Prácticas de Visión por Computador
Grupo 2

Presentación de la Práctica 2:
Redes Neuronales Convolucionales

Pablo Mesejo
Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD
DE GRANADA

Índice

- Normas de entrega
- Breve repaso de redes convolucionales
- Introducción a Keras
- Presentación de la práctica

Índice

- **Normas de entrega**
- Breve repaso de redes convolucionales
- Introducción a Keras
- Presentación de la práctica

Normas de la Entrega de Prácticas

- Uno o dos ficheros Python (podéis usar como plantillas: **`esquemaCodigo_parte1y2.py`** y **`esquemaCodigo_parte3.py`**).
- El código debe estar comentado.
- Se entrega memoria (PDF) y código (Python) → ZIP/RAR

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”
- Todos los resultados numéricos serán mostrados por pantalla. No escribir nada en el disco.
- La práctica deberá poder ser ejecutada de principio a fin sin necesidad de ninguna selección de opciones. Hay que fijar los parámetros que se consideren óptimos.
- Puntos de parada para mostrar imágenes, o datos por terminal.

Entrega

- Fecha límite: 28 de Noviembre
- Valoración: hasta 8 puntos
- Lugar de entrega: PRADO
  https://pradogrado2021.ugr.es/course/view.php?id=14596#section-4

- **Se valorará mucho la memoria:** descripción de qué se ha hecho y cómo, justificación de las decisiones tomadas, discusión de los resultados obtenidos

Organización Prácticas

- Podéis ir al grupo que queráis, pero os corregirá vuestro ejercicio el profesor del grupo al que estéis asignados.
- Enviar dudas o solicitar tutorías online a pmesejo@decsai.ugr.es, pmesejo@go.ugr.es o pablomesejo@gmail.com.
  - Preferentemente Martes y Miércoles de 10:00 a 12:00 y Viernes de 11:30 a 13:30.

Índice

- Normas de entrega
- **Breve repaso de redes convolucionales**
- Introducción a Keras
- Presentación de la práctica

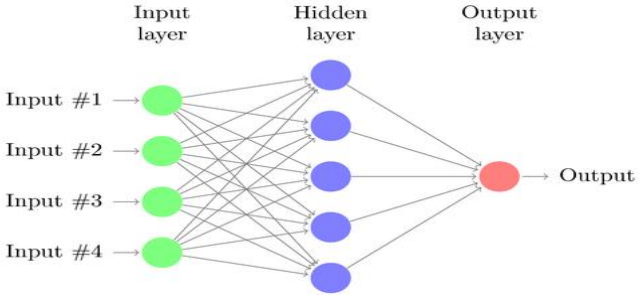
Breve repaso de ConvNets

- Muy recomendable consultar el curso de Stanford:
  - CS231n: Convolutional Neural Networks for Visual Recognition (http://cs231n.stanford.edu/2018/index.html)

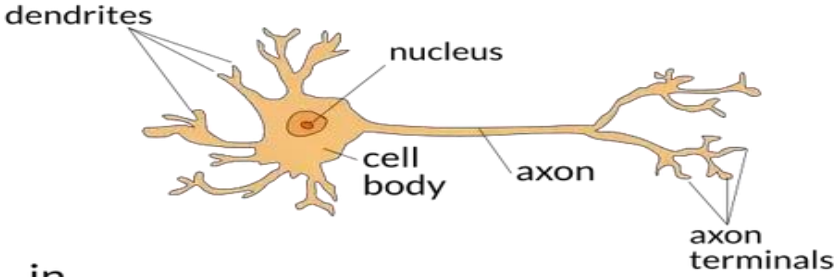
- En concreto, diapositivas sobre Convolutional Neural Networks (ConvNets): http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture05.pdf

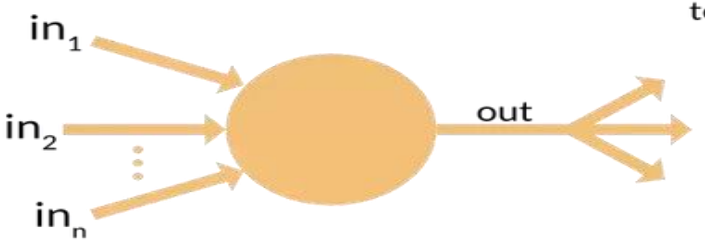
¿Qué son las Redes de Neuronas Artificiales (RNNs)?

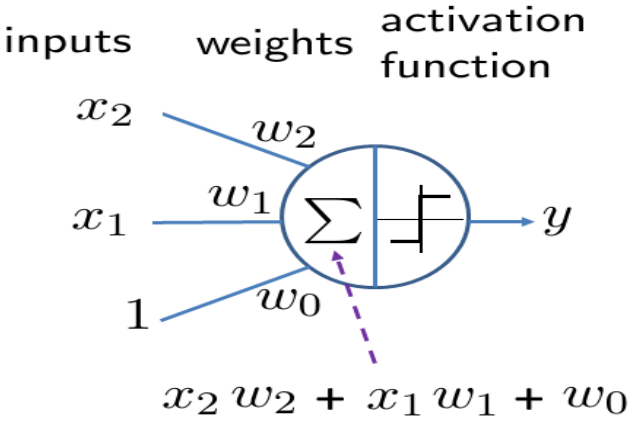
Conjunto de unidades simples de procesado altamente interconectadas y que, a través de un proceso de aprendizaje, resuelven una tarea concreta



¿Qué es una neurona artificial?

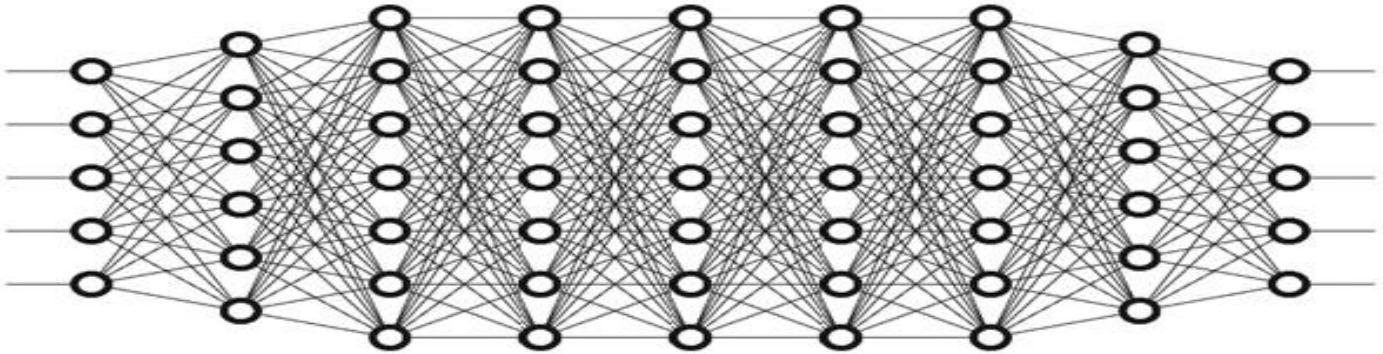




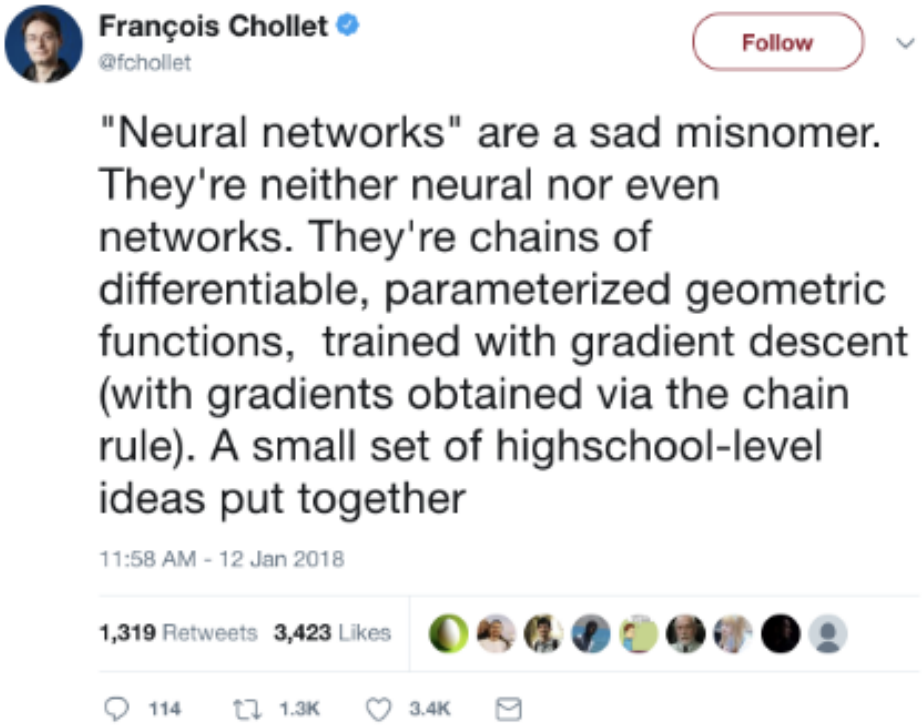


¿Qué son las Redes Neuronales Profundas?

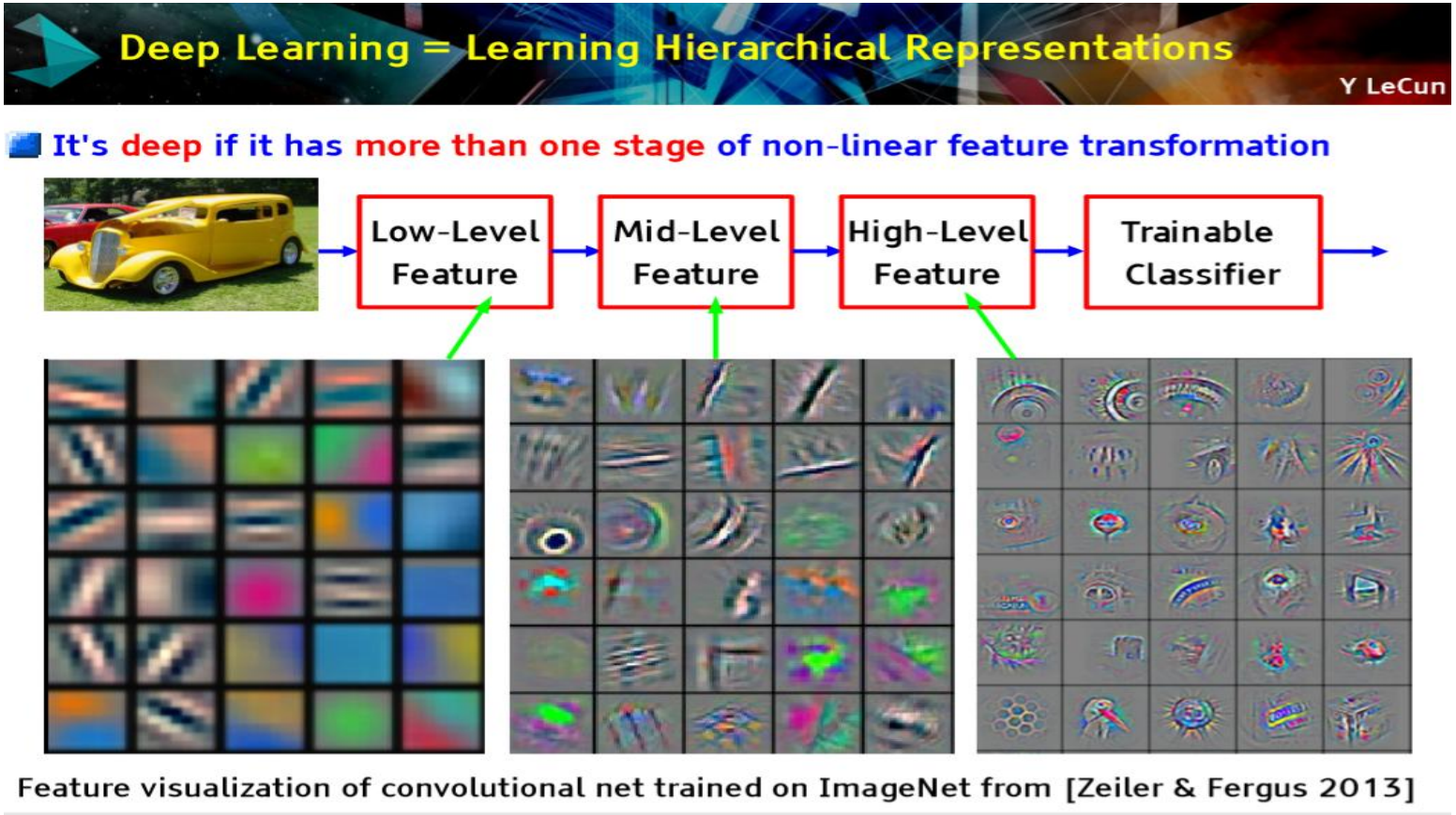
- Modelos computacionales compuestos por **numerosas capas de procesado** y empleados para aprender **representaciones con múltiples niveles de abstracción** a partir de los datos de entrada.



¿Qué son las Redes Neuronales Profundas?



¿Qué son las Redes Neuronales Profundas?



Breve repaso de ConvNets

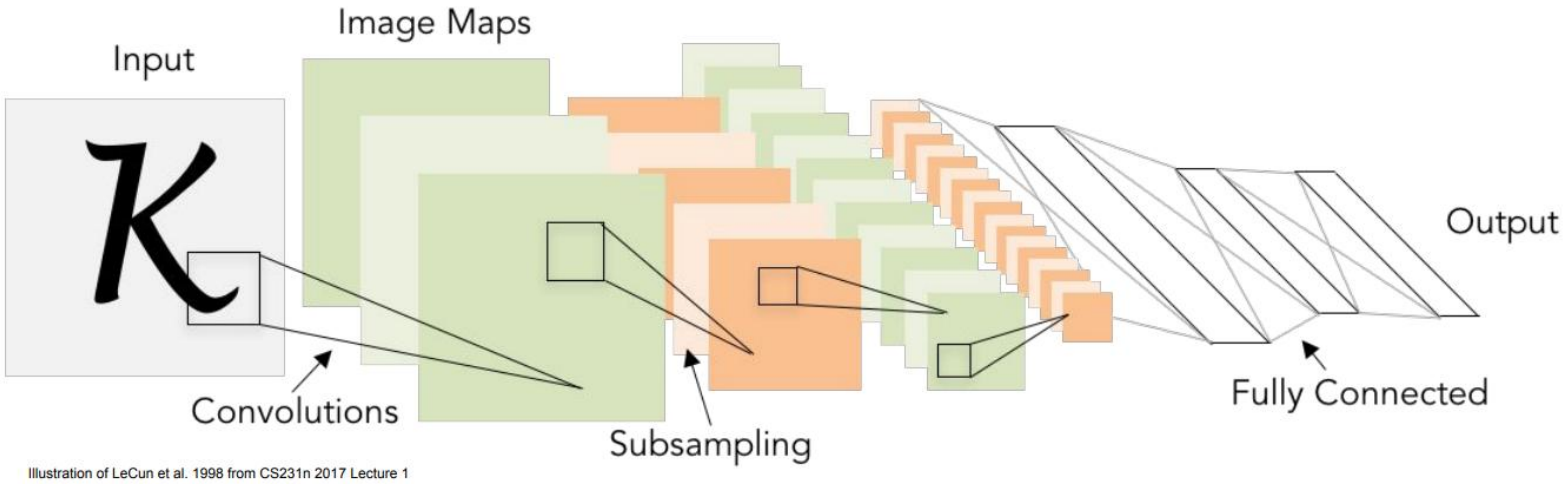


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

- Red neuronal aplicada comúnmente al análisis de imágenes.
- Emplea convoluciones para extraer características (*feature maps*).
- Las máscaras de convolución se aprenden (pesos de la red).

Breve repaso de ConvNets

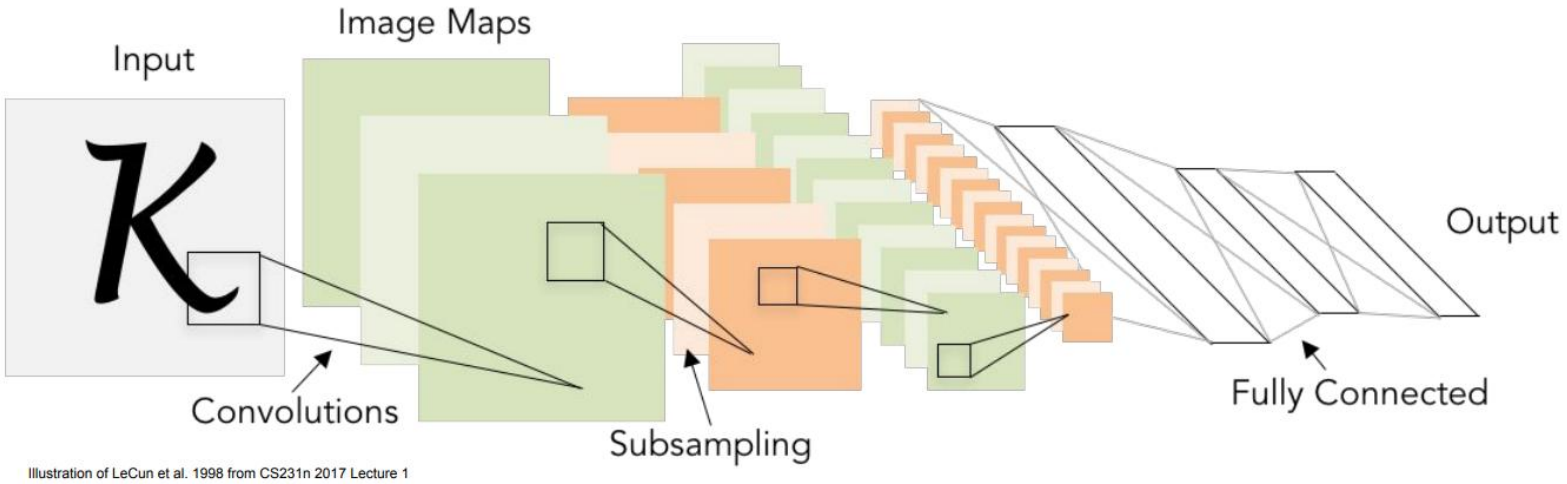
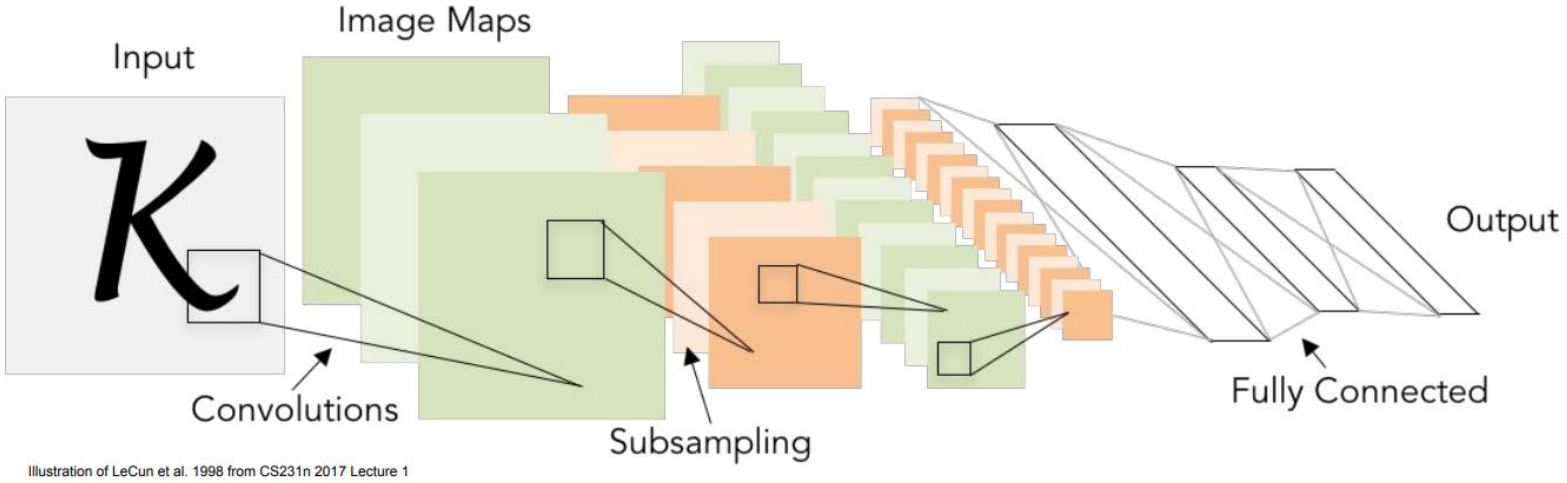


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

- Puede contener, o no, capas totalmente conectadas (*fully connected layers*).
- Incluye capas de *pooling* para reducir dimensionalidad (*subsampling*).

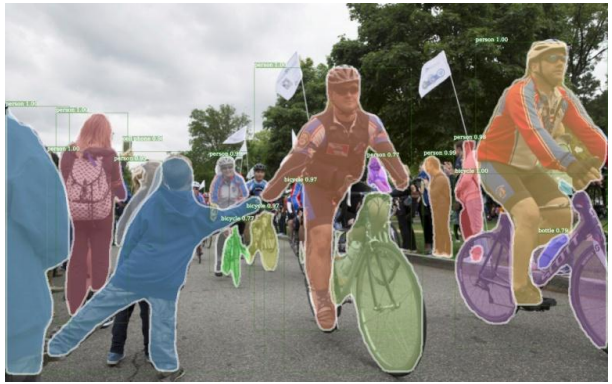
Breve repaso de ConvNets



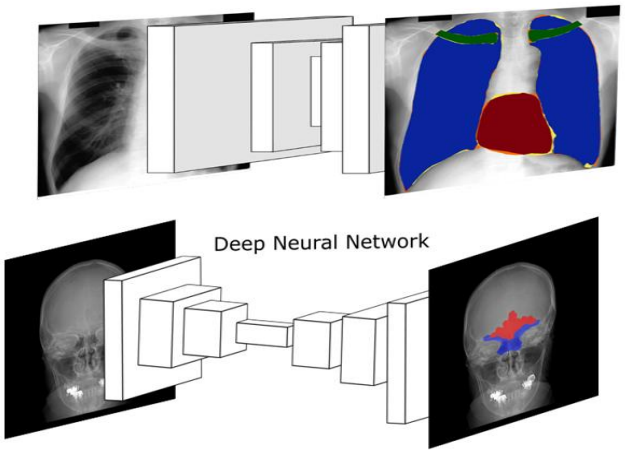
- El origen de las ConvNets puede encontrarse en el Neocognitron (Fukushima, 1980): modelo inspirado por los trabajos de Hubel y Wiesel (años 50 y 60) sobre la estructura y función del córtex visual.

Breve repaso de ConvNets

- A día de hoy representan el estado del arte en multitud de problemas en visión por computador: clasificación de imágenes, detección de objetos, segmentación semántica, reconocimiento facial, estimación de la pose de la cabeza o del cuerpo, descripción automática de imágenes y vídeos, estimación de la edad… y tantos otros.



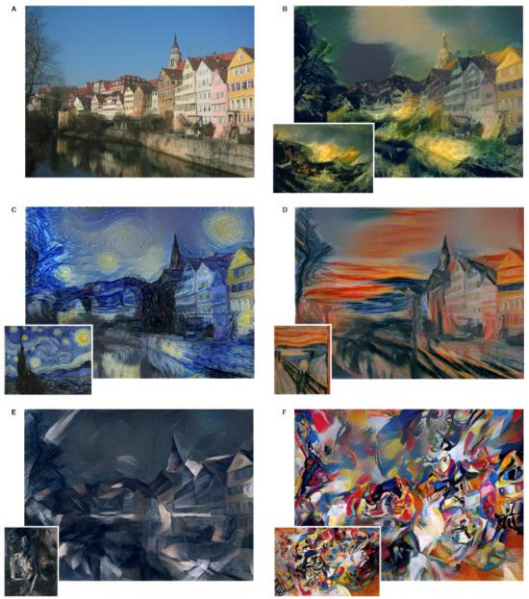
Mask R-CNN output from
https://github.com/facebookresearch/Detectron



"Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields" (CVPR'17)

Breve repaso de ConvNets

Transferencia de Estilo

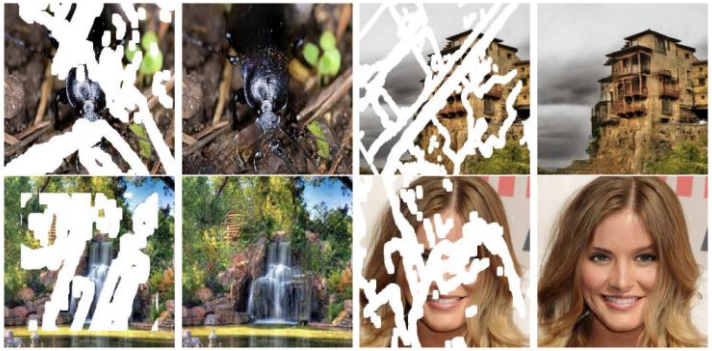


"A Neural Algorithm of Artistic Style" (Gatys et al., 2015)

Colorización de Imágenes

"Colorful Image Colorization" (Zhang et al., 2016)

Reconstrucción de Imágenes



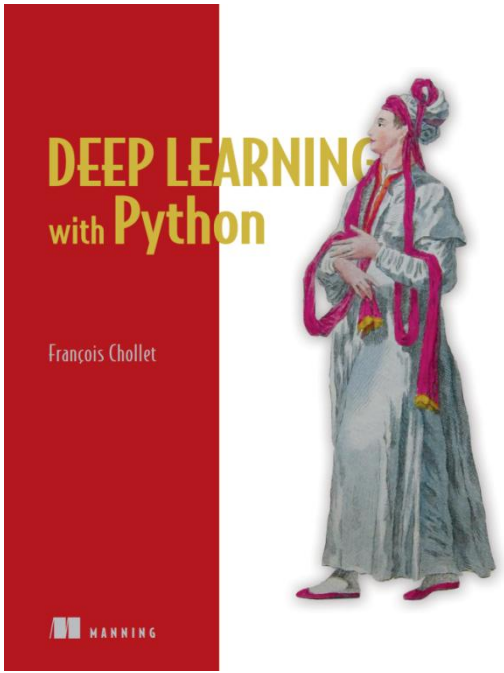
"Image Inpainting for Irregular Holes Using Partial Convolutions" (Liu et al., 2018)

Índice

- Normas de entrega
- Breve repaso de redes convolucionales
- **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: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.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 2.4.0 (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)
- Este vector podremos tenerlo en memoria si todas las imágenes entran en ella (como es el caso de esta práctica). Si no entrasen, Keras tiene la función **`flow_from_directory()`**, que lee las imágenes de un directorio, las descarta después de usarlas y coge el siguiente *batch* de imágenes.

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.py`**, 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.

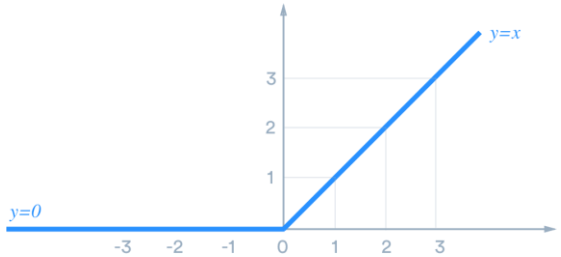
$$\mathrm{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{N} \exp(z_j)}$$

donde **z** es el vector de salida de la capa Dense y *softmax*(**z**) es el vector que contiene en la componente *j* la probabilidad de que la imagen pertenezca a la clase *j*, para $j = 1,\ldots,N$, 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 la capa *Dense*:

```
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
_________________________________________________________________
```

```python
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.

```
from keras.optimizers import SGD
opt = SGD(lr = 0.01, decay = 1e-6, momentum = 0.9,
   nesterov = True)
```

- 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(optimizer, loss =
  'categorical_crossentropy', metrics =
  ['accuracy'], ...)
```

Keras: Entrenamiento

- Una vez el modelo está compilado, podemos pasar a entrenarlo. Para ello, se puede usar:
  - el método *fit()*: recibe las imágenes de entrenamiento directamente. Se usa si todas las imágenes están cargadas en memoria y no vamos a usar la clase *ImageDataGenerator.*
  - el método *fit_generator()*: recibe un generador que será el que se encargará de ir generando las imágenes. Se usa cuando las imágenes no están todas en memoria o se quiere usar *ImageDataGenerator* (bien para *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()* o *fit_generator()*, Keras guarda el estado del modelo por donde se ha quedado entrenando.
  - Esto quiere decir que si volvemos a usar una de las funciones anteriores, el entrenamiento seguirá por donde se ha quedado, y no empezará desde el principio.
  - Recomendación: si vamos a usar varias veces *fit()* o *fit_generator()* sobre el mismo modelo definido previamente (con distintos argumentos en *ImageDataGenerator* para, por ejemplo, probar distintos tipos de *data augmentation*) tenemos que reestablecer 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

- La clase *ImageDataGenerator*
  - Esta clase nos permite normalizar los datos (bien con media y varianza, o usando una función de preprocesado), 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.
  - 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*
  - Si se usan los argumentos *featurewise_center* y/o *featurewise_std_normalization,* para normalizar con media 0 y varianza 1 los conjuntos de datos, es necesario usar la función *fit()* sobre el generador creado:

```
datagen = ImageDataGenerator(
  featurewise_center = True,
  featurewise_std_normalization = True)
datagen.fit(imagenes_train)
```

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)
```

  - Si hemos especificado el argumento *validation_split*, cuando estemos generando imágenes con la función *flow()* sobre el generador, podremos especificar si queremos generar el conjunto de entrenamiento o el de validación:

```
datagen.flow(imagenes_train, etiquetas_train,
  batch_size = 32, subset = 'training')
```

Keras: Entrenamiento

- Usar *fit_generator* con el conjunto de validación
  - Este *datagen.flow()* será el primer argumento de la función *fit_generator*.
  - 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_generator(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

- Usar *fit_generator*
  - Fit_generator 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. Típicamente es igual al número de imágenes dividido por el tamaño del *batch* (es decir, *ceil(num_samples / 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. Suele ser igual al número de imágenes en validación entre el tamaño de cada *batch*.

Keras: Entrenamiento

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

```
my_model.fit_generator(datagen.flow(imagenes_train,
     etiquetas_train, batch_size = 32, subset =
     'training'),
    steps_per_epoch = len(imagenes_train)*0.9/32,
    epochs = 30,
    validation_data = datagen.flow(imagenes_train,
     etiquetas_train, batch_size = 32, subset =
     'validation'),
    validation_steps = len(imagenes_train)*0.1/32)
```

Keras: Entrenamiento

- La clase ImageDataGenerator
  - En el modelo final, la llamada a la clase *ImageDataGenerator* para la creación de los objetos *datagen* (uno para entrenamiento y otro para test) tendrá varios parámetros:
    - 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 usan las funciones *predict_generator()* o *predict()*, en función de si se está usando un generador para el conjunto de test o no, de forma análoga a *fit()*.
- Si se usa *predict_generator()* es necesario usar los argumentos *shuffle = False* y *batch size = 1* de *flow* para que las predicciones de las nuevas imágenes estén en el mismo orden que las imágenes de test, y se pueda comparar con las etiquetas reales.

```
predicciones = my_model.predict_generator(
    datagen_test.flow(imagenes_test,
    batch_size = 1, shuffle = False),
    steps = len(imagenes_test))
```

Keras: Cálculo de Accuracy

- Una vez tenemos las predicciones hechas, podemos calcular el porcentaje de imágenes de test que el modelo ha clasificado bien (la métrica *accuracy*).
- Para ello, en las funciones dadas para la práctica, se proporciona la función *calcularAccuracy(labels, preds)*
  - *labels* es el vector de las etiquetas de test reales
  - *preds* es el vector de predicciones devuelto por *fit_generator*.

Keras: Redes Pre-entrenadas

- Keras tiene algunas de las redes más populares ya creadas, por lo que no es necesario construirlas desde cero cada vez.
- Además, también están preentrenadas en *ImageNet*, de forma que si se quiere, se puede partir el entrenamiento desde ahí.
- Estos modelos están en Keras Applications: 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, preentrenado en ImageNet y habiéndole quitado la última capa de 1000 neuronas, lo podemos usar como **extractor de características** (usando *predict_generator()*).
- En concreto, 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).

Keras: Redes Pre-entrenadas

- Otra opción sería reentrenar la red entera (o parte de ella) para que clasificase nuestro problema. 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 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
- Breve repaso de redes convolucionales
- 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:

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 (2 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 (2 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 (2 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 (3 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 (3 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:

A.- Eliminar las FC y la salida, substituirlas por nuevas FC y salida y reentrenarlas con CALTECH. Comparar resultados con un modelo en el que unicamente se cambia y reentrena la capa de salida

B.- Eliminar las capas de salida, FC y AveragePooling. Añadir nuevas capas, entrenar la red resultante y comparar 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 ejeutar debe ser pequeño.

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) 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 ML / CV sobre los que haya leído.

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

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.

Agradecimientos

- Gran parte de los materiales de introducción a Keras fueron desarrollados por Anabel Gómez Ríos, que permitió su reutilización y modificación para esta clase.

Prácticas de Visión por Computador
Grupo 2

Presentación de la Práctica 2:
Redes Neuronales Convolucionales

Pablo Mesejo
Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD
DE GRANADA