadas

- Keras tiene algunas redes populares ya creadas
 - no es necesario construirlas desde cero cada vez.
- Están preentrenadas en *ImageNet*
 - si se quiere, se puede partir el entrenamiento desde ahí.
- Estos modelos están en <https://keras.io/applications/>
- Cada red dispone de una función de preprocesado *preprocess_input()* distinta, que habrá que usar con cada modelo. Se le puede pasar como argumento al generador de la clase *ImageDataGenerator*.

Keras: Redes Pre-entrenadas

- Supongamos que queremos cargar una ResNet50 preentrenada en ImageNet. Tenemos que quitarle la última capa de 1000 neuronas (usando el argumento *include_top* al cargar la red):

```
from keras.applications.resnet import  
    ResNet50, preprocess_input  
  
resnet50 = ResNet50(include_top = False, weights  
    = 'imagenet', pooling = 'avg')
```

- El argumento pooling = 'avg' introduce *GlobalAveragePooling* después de lo que ahora será la última capa, que es una convolución2D.

Keras: Redes Pre-entrenadas

- El modelo resnet50 que acabamos de crear lo podemos usar como **extractor de características** (usando *predict()*).
- La última capa de nuestro modelo tiene 2048 neuronas.
 - Por tanto, podemos considerar que estamos transformando cada imagen en un vector de 2048 características.

Con este vector de características podríamos entrenar otro modelo, como un SVM, un Random Forest o una red con varias capas *fully connected* (perceptrón multicapa).

```
conv5_block3_2_bn (BatchNormal (None, None, None, 2048  ['conv5_block3_2_conv[0][0]']
ization) 512)

conv5_block3_2_relu (Activatio (None, None, None, 0  ['conv5_block3_2_bn[0][0]']
n) 512)

conv5_block3_3_conv (Conv2D) (None, None, None, 1050624  ['conv5_block3_2_relu[0][0]']
2048)

conv5_block3_3_bn (BatchNormal (None, None, None, 8192  ['conv5_block3_3_conv[0][0]']
ization) 2048)

conv5_block3_add (Add) (None, None, None, 0  ['conv5_block2_out[0][0]',
2048)  'conv5_block3_3_bn[0][0]']

conv5_block3_out (Activation) (None, None, None, 0  ['conv5_block3_add[0][0]']
2048)

avg_pool (GlobalAveragePooling (None, 2048) 0  ['conv5_block3_out[0][0]']
2D)
```

```
=====
Total params: 23,587,712
Trainable params: 0
Non-trainable params: 23,587,712
```

Keras: Redes Pre-entrenadas

- Otra opción sería reentrenar la red entera (o parte de ella) adaptándola a nuestro problema concreto.
 - Es lo que se llama **fine-tuning** (ajuste fino).
- Para ello, como mínimo, es necesario añadir al final del modelo una capa *fully connected* con tantas neuronas como clases tenga el problema y activación *softmax*.

```
x = resnet50.output
x = Dense(32, activation = 'relu')(x)
last = Dense(10, activation = 'softmax')(x)
new_model = Model(inputs = resnet50.input, outputs = last)
```

Índice

- Normas de entrega
- Introducción a Keras
- **Presentación de la práctica**

Objetivos

El objetivo de esta práctica es obtener experiencia práctica en el diseño y entrenamiento de redes neuronales convolucionales profundas, usando Keras. A partir de una arquitectura base de red que se proporciona, hay que aprender a experimentar con ella y mejorarla a partir de añadir, modificar o suprimir capas de dicha arquitectura en la tarea de clasificar imágenes en 25 categorías.

Para realizar esta práctica se proporciona el siguiente código/funciones de ayuda:

Revisad Ejemplo_mnist.ipynb!!!

1. Funciones básicas de lectura de datos
2. Creación de gráficas para la evolución del porcentaje de clasificación en el conjunto de entrenamiento y en el de validación. (apartados 1 y 2)
3. Cálculo del porcentaje de clasificación en el conjunto de prueba (apartado 3)

Apartado 1: BaseNet en CIFAR100 (4 ptos)

Conjunto de datos

En este apartado, se trabajará con una parte del conjunto de datos CIFAR100. Este conjunto de datos consta de 60K imágenes en color de dimensión 32x32x3 (RGB) de 100 clases distintas, con 600 imágenes por clase. Hay 50K imágenes para entrenamiento y 10K imágenes de prueba. Para el desarrollo de práctica solo consideraremos 25 clases de las 100, por tanto el conjunto de entrenamiento tiene 12500 imágenes y el de prueba 2500. Del conjunto de entrenamiento se usará un 10% para validación. Usar las funciones dadas para conseguir dicha reducción.



Apartado 1: BaseNet en CIFAR100 (4 ptos)

Modelo base: BaseNet

Comenzamos creando un modelo base llamado BaseNet, que tras su entrenamiento y ejecución nos dará un porcentaje de clasificación de referencia para las posteriores mejoras.

Layer No.	Layer Type	Kernel size (for conv layers)	Input Output dimension	Input Output channels (for conv layers)
1	Conv2D	5	32 28	3 6
2	Relu	-	28 28	-
3	MaxPooling2D	2	28 14	-
4	Conv2D	5	14 10	6 16
5	Relu	-	10 10	-
6	MaxPooling2D	2	10 5	-
7	Linear	-	400 50	-
8	Relu	-	50 50	-
9	Linear	-	50 25	-

**Arquitectura
que tenéis que
implementar
en Keras**

Apartado 1: BaseNet en CIFAR100 (4 ptos)

- 1.- Familiarizarse con la arquitectura BaseNet ya proporcionada, el significado de los hiperparámetros y la función de cada capa. Crear el código para el modelo BaseNet
- 2.- Entrenar el modelo y extraer los valores de accuracy y función de perdida para el conjunto de test. Presentar los resultados de entrenamiento y test usando las funciones proporcionadas.

Apartado 2: Mejora del modelo BaseNet (5 puntos)

- Una vez habéis implementado y validado BaseNet, debéis mejorar la red por medio de aquellas alternativas que juzguéis vosotros:
 - Normalización de datos
 - Aumento de datos
 - Aumento de profundidad de la red
 - Batch Normalization
 - Regularización
 - Dropout
 - Early-Stopping
 - ¿Otros?
- Recordad justificar siempre vuestras decisiones y mostrar claramente en la memoria la arquitectura final resultante.

Apartado 3: Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD (6 puntos).

1.- Usar ResNet50 como un extractor de características para los datos de **Caltech-UCSD** disponible en (<http://www.vision.caltech.edu/visipedia/CUB-200-2011.html>). Para ello eliminaremos al menos las dos últimas capas del modelo descargado, añadiremos algunas capas de cálculo adicional y la capa de salida. En concreto realizar los siguientes experimentos:

<http://www.vision.caltech.edu/visipedia/CUB-200.html>

Si tenéis problemas a la hora de leer los datos desde Google Drive:
<https://stackoverflow.com/questions/59120853/google-colab-is-so-slow-while-reading-images-from-google-drive>

Apartado 3: Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD (6 puntos).

A.- a) Adaptar el modelo ResNet50 entrenado con ImageNet a los datos de Caltech-UCSD y estimar su desempeño con estos datos. b) Eliminar las capas finales FC y la salida, sustituirlas por nuevas FC y salida y reentrenarlas con Caltech-UCSD. c) Comparar resultados con el modelo anterior en el que únicamente se cambia y reentrena la capa de salida

B.- Eliminar las capas de salida, FC y AveragePooling. En este momento tendrá un extractor de características. Añada nuevas capas que mezclen dichas características y den una clasificación. Entrene la red resultante y compare sus resultados con los resultados del punto.A

2.- Realizar un ajuste fino de toda la red ResNet50, al conjunto de datos. Caltech-UCSD. Recordar que el número de épocas a ejecutar debe ser pequeño.

ResNet50 como extractor de características

Eliminamos la FC 1000, y metemos una FC adecuada a la dimensionalidad de nuestro problema. Podemos añadir también lo que queramos (Dropout, BN, varias FC,...).
Reentrenamos los nuevos pesos.

Luego, comparamos los resultados anteriores con solo meter una FC adecuada a la dimensionalidad de nuestro problema, y reentrenar los nuevos pesos.

Apartado 3: Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD (6 puntos).

A.- a) Adaptar el modelo ResNet50 entrenado con ImageNet a los datos de Caltech-UCSD y estimar su desempeño con estos datos. b) Eliminar las capas finales FC y la salida, sustituirlas por nuevas FC y salida y reentrenarlas con Caltech-UCSD. c) Comparar resultados con el modelo anterior en el que únicamente se cambia y reentrena la capa de salida

B.- Eliminar las capas de salida, FC y AveragePooling. En este momento tendrá un extractor de características. Añada nuevas capas que mezclen dichas características y den una clasificación. Entrene la red resultante y compare sus resultados con los resultados del punto.A

2.- Realizar un ajuste fino de toda la red ResNet50, al conjunto de datos. Caltech-UCSD. Recordar que el número de épocas a ejecutar debe ser pequeño.

ResNet50 como extractor de características

Hacemos lo mismo que antes, pero también eliminando el Global Average Pooling. En este caso, se pueden incluir también capas convolucionales.

Apartado 3: Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD (6 puntos).

A.- a) Adaptar el modelo ResNet50 entrenado con ImageNet a los datos de Caltech-UCSD y estimar su desempeño con estos datos. b) Eliminar las capas finales FC y la salida, sustituirlas por nuevas FC y salida y reentrenarlas con Caltech-UCSD. c) Comparar resultados con el modelo anterior en el que únicamente se cambia y reentrena la capa de salida

B.- Eliminar las capas de salida, FC y AveragePooling. En este momento tendrá un extractor de características. Añada nuevas capas que mezclen dichas características y den una clasificación. Entrene la red resultante y compare sus resultados con los resultados del punto.A

2.- Realizar un ajuste fino de toda la red ResNet50, al conjunto de datos. Caltech-UCSD. Recordar que el número de épocas a ejecutar debe ser pequeño.

Ajuste fino de ResNet50

Partimos de los pesos de ResNet50 (entrenados con ImageNet) y debemos hacer fine-tuning con el nuevo dataset. Se hace esto con toda la red, y no con un fragmento de la misma.

Bonus: propuestas innovadoras de mejora de los modelos propuestos

BONUS : Solo se tendrán en cuenta los bonus si se ha logrado al menos el 75% de los puntos en la parte obligatoria.

Bonus.1 (1-3 puntos) (Hacer propuestas y discutirla con el profesor) Hay muchas otras posibilidades para mejorar el modelo BaseNet sobre CIFAR-100 usando combinaciones adecuadas de capas. Siéntase libre de probar sus propias ideas o enfoques interesantes de AA / VC sobre los que haya leído.

Se valorará cada propuesta en función de su innovación, complejidad y buen uso de Keras. El número de clases usadas en el experimento también se tendrá en cuenta.

Dado que Colab solo ofrece recursos computacionales limitados, intente limitar racionalmente el tiempo de entrenamiento y el tamaño del modelo.

Prácticas de Visión por Computador

Grupo 2

Presentación de la Práctica 3

Introducción a Keras

Pablo Mesejo

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial



UNIVERSIDAD
DE GRANADA

