

Laboratorio Nº 8- Aprendizaje Automático en Visión Artificial			
Asignatura	Tópicos Especiales - Visión Artificial	Código	0756
Profesor	José Carlos Rangel Ortiz		

Introducción

Siguiendo con el módulo de Machine Learning, dentro de este podemos encontrar el Deep Learning, el cual se fundamenta en la utilización de Redes Neuronales Artificiales con una gran cantidad de capas para el procesamiento y aprendizaje de la información que se suministre.

Dentro de los algoritmos que utilizan redes neuronales se pueden citar las Redes Neuronales Convolucionales (CNN, por sus siglas en inglés) las cuales se utilizan principalmente en el ámbito de la visión artificial, en problemas de localización, detección y clasificación de objetos, personas y lugares.

En este laboratorio se presentará una forma de construir modelos de clasificación utilizando una CNN. La diferencia radicará en que durante esta parte del laboratorio el modelo será entrenado utilizando datos (imágenes) que se encuentran localmente en nuestra computadora, o dicho de otra manera, un dataset local y también mediante una librería especializada en el trabajo con redes neuronales profundas. El laboratorio muestra todo el procedimiento y permite también evaluar el modelo generado utilizando una imagen externa a las disponibles en el dataset.

De igual manera se presenta la forma para la utilización de un modelo que ha sido previamente entrenado con un dataset de gran tamaño y con arquitecturas de múltiples capas.

El laboratorio puede ser desarrollado construyendo el *script* completamente y ejecutando desde consola o mediante *jupyter lab* realizando una ejecución por celdas.

Objetivo

Aplicar los diferentes conceptos relacionados al aprendizaje automático enfocado en problemas de visión artificial.

Entregables

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

1. Clasificador con Redes Convolucionales y TensorFlow

1.1. Dataset para el laboratorio

En este código construiremos desde cero el código para crear un modelo de clasificación entre gatos, perros y loros. Para ello utilizaremos un dataset que emplea las imágenes de las categorías “Cat”, “Dog” y “Parrot” del dataset Imagenet y recopiladas de Google Images. Para ello, el dataset ya viene separado en las secuencias de entrenamiento(*train*), *test* y validación(*validation*), los cuales se utilizan para entrenar, verificar y evaluar el modelo, respectivamente.

Una vez descargue el dataset (Disponible en [Pets_Dataset](#)), debe verificar que mantenga la estructura requerida, la cual se aprecia en las siguientes imágenes. La carpeta raíz recibe el nombre de “pets_dataset”. El cual debe tener la siguiente estructura una vez que se abre su carpeta contenedora (Ver Figura 1). Disponible también en Google Drive [Pets_Dataset](#).

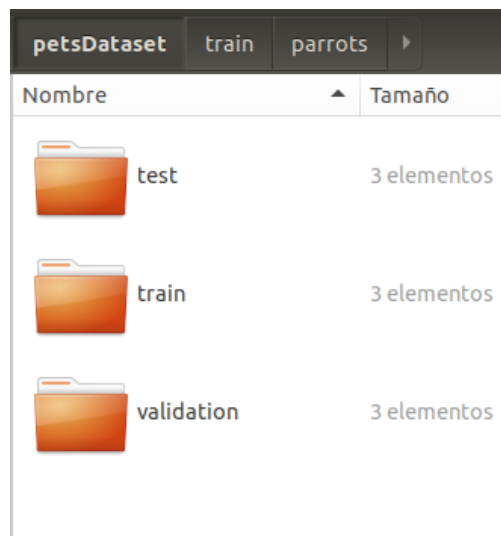
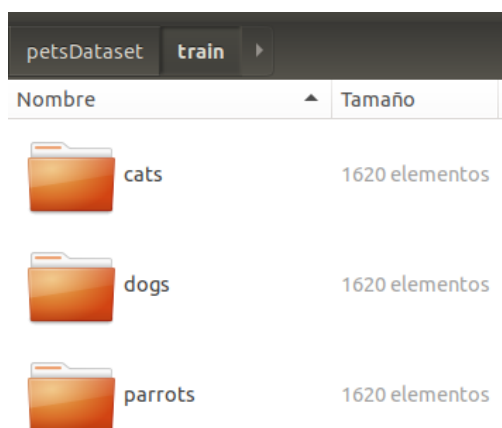


Figura 1: Carpeta Raíz del Dataset.

Dentro de cada una de estas carpetas se encuentran con conjunto de imágenes separadas en 3 clases/categorías. Siendo estas gatos, loros y perros. El interior de cada una de estas carpetas se debe ver como las Figuras 2, 3, 4, para los conjuntos de Entrenamiento, Test y Validación, respectivamente:



Nombre	Tamaño
cats	1620 elementos
dogs	1620 elementos
parrots	1620 elementos

Figura 2: Carpeta Train del Dataset

Observe la distribución de imágenes de cada categoría en las carpetas del dataset. Es importante mantener en la medida de lo posible las proporciones recomendadas de 60 %/20 %/20 % para las secuencias de *train/test/validation* respectivamente. En este laboratorio se contaban con un dataset que mantenía la misma cantidad de imágenes para cada categoría, por lo cual es posible una división tan precisa y balanceada de los datos, sin embargo, esto no es siempre posible y suele suceder que se tiene más información de una clase que de otra, por lo cual en ese caso las proporciones se deben cuidar también, pero basándonos en la cantidad de elementos de clase. En nuestro caso, nuestro código está preparado para trabajar con 3 categorías o clases, por lo cual usted puede utilizar otro dataset de imágenes de otros animales y adaptar los aspectos necesarios dentro del mismo para que pueda usar sus propias imágenes.

1.2. Entrenamiento

Como siguiente paso iniciaremos la construcción del código fuente que permitirá el entrenamiento de nuestro modelo. Este código se puede trabajar en Jupyter Notebook o mediante un editor de texto para crear el script como un archivo de ejecución completa.

Para ello copie exactamente las siguientes líneas de código. Su archivo de código deberá estar guardado al mismo nivel que la carpeta con el dataset, de lo contrario deberá modificar la ruta propuesta en este laboratorio.

Esta sección del laboratorio fue desarrollada tomando en cuenta los ejemplos publicados en [1, 2, 3].

Para ello debemos abrir el siguiente archivo `CNN_Entrenamiento_Categorical_EDIT.py`. En este documento tome como referencia las marcas enumeradas con la siguiente estructura `# In[11]`:

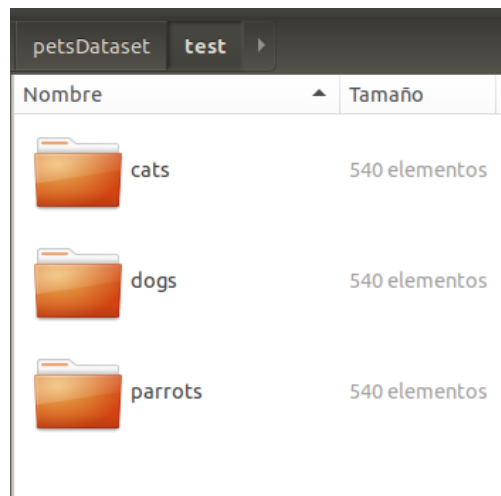
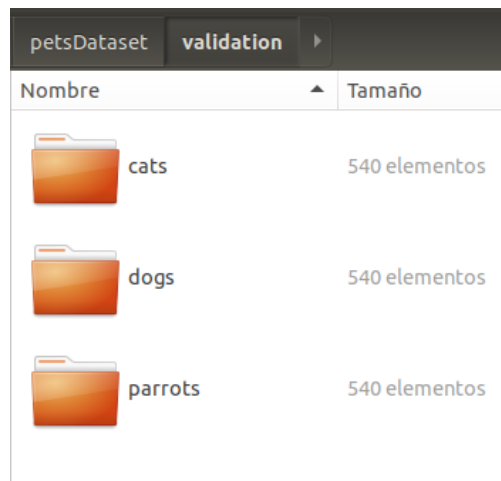


Figura 3: Carpeta Test del Dataset

- Como primera modificación se buscará la etiqueta **# In[9]**: y se agrega el siguiente código (ver Figura 5):
- En este fragmento de código se procede a definir la arquitectura de nuestra CNN y que será entrenada posteriormente. Se definen las siguientes capas:
 - Capas de convoluciones (Conv2D), con función de activación relu
 - Capas de *pooling* (MaxPooling2D)
 - Capa *Flatten*: esta capa se encarga de convertir las salidas de resultante de la parte de extracción de características en un vector de una sola dimensión.
 - Capa de *Dropout*: estas capas se encargan de desactivar un conjunto de neuronas para favorecer el proceso de aprendizaje del modelo.
 - Capas totalmente conectadas (*Dense*): también llamadas *fully connected* o perceptrones multicapa, estas capas se encargan de realizar la clasificación tomando como entrada las características extraídas por las capas de convolución. En este ejemplo se definen una capa Dense que produce una salida de 512 elementos. Dicha salida es la entrada de la 2 capa Dense, la cual produce una salida de 3 elementos, la cual consiste en la respuesta que produce la red neuronal. La activación softmax de esta capa de salida implica que el valor producido para cada categoría del dataset será menor a 1, tratándose como una distribución de probabilidad. Donde la suma de las salidas será 1,0.






petsDataset		validation ▶
Nombre		Tamaño
	cats	540 elementos
	dogs	540 elementos
	parrots	540 elementos

Figura 4: Carpeta Validation del Dataset

- En la definición de la primera capa se aprecia el tamaño que deben tener las imágenes de entrada a la red en este caso $150 \times 150 \times 3$. Lo cual nos dice que debemos trabajar con imágenes de 150 píxeles de alto y ancho, con los canales RGB. En cada capa de convolución se indica la cantidad de filtros que aplicará la capa, el tamaño de dichos filtros y la activación que procesa la salida de la capa.
- El siguiente paso consiste añadir el siguiente código a la etiqueta **# ln[11]** (ver Figura 6):
- Este fragmento de código define los parámetros a utilizar para el entrenamiento de nuestra CNN. Definimos la función de costo que se utilizará, en este caso debido a que se trata de un dataset con varias categorías, se utilizará *categorical_crossentropy*, se define el *accuracy* como métrica que será monitorizada durante el entrenamiento y se asigna como optimizador Adam con una tasa de aprendizaje (lr) de 0,0001, el cual se encarga ajustar los parámetros internos de la red, tomando en cuenta el valor obtenido por la función de *loss*.
- El siguiente paso será añadir el siguiente fragmento bajo la etiqueta **# ln[12]** (ver Figura 7):
- En este utilizamos la función de *ImageDataGenerator* de Keras para disminuir la resolución de cada uno de los conjuntos de imágenes.

```
# In[9]:
import tensorflow as tf
from tensorflow.keras import Model

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 3)))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.1))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(3, activation='softmax'))
```

Figura 5: Etiqueta 9

```
# In[11]:
from tensorflow.keras.optimizers import RMSprop, Adam
model.compile(optimizer=Adam(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics = ['acc'])
```

Figura 6: Etiqueta 11

- También se definen los generadores, este tipo de estructura de Python, actúa como un iterador, por lo cual, podemos recorrer los elementos que genera. En nuestro código los generadores serán utilizados en el proceso de entrenamiento para suplir las imágenes que se utilizarán. Cada generador se encarga de crear un sub-grupo con una cantidad de imágenes, esta cantidad está basada en el parámetro *batch_size*. Por lo cual, este generador creará grupos compuestos de 20 imágenes, con un tamaño de 150×150 . Se indica también que la manera de definir las clases será de forma *categorical* ya que se trata de un dataset con 3 categorías.
- Añadir el siguiente fragmento en la etiqueta **# In[14]** (ver Figura 8):
- En este fragmento se inicia formalmente el entrenamiento del modelo, y se guarda la información producida por el mismo en un objeto de nombre *history*. La primera sección del código se le dice al modelo cual es el generador que se utilizará para cargar las imágenes de entrenamiento, en este caso el *train_generator* que se definió en la etiqueta **# In [12]** (Figura 7). De igual manera se le asigna el conjunto de datos que se empleará para la validación *validation_generator*, el cual se definió en la ya mencionada etiqueta. El

```
# In[12]:
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator( rescale = 1.0/255. )
validation_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen = ImageDataGenerator( rescale = 1.0/255. )

train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='categorical',
                                                    target_size=(150, 150))

validation_generator = validation_datagen.flow_from_directory(validation_dir,
                                                            batch_size=20,
                                                            class_mode = 'categorical',
                                                            target_size = (150, 150))

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  batch_size=20,
                                                  class_mode = 'categorical',
                                                  target_size = (150, 150))
```

Figura 7: Etiqueta 12

```
# In[14]:
history = model.fit(
    train_generator,
    steps_per_epoch= steps_per_epoch,
    epochs=180,
    validation_data=validation_generator,
    validation_steps= validation_steps,
    verbose=2)

#Guardar el modelo entrenado
model.save('pets_categorical.h5')
```

Figura 8: Etiqueta 14

parámetro *epoch* expresa cuantas veces el algoritmo verá(estudiará) todas las imágenes del conjunto de entrenamiento, es decir en nuestro caso tenemos un conjunto de entrenamiento con 4860 imágenes cuando el modelo ha visto 1 vez cada una de las 4860 imágenes se cumple un *epoch* o época, y se inicia el nuevo ciclo de la *epoch* #2.

- El entrenamiento de un modelo requiere de muchas *epochs*, pero a más de estas no asegura un mejor aprendizaje por parte del modelo.
- El parámetro *steps_per_epoch*, se relaciona con la cantidad de imágenes disponibles, el entrenamiento suele realizarse mediante pequeños grupos de imágenes, debido a que sería muy difícil para un ordenador cargar todas las imágenes en memoria a la vez, por lo tanto, este parámetro indica la cantidad de grupos(*steps*) que se harán y por lo tanto la cantidad que se requieren para completar un *epoch*.
- De igual manera se le indica al modelo qué generador de datos se utilizará para la validación

del modelo, será con este conjunto de datos que se calcula la función de pérdida y se ajustan los parámetros del modelo.

- La ultima sección del código se utiliza el método `model.save()` para guardar el modelo recién creado para que se pueda utilizar en otros programas.
- De igual manera si observamos la etiqueta **# In[15]** (ver Figura 9) podemos apreciar un código que se utiliza para almacenar en un archivo `.json`, la lista de categorías identificadas por el sistema antes del entrenamiento.

```
# In[15]:
import json
print(train_generator.class_indices)

a_file = open("pets_indices.json", "w")
a_file = json.dump(train_generator.class_indices, a_file)

#print(model.class_indices)
history_dict = history.history
print(history_dict.keys())
```

Figura 9: Etiqueta 15

- Posteriormente debemos añadir el siguiente código en la etiqueta **# In[16]** (ver Figura 10):
- Este fragmento se encarga de visualizar la información contenida en el objeto `history`, creará las gráficas de *accuracy* y *loss*, que nos permitirán evaluar cómo se ha comportado el modelo durante cada *epoch*, es decir cuál es el *accuracy* y *loss* que se calculó en cada una lo cual nos permite tomar decisiones sobre la utilidad que pueda tener nuestro modelo basadas en el comportamiento durante el entrenamiento.
- Como última modificación se debe añadir el siguiente código, bajo la etiqueta **# In[17]** (Ver Figura 11).
- En este código utilizamos el generador de test (`test_generator`), o sea el objeto que genera nuestros sub-conjuntos de imágenes. En este caso el objetivo es verificar la capacidad de generalización que tiene el modelo, usando imágenes que nunca ha visto. Por lo cual, tendríamos una idea de que tanto el modelo puede comportarse en un mundo real con información nueva y cambiante.


```
# In[16]:
acc      = history.history[ 'acc' ]
val_acc  = history.history[ 'val_acc' ]
loss     = history.history[ 'loss' ]
val_loss = history.history[ 'val_loss' ]

epochs   = range(1,len(acc)+1,1) # obtener número de epochs del eje X

plt.plot ( epochs, acc, 'r--', label='Training acc' )
plt.plot ( epochs, val_acc, 'b', label='Validation acc' )
plt.title ('Training and Validation Accuracy')
plt.ylabel('acc')
plt.xlabel('epochs')

plt.legend()
plt.figure()

plt.plot ( epochs, loss, 'r--', label='Training loss' )
plt.plot ( epochs, val_loss, 'b', label='Validation loss' )
plt.title ('Training and Validation Loss' )
plt.ylabel('loss')
plt.xlabel('epochs')

plt.legend()
plt.figure()
```

Figura 10: Etiqueta 16

```
# In[17]:
test_steps = test_generator.n // batch_size
test_loss, test_acc = model.evaluate(test_generator, steps=test_steps)
print ("Test Accuracy:", test_acc)
```

Figura 11: Etiqueta 17

Entregables para la Sección

1. Los entregables de esta sección se solicitarán en la siguiente.

2. Código para hacer inferencia de un modelo entrenado

En la sección anterior se entrenó un modelo utilizando el algoritmo de CNN. Este producía un modelo con la capacidad de distinguir entre 3 tipos de animales, gatos, loros y perros.

En esta sección nos enfocaremos en la utilización del archivo que guarda el modelo generado en la sección anterior. Este archivo se cargará a nuestro *script* a través del comando `load_model()`, este abrirá el archivo que guardamos en la sección anterior. Su código es mucho más sencillo ya que solo debemos procurar que la ruta a la imagen que se desea evaluar este escrita de manera correcta.

De igual manera se carga también el archivo con los índices/categorías que se encuentran en el dataset, este fue el archivo que se generó en el proceso de entrenamiento.

Para ello copie el siguiente código y guárdelo con el nombre `CNN_Test_Categorical.py`. Debe modificar este código señalando la ruta de una imagen de perro o gato o loro que este disponible en su computadora. De igual manera debe verificar que el nombre asignado al modelo guardado sea el mismo que se encuentra almacenado en la carpeta donde están los códigos. Este código prueba el modelo cargando directamente una imagen y también utilizando una interfaz de Gradio para buscar la imagen en los archivos del computador.

1. Copie el siguiente código en una Notebook o en un archivo de Python

```
# Este código emplea un modelo CNN ya entrenado para realizar inferencias
fuera del código de entrenamiento

import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import load_model

path='../gato5.jpg'
img=image.load_img(path, target_size=(150, 150))

x=image.img_to_array(img)
imageFile=np.expand_dims(x, axis=0)

import json
a_file = open("pets_indices.json", "r")
class_indices = json.load(a_file)
model = load_model('pets_categorical.h5')

classes = model.predict(imageFile)
print(classes)
clas =np.where(classes[0] == max(classes[0]))
categoria = clas[0][0]
print(categoria)
print(list(class_indices.keys())[list(class_indices.values()).index(categoria)])
```

```

import matplotlib.image as mpimg
import matplotlib.pyplot as plt
plt.imshow(img)
plt.show()

def clasificar(imagePath):
    global model
    img=image.load_img(imagePath, target_size=(150, 150))
    x=image.img_to_array(img)
    imgFile=np.expand_dims(x, axis=0)
    classes = model.predict(imgFile)
    clas =np.where(classes[0] == max(classes[0]))
    categoria = clas[0][0]
    labelCategoria= (list(class_indices.keys())[list(class_indices.values()).index(categoria)])
    labelResultado = f'En la imagen aparece un {labelCategoria}.'
    return labelResultado

import gradio as gr

gr.Interface(
    fn = clasificar, # funcion
    inputs = gr.Image(type="filepath"), # tipo de entrada
    outputs = "textbox"
).launch()

```

3. Cuestionario

1. Luego de la ejecución del código fuente aproximadamente ¿qué tiempo tomo para producir y guardar el modelo?
2. Presente una captura de la consola o jupyter notebook luego de la ejecución de este código, con las gráficas de *accuracy* producidas.
3. ¿Cuál fue el *accuracy* y el *loss* obtenido por este modelo para *train*, *validation* y *test*? Puede presentar su respuesta completando la siguiente tabla.

SubGrupo	Accuracy	Loss
Train		
Validation		
Test		

4. Una vez entrenado el modelo, pruebe dicho modelo con el código de la Parte 2 del laboratorio, utilice 5 imágenes diferentes y que no estén presentes en el dataset y evalúe si el

modelo es capaz de identificar correctamente el animal de la foto. Presente la evidencia de la ejecución de este código con los 5 elementos diferentes así como también la clase producida por el modelo.

5. Modifique el código de entrenamiento, utilizando ahora un `epochs=70`. ¿Aproximadamente cuánto tiempo tarda en el ejecutar?* (**Ver observación en la siguiente pagina antes de entrenar**).
6. ¿Cuál fue el *accuracy* y el *loss* obtenidos por este modelo para *train*, *validation* y *test*, luego de esta modificación? Puede presentar su respuesta completando la siguiente tabla.

SubGrupo	Accuracy	Loss
Train		
Validation		
Test		

7. Una vez entrenado el modelo utilizando 70 épocas, pruebe dicho modelo con el código de la Parte 2 del laboratorio, utilice 5 imágenes diferentes y que no estén presentes en el dataset y evalúe si el modelo es capaz de identificar correctamente el animal de la foto. Presente la evidencia de la ejecución de este código con los 5 elementos diferentes así como también la clase producida por el modelo, puede usar las mismas imágenes que utilizó para la Pregunta 4.
8. Luego de la ejecución de ambos modelos. ¿Cuál obtuvo mejores resultados cuando se le mostraron sus imágenes?

*Observación

La cantidad de épocas sugeridas para entrenar la CNN dependerá de la capacidad de su equipo. Se recomienda como primer paso utilizar solo 2 épocas y estimar el tiempo requerido para dicho entrenamiento. Para completar la información solicitada en las tablas, se recomienda definir las épocas tomando en cuenta este tiempo. Por lo cual, para la primera tabla (Pregunta 3) se recomienda un numero bajo de épocas (como mínimo 5) y para la segunda tabla (Pregunta 5 y 6) se recomienda un número mayor (tomando en cuenta la capacidad de su computadora). Para apreciar las diferencias que tiene en el entrenamiento el número de épocas. Indique en cada tabla la cantidad de épocas utilizadas.

Entregables para la Sección

1. Responda el cuestionario del laboratorio

2. Aporte las evidencias de la prueba del modelo, utilizando sus propias imágenes.
3. Adjunte una captura de pantalla donde se aprecie el resultado final después de terminar todas las épocas utilizadas para entrenar su modelo.

4. Utilización de Modelos Pre-Entrenados por terceras personas

En la sección anterior se mostró el código que permitía el entrenamiento de una CNN utilizando nuestras imágenes o nuestro propio dataset, de igual manera se expuso la manera en la cual se puede utilizar dicho modelo entrenado.

En esta sección nos enfocaremos en la utilización de modelos de CNN que han sido previamente entrenados, por terceras personas u organizaciones y que han sido entrenados utilizando un dataset de una gran dimensión, por lo tanto, son modelos optimizados y mejor preparados para la generalización.

En estos casos el uso de estos modelos significa que ya no es necesario llevar un proceso de entrenamiento de nuestro modelo, ahorrándonos tiempo y esfuerzo en la recolección y preparación de los datos.

Estos modelos son entrenados en un dataset específico que tendrá un cantidad determinada de categorías, por lo tanto, si deseamos utilizar uno de estos modelos, lo prudente es consultar si las clases o categorías que puede reconocer, incluyen los objetos, lugares o elementos que necesitamos reconocer en nuestro enfoque.

OpenCV incluye dentro de sus módulos el paquete DNN, el cual cuenta con la capacidad de utilizar modelos pre-entrenados de diversos *frameworks* de Deep Learning. Entre estos podemos mencionar Caffe¹, TensorFlow², Torch/PyTorch³, Darknet⁴ y ONNX⁵.

Generalmente para utilizar estos modelos se debe contar con al menos 3 archivos en los cuales se almacenan los pesos de la red entrenada, la arquitectura o diseño de la red y una lista que indique cuales son las categorías que es capaz de reconocer el modelo, esta lista debe estar en orden para lograr que se correspondan con la salida de la red. La lista de categorías dependerá de manera exclusiva del dataset utilizado para entrenar el modelo. Los archivos de pesos suelen tener un tamaño mayor, por lo cual, para esta sección estos archivos se encuentran disponibles en el enlace ⁶.

¹<https://caffe.berkeleyvision.org/>

²<https://www.tensorflow.org/>

³<https://torch.ch/>

⁴<https://pjreddie.com/darknet/yolov1/>

⁵<https://onnx.ai/>

⁶Link Pesos

4.1. CNN - RESNET50

Como primer ejemplo utilizaremos el modelo con la arquitectura RESNET50 [4], la cual como su nombre lo indica posee 50 capas de procesamiento interno. Los procesos de carga de los archivos del modelo serán similares para modelos pre-entrenados de cualquiera de los *frameworks* que se han mencionado anteriormente y que están soportados por es el módulo DNN de OpenCV.

En este caso, el modelo que utilizaremos fue creado en el *framework* Caffe[5]. Este modelo ha sido entrenado con el dataset ImageNet⁷ [6], por lo cual, puede reconocer 1000 categorías diferentes, entre las cuales se pueden mencionar, grúa, fijador, saxofones, autos, edificios, entre otros.

Para este ejemplo se utilizará el código: `image_classification_opencv_resnet_50_caffe.py` disponible en la carpeta `resnet_50`, de igual manera en esta carpeta se encuentran los 3 archivos necesarios para la carga del modelo:

- **ResNet-50-deploy.prototxt**: contiene el diseño de la red neuronal, ResNet50 en ese caso.
- **ResNet-50-model.caffemodel**: son los pesos o ponderaciones de los parámetros para la arquitectura de red seleccionada. Se debe descargar desde este enlace suministrado.
- **synset_words.txt**: archivo con los nombres de las categorías que puede reconocer el modelo.

Este fragmento de código muestra como es el proceso de carga de los elementos para que se pueda crear una instancia del modelo entrenado en nuestro programa.

```
# Cargar el nombre de las clases del dataset:
rows = open('synset_words.txt').read().strip().split('\n')
classes = [r[r.find(' ') + 1:].split(',')[0] for r in rows]

# Cargar el modelo serializado de caffe:
net = cv2.dnn.readNetFromCaffe("ResNet-50-deploy.prototxt", "ResNet-50-model.caffemodel")
```

El siguiente bloque se encarga de crear el blob que contiene la imagen que será analizada por el modelo, con la función `blobFromImage()` se realiza un preprocesado de la imagen que se enviará a la CNN. Con esta se pueden definir las dimensiones, sustraer la imagen media (para normalizar los valores de los píxeles), cambiar el orden de los canales y escalar la imagen. Un blob se puede considerar como un grupo de imágenes que se pasan a la vez a nuestra red para que las analice y produzca una respuesta. En este caso se crea un blob de solo una imagen para probar el código.

De igual manera el blob creado se envía a la red a través de la función `setInput()` y se obtienen las predicciones para la/las imágenes enviadas el método `net.forward()`.

```
# Crear el blob con un tamaño (224,224), valores mean subtraction (104, 117, 123)
blob = cv2.dnn.blobFromImage(image, 1, (224, 224), (104, 117, 123))
print(blob.shape)
```

⁷<http://www.image-net.org/>


```
# Enviar el blob a la red, realizar la inferencia y obtener la salida
net.setInput(blob)
preds = net.forward()
```

El siguiente paso consiste en obtener las etiquetas de las predicciones con mayor valor de probabilidad. Se debe recordar que la red producirá una salida que consiste en un vector de n dimensiones, n se corresponderá con la cantidad de categorías que es capaz de identificar la red, por lo cual para modelos entrenados con ImageNet se tiene un $n = 1000$. En consecuencia el vector de salida tendrá 1000 dimensiones y cada dimensión representa un valor de probabilidad para cada una de las categorías del dataset. Este vector se presenta arreglado en el orden que se presentan las categorías en la lista de las etiquetas del modelo.

El valor de cada una de las dimensiones estará en un rango de $0 \rightarrow 1$ y la sumatoria de todos estos valores producirá un valor de 1,0, esto debido a la capa de Softmax que se encuentra al final de la CNN.

En este caso se utiliza el siguiente fragmento para ordenarlos de mayor a menor y tomar solo los 10 primeros valores. Obtenemos los índices los cuales serán utilizados con la lista de categorías del dataset, para obtener la etiqueta textual que ha sido identificada en la imagen de entrada.

```
# Obtener los 10 índices con las mayores probabilidades
# De esta manera, el índice con la probabilidad más alta será el primero en la lista
indexes = np.argsort(preds[0])[:, -1] [:10]
```

Como siguiente paso se procede a imprimir la clase y probabilidad identificada en la imagen de entrada

```
# Mostrar la imagen, la clase y probabilidad de mayor valor
text = "label: {} \nprobability: {:.2f}%".format(classes[indexes[0]], preds[0][indexes[0]] * 100)
y0, dy = 30, 30
for i, line in enumerate(text.split('\n')):
    y = y0 + i * dy
    cv2.putText(image, line, (5, y), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)

# Imprimir las 10 primeras predicciones:
for (index, idx) in enumerate(indexes):
    print("{} label: {}, probability: {:.10}".format(index + 1, classes[idx], preds[0][idx]))
```

Para la ejecución de este código debe cambiar la imagen que enviará a la red.

4.2. RCNN - YOLO

En esta sección utilizaremos un modelo pre-entrenado de una RCNN, estas arquitecturas se especializan en la detección y clasificación de objetos dentro de las imágenes. El modelo utilizado fue creado en **Darknet** con la arquitectura Yolo [7], la cual en la actualidad se considera como el estado del arte en la detección de objetos en imágenes.

Este modelo fue entrenado usando el dataset COCO⁸ [8] y tiene la capacidad de reconocer 80 categorías entre animales, equipo deportivo, utensilios, electrodomésticos entre otros. La lista completa la puede encontrar en el archivo *coco.names* dentro de la carpeta *yolo*.

En esta ocasión se utiliza el código `object_detection_opencv_yolo_darknet.py`, disponible en la carpeta *yolo*.

Para crear la instancia de este modelo se requieren de igual forma 3 archivos

- **yolov3.cfg**: archivo que contiene la arquitectura de la red, disponible en la carpeta *yolo*
- **yolov3.weights**: archivo con los pesos del modelo entrenado, este archivo se debe descargar del enlace suministrado debido al tamaño del archivo *250mb*.
- **coco.names**: lista con los nombres de las categorías del dataset disponible dentro de la carpeta *yolo*

Este modelo produce para cada imagen una lista de etiquetas y un conjunto de coordenadas. Para cada categoría/objeto identificado en la imagen se genera un *bounding box* que señala la ubicación del objeto dentro de la imagen. En este caso la red no produce un valor para cada una de las categorías del dataset, sino que por imagen produce una lista de los objetos detectados.

El siguiente código permite la creación de una instancia del modelo entrenado con Yolo en nuestro programa.

```
# load the COCO class labels:
class_names = open("coco.names").read().strip().split("\n")

# Load the serialized caffe model from disk:
net = cv2.dnn.readNetFromDarknet("yolov3.cfg", "yolov3.weights")
```

De manera similar a la sección anterior debemos crear un blob para la imagen que deseamos analizar.

Para Yolo se hace necesario un proceso de filtrado de las regiones detectadas en la imagen, este filtrado se realiza utilizando un valor de *confidence* producido por la red. Para ello se recorren las capas de salida de la red y se analiza cada detección producida por estas capas, si el valor de *confidence* para cada detección es mayor a un umbral, entonces esta se toma en cuenta para la siguiente etapa del proceso de detección.

⁸<https://cocodataset.org/#home>

```

# loop over each of the layer outputs
for output in layerOutputs:
    # loop over each of the detections
    for detection in output:
        # Get class ID and confidence of the current detection:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]

        # Filter out weak predictions:
        if confidence > 0.25:
            # Scale the bounding box coordinates
            # (center, width, height) using the dimensions of the original image:
            box = detection[0:4] * np.array([W, H, W, H])
            (centerX, centerY, width, height) = box.astype("int")

            # Calculate the top-left corner of the bounding box:
            x = int(centerX - (width / 2))
            y = int(centerY - (height / 2))

            # Update the information we have for each detection:
            boxes.append([x, y, int(width), int(height)])
            confidences.append(float(confidence))
            class_ids.append(class_id)

```

Una vez filtrados los bounding boxes con un valor de *confidence* alto, se realiza un proceso de Non-maxima Suppression, con el cual se busca eliminar las regiones que se solapan y las regiones que se consideran predicciones débiles, debido a que no se solapan con otras regiones.

```

# We can apply non-maxima suppression (eliminate weak and overlapping bounding boxes):
indices = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.3)

```

Como ultimo paso del proceso, se pasa a dibujar las regiones (bounding boxes) que han pasado el proceso de filtrado con el Non-maxima suppression. Siendo estos los resultados que arroja la red y que se marcan en la imagen.

```

# Show the results (if any object is detected after non-maxima suppression):
if len(indices) > 0:
    for i in indices.flatten():
        # Extract the (previously recalculated) bounding box coordinates:
        (x, y) = (boxes[i][0], boxes[i][1])
        (w, h) = (boxes[i][2], boxes[i][3])

        # Draw label and confidence:
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
        label = "{}: {:.4f}".format(class_names[class_ids[i]], confidences[i])
        labelSize, baseLine = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 1, 2)
        y = max(y, labelSize[1])
        cv2.rectangle( image,
                        (x, y - labelSize[1]),

```

```

(x + labelSize[0],
y + 0),
(0, 255, 0),
cv2.FILLED)
cv2.putText(image, label, (x, y), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 2)

```

Luego de la ejecución de código, la salida de la red mostrará una imagen como se muestra en la Figura 12.

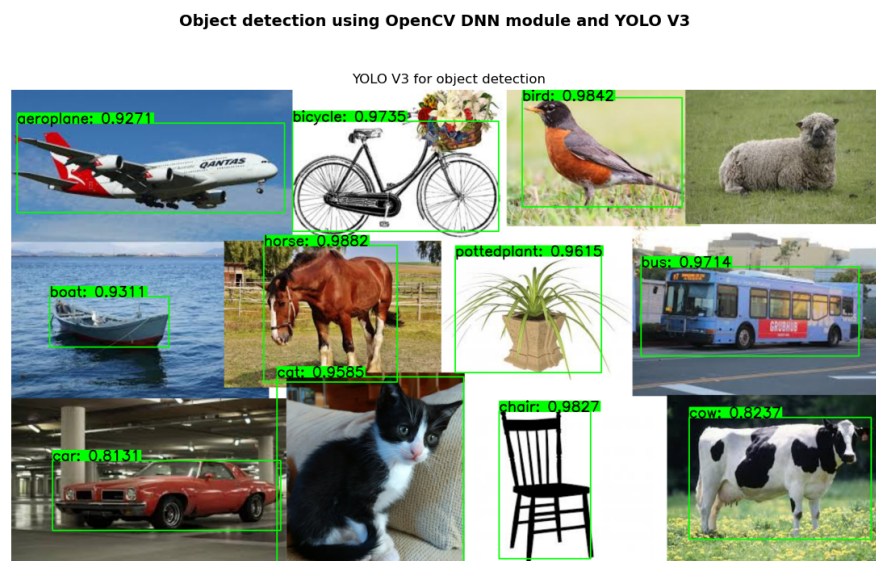


Figura 12: Salida Producida por Yolo

Entregables para la Sección

1. Para esta sección se propone utilizar el código de ResNet50 como base y construir un clasificador en tiempo real utilizando la cámara (o vídeo en su defecto) y que muestre la etiqueta y la probabilidad del objeto que se observa. Aporte capturas de pantalla de la ejecución de su clasificador.
2. Para probar el código de YOLO confeccione 3 collages de imágenes diferentes con entre 6 – 10 imágenes de posibles objetos que puedan ser detectados por la red.
3. Utilizando el código proporcionado para YOLO como base, modifíquelo para que envíe cada collage mediante una interfaz de Gradio y se muestre el resultado. En este caso su interfaz de Gradio tendrá como entrada la ruta de la imagen (similar al problema de inferencia de la CNN) y su salida será también una imagen, por lo cual, deberá ajustar el resultado a esta configuración mencionada. Presente las capturas del resultado de esta sección para cada collage.

Referencias

- [1] A. Fernández Villán, *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7. Mastering Open Source Computer Vision four with Python*. Birmingham: Packt Publishing, 2019. [Online]. Available: <https://cds.cern.ch/record/2674578>
- [2] F. Chollet, *Deep Learning with Python*. Manning Publications Company, 2017. [Online]. Available: <https://books.google.com.pa/books?id=Yo3CAQAACAAJ>
- [3] J. Torres, *Python deep learning: Introduccion practica con Keras y TensorFlow 2*. Marcombo, 2020. [Online]. Available: <https://books.google.com.pa/books?id=ooqqzQEACAAJ>
- [4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: <http://arxiv.org/abs/1512.03385>
- [5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.
- [6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, “Imagenet large scale visual recognition challenge,” *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: <http://arxiv.org/abs/1409.0575>

- [7] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: <http://arxiv.org/abs/1804.02767>
- [8] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: <http://arxiv.org/abs/1405.0312>