

DEEP LEARNING

PROYECTO FINAL

Rafael Rodríguez Vázquez



RAFAEL RODRÍGUEZ VÁZQUEZ

RESPONSABLE DEL DESARROLLO

Fecha: 01, 05, 2024



DESARROLLO

Objetivo principal e hipótesis:

Partimos de la base de que un usuario está conduciendo y se producen cambios meteorológicos, a tiempo real, en el entorno donde se encuentra su vehículo autónomo.

La idea principal es establecer un sistema de clasificación de imágenes, con contenido meteorológico (datos), para que, a través de un modelo de red neuronal convolucional (CNN), previamente entrenado, nos aporte una predicción sobre qué cambio meteorológico se ha producido.

De la detección de estos fenómenos se encargarán unos sensores en el mismo vehículo. Los sensores enviarán esta información a una api y otros sistemas dentro del vehículo y este le ayudará al usuario en la conducción, a través de alertas u otras ayudas.

La ayuda puede establecerse ofreciendo recomendaciones o procesos automáticos como:




- Encender las luces.
 - Poner cadenas.
 - Reducir la velocidad.
 - Activarse la calefacción automáticamente.
 - Etc.
-
- **Objetivo:** establecer un sistema de clasificación que nos prediga correctamente o con mayor probabilidad, que fenómeno se está produciendo en el exterior; al mismo tiempo que esta predicción pueda ser ejecutado por una API tipo REST.

Proceso de pre-procesamiento de datos y construcción de red neuronal (CNN):

1. Pre-procesamiento de datos

Hemos navegado en la página web de Kaggle hasta encontrar un dataset adecuado (link en el archivo de jupyter).

Este dataset tiene un tamaño de 700 MB y viene comprimido en un archivo Zip llamado archive.zip, el cual ha sido descargado en nuestra carpeta donde desarrollaremos el proyecto.

Este equipo > OS (C:) > Usuarios > Kocxi > AnacondaProjects > Deep Learning > Proyecto final >Codigo				
Nombre	Fecha de modificación	Tipo	Tamaño	
 .ipynb_checkpoints	23/04/2024 0:52	Carpeta de archivos		
 archive.zip	18/04/2024 1:42	Archivo WinRAR ZIP	711.710 KB	
 Proyecto_DL_Rafa_Rodriguez_Vazquez.ip...	23/04/2024 17:44	Archivo IPYNB	6.931 KB	

Una vez que tenemos nuestro dataset descargado, lo descomprimos a través de código, y extraemos todos los datos en la carpeta del proyecto.

```
import os          # importamos el modulo os para poder operar con los directorios desde jupyter directamente
import zipfile     # importamos el modulo zipfile, para abrir y cargar los datos descargados de kaggle e

# Generamos un bucle que nos permita extraer los archivos de nuestro zip y en caso de que hayan sido e

local_zip='./archive.zip' # establecemos una variable que tendra en valor nuestro fichero zip

if not os.path.exists('./dew fogsmog frost glaze hail lightning rain rainbow rime sandstorm snow'):
    zip_ref = zipfile.ZipFile(local_zip, 'r')
    zip_ref.extractall('.')
    zip_ref.close()
    print("Archivos extraídos correctamente")
else:
    print("Los archivos ya han sido extraídos correctamente")
```

Archivos extraídos correctamente

Este equipo > OS (C:) > Usuarios > Kocxi > AnacondaProjects > Deep Learning > Proyecto final >Codigo

Nombre	Fecha de modificación	Tipo	Tamaño
.ipynb_checkpoints	23/04/2024 0:52	Carpeta de archivos	
dew	23/04/2024 17:58	Carpeta de archivos	
fogsmog	23/04/2024 17:58	Carpeta de archivos	
frost	23/04/2024 17:58	Carpeta de archivos	
glaze	23/04/2024 17:58	Carpeta de archivos	
hail	23/04/2024 17:58	Carpeta de archivos	
lightning	23/04/2024 17:59	Carpeta de archivos	
rain	23/04/2024 17:59	Carpeta de archivos	
rainbow	23/04/2024 17:59	Carpeta de archivos	
rime	23/04/2024 17:59	Carpeta de archivos	
sandstorm	23/04/2024 17:59	Carpeta de archivos	
snow	23/04/2024 17:59	Carpeta de archivos	
archive.zip	18/04/2024 1:42	Archivo WinRAR ZIP	711.710 KB
Proyecto_DL_Rafa_Rodriguez_Vazquez.ip...	23/04/2024 18:00	Archivo IPYNB	25 KB

Con nuestros datos extraídos correctamente procedemos a comprobar su contenido

```
# Mostramos la informacion de nuestros datos
print(
    "Nº de ejemplos según directorio:\n"
    "-dew:", len(dew_list),
    "\n-fogsmog:", len(fogsmog_list),
    "\n-frost:", len(frost_list),
    "\n-glaze:", len(glaze_list),
    "\n-hail:", len(hail_list),
    "\n-lightning:", len(lightning_list),
    "\n-rain:", len(rain_list),
    "\n-rainbow:", len(rainbow_list),
    "\n-rime:", len(rime_list),
    "\n-sandstorm:", len(sandstorm_list),
    "\n-snow:", len(snow_list)
)
```

```
Nº de ejemplos según directorio:
-dew: 700
-fogsmog: 700
-frost: 700
-glaze: 700
-hail: 700
-lightning: 700
-rain: 700
-rainbow: 700
-rime: 700
-sandstorm: 700
-snow: 700
```

Una vez hacemos una comprobación genérica de nuestros datos, comenzamos con la limpieza y la preparación de cara al modelo.

Eliminamos las carpetas de datos que no nos son relevantes por similitud o por ser innecesarias como son: glaze, rainbow y rime.

```
import shutil # este modulo nos ayudará gracias a su facilidad de manipulacion de directorios

# Establecemos variables que nos son de interés para el proceso de eliminar las carpetas
dir_eliminar = [glaze, rainbow, rime]
ruta_base = './'












# Generamos un bucle anidado que nos eliminará los directorios incluidos en dir_eliminar
for directorio in dir_eliminar:
    ruta_dir = os.path.join(ruta_base, directorio)

    if os.path.exists(ruta_dir):
        shutil.rmtree(ruta_dir)
        print("Directorio/s eliminados correctamente")
    else:
        print("Directorio/s a eliminar no existentes")

# Nuestros datos estan preparados ahora para realizar correctamente la division de los conjuntos de train, validation y test.

Directorio/s eliminados correctamente
Directorio/s eliminados correctamente
Directorio/s eliminados correctamente
```

Este equipo > OS (C:) > Usuarios > Kocxi > AnacondaProjects > Deep Learning > Proyecto final > Codigo

Nombre	Fecha de modificación	Tipo	Tamaño
 .ipynb_checkpoints	23/04/2024 0:52	Carpeta de archivos	
 dew	23/04/2024 17:58	Carpeta de archivos	
 fogsmog	23/04/2024 17:58	Carpeta de archivos	
 frost	23/04/2024 17:58	Carpeta de archivos	
 hail	23/04/2024 17:58	Carpeta de archivos	
 lightning	23/04/2024 17:59	Carpeta de archivos	
 rain	23/04/2024 17:59	Carpeta de archivos	
 sandstorm	23/04/2024 17:59	Carpeta de archivos	
 snow	23/04/2024 17:59	Carpeta de archivos	
 archive.zip	18/04/2024 1:42	Archivo WinRAR ZIP	711.710 KB
 Proyecto_DL_Rafa_Rodriguez_Vazquez.ip...	23/04/2024 18:06	Archivo IPYNB	26 KB

Una vez terminada la limpieza empezamos con la preparación de los conjuntos de datos (entrenamiento, test y validación), mediante código; y a su vez con la creación de sus directorios, dentro de la carpeta del proyecto, para que recojan estos datos.

```
train_snow = snow_list[:500]
val_snow = snow_list[500:600]
test_snow = snow_list[600:700]

print(f"Nº de ejemplos en train:\ndew: {len(train_dew)} ~ fogsmog: {len(train_fogsmog)} ~ frost: {len(train_frost)} ~ hail:{
print(f"\nNº de ejemplos en validation:\ndew: {len(val_dew)} ~ fogsmog: {len(val_fogsmog)} ~ frost: {len(val_frost)} ~ hail:{
print(f"\nNº de ejemplos en test:\ndew: {len(test_dew)} ~ fogsmog: {len(test_fogsmog)} ~ frost: {len(test_frost)} ~ hail:{len
<
```

Nº de ejemplos en train:
dew: 500 ~ fogsmog: 500 ~ frost: 500 ~ hail:500 ~ lightning: 500 ~ rain: 500 ~ sandstorm: 500 ~ snow: 500

Nº de ejemplos en validation:
dew: 100 ~ fogsmog: 100 ~ frost: 100 ~ hail:100 ~ lightning: 100 ~ rain: 100 ~ sandstorm: 100 ~ snow: 100

Nº de ejemplos en test:
dew: 100 ~ fogsmog: 100 ~ frost: 100 ~ hail:100 ~ lightning: 100 ~ rain: 100 ~ sandstorm: 100 ~ snow: 100

```
# creamos rutas hacia nuestros directorios
source = './'
train_dir = './train'
validation_dir = './validation'
test_dir = './test'

# creamos las carpetas de train, validation y test
if not os.path.exists(train_dir) or not os.path.exists(validation_dir) or not os.path.exists(test_dir):
    os.makedirs(train_dir, exist_ok=True)
    os.makedirs(validation_dir, exist_ok=True)
    os.makedirs(test_dir, exist_ok=True)
    print("Carpetas creadas correctamente")
else:
    print("Las carpetas ya se encuentran creadas")

# definimos nuestras etiquetas
categorias = ['dew', 'fogsmog', 'frost', 'hail', 'lightning', 'rain', 'sandstorm', 'snow' ]

for categoria in categorias:
    # creamos subcarpetas en train, validation y test para cada clase
    train_categoria = os.path.join(train_dir, categoria)
    validation_categoria = os.path.join(validation_dir, categoria)
    test_categoria = os.path.join(test_dir, categoria)

    for directory in [train_categoria, validation_categoria, test_categoria]:
        os.makedirs(directory, exist_ok=True)

# movemos las imágenes de la clase a las subcarpetas correspondientes en las carpetas de train, validation y test
for i, split in enumerate(['train', 'val', 'test']):
    fnames = globals()[f'{split}_{categoria}']
    dst = os.path.join([train_categoria, validation_categoria, test_categoria][i])

    for fname in fnames:
        src = os.path.join(source, categoria, fname)
        shutil.copyfile(src, os.path.join(dst, fname))
```

Carpetas creadas correctamente

Este equipo > OS (C:) > Usuarios > Kocxi > AnacondaProjects > Deep Learning > Proyecto final >Codigo

Nombre	Fecha de modificación	Tipo	Tamaño
.ipynb_checkpoints	23/04/2024 0:52	Carpeta de archivos	
dew	23/04/2024 17:58	Carpeta de archivos	
fogsmog	23/04/2024 17:58	Carpeta de archivos	
frost	23/04/2024 17:58	Carpeta de archivos	
hail	23/04/2024 17:58	Carpeta de archivos	
lightning	23/04/2024 17:59	Carpeta de archivos	
rain	23/04/2024 17:59	Carpeta de archivos	
sandstorm	23/04/2024 17:59	Carpeta de archivos	
snow	23/04/2024 17:59	Carpeta de archivos	
test	23/04/2024 18:20	Carpeta de archivos	
train	23/04/2024 18:20	Carpeta de archivos	
validation	23/04/2024 18:20	Carpeta de archivos	
archive.zip	18/04/2024 1:42	Archivo WinRAR ZIP	711.710 KB
Proyecto_DL_Rafa_Rodriguez_Vazquez.ip...	23/04/2024 18:20	Archivo IPYNB	27 KB

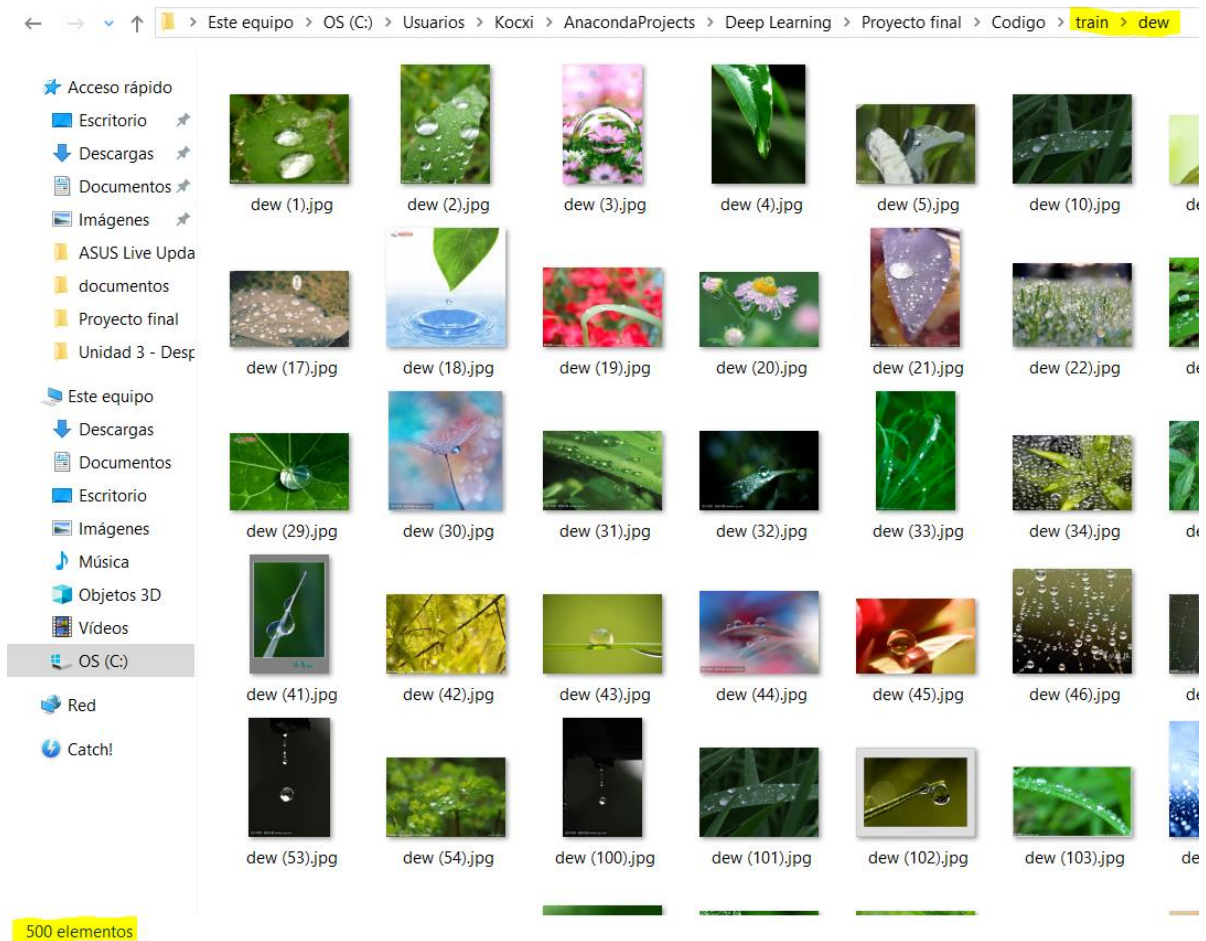
Con esto tendríamos las divisiones establecidas en carpetas (train, test y validation), divididas del siguiente modo:

- Train: 500 imágenes por cada clase; tenemos 8 clases.
- Test: 100 imágenes por cada clase.
- Validation: 100 imágenes por clase.

Mostraremos como ejemplo la carpeta train:

Este equipo > OS (C:) > Usuarios > Kocxi > AnacondaProjects > Deep Learning > Proyecto final >Codigo > train

Nombre	Fecha de modificación	Tipo	Tamaño
dew	23/04/2024 18:20	Carpeta de archivos	
fogsmog	23/04/2024 18:20	Carpeta de archivos	
frost	23/04/2024 18:20	Carpeta de archivos	
hail	23/04/2024 18:20	Carpeta de archivos	
lightning	23/04/2024 18:20	Carpeta de archivos	
rain	23/04/2024 18:20	Carpeta de archivos	
sandstorm	23/04/2024 18:20	Carpeta de archivos	
snow	23/04/2024 18:20	Carpeta de archivos	



Una vez tenemos nuestros conjuntos de datos divididos correctamente, continuamos con una pequeña visualización del conjunto total de datos (4 ejemplos por clase).

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# parámetros utilizados en nuestro gráfico (cantidad de muestras)
nrows = 16
ncols = 16

# índice para generar la iteración sobre imágenes
pic_index = 0

# Configuración de la librería matplotlib para que represente en una configuración de 4x4.
fig = plt.gcf()
fig.set_size_inches(ncols * 4, nrows * 4)

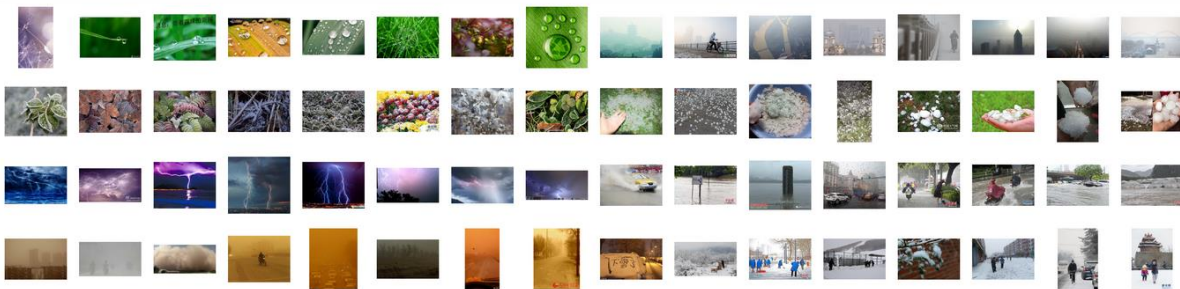
pic_index += 32
next_dew_pix = [os.path.join(dew, fname)
                for fname in train_dew[pic_index-8:pic_index]]
next_fogsmog_pix = [os.path.join(fogsmog, fname)
                    for fname in train_fogsmog[pic_index-8:pic_index]]
next_frost_pix = [os.path.join(frost, fname)
                  for fname in train_frost[pic_index-8:pic_index]]
next_hail_pix = [os.path.join(hail, fname)
                 for fname in train_hail[pic_index-8:pic_index]]
next_lightning_pix = [os.path.join(lightning, fname)
                      for fname in train_lightning[pic_index-8:pic_index]]
next_rain_pix = [os.path.join(rain, fname)
                 for fname in train_rain[pic_index-8:pic_index]]
next_sandstorm_pix = [os.path.join(sandstorm, fname)
                      for fname in train_sandstorm[pic_index-8:pic_index]]
next_snow_pix = [os.path.join(snow, fname)
                  for fname in train_snow[pic_index-8:pic_index]]

for i, img_path in enumerate(next_dew_pix+next_fogsmog_pix+next_frost_pix+next_hail_pix+
                              next_lightning_pix+next_rain_pix+next_sandstorm_pix+next_snow_pix):

    # configuramos para que los índices del subplot empiecen en 1
    sp = plt.subplot(nrows, ncols, i + 1)
    sp.axis('Off')

    img = mpimg.imread(img_path)
    plt.imshow(img)

plt.show()
```



De este modo concluye todo el proceso de pre-procesamiento de los datos descargados y nos encaminamos a la formación del modelo.

2. Construcción de la red neuronal (CNN)

Con nuestros datos listos, procedemos a construir nuestra red de neuronas profundas.

Antes de entrar con la creación de las capas, realizamos una augmentación de datos sobre nuestros datos, para obtener una mayor variabilidad y cantidad de ellos.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# comenzamos la augmentación para los conjuntos de train, validation y test

# train
train_datagen = ImageDataGenerator(
    rescale = 1./255,      # escalamos el valor del pixel entre 0 y 1
    rotation_range = 30,   # le damos un ángulo de rotación a cada imagen de 30°
    width_shift_range=0.2, # desplazamos aleatoriamente las imágenes un 20% horizontalmente
    height_shift_range=0.2, # desplazamos aleatoriamente las imágenes un 20% verticalmente
    shear_range=0.2,      # aplicamos cizallamiento en las imágenes
    zoom_range=0.2,       # aplicamos zoom en las imágenes
    horizontal_flip=True,  # volteamos aleatoriamente las imágenes horizontalmente (de derecha a izquierda y viceversa)
    fill_mode='nearest'   # rellena píxeles nuevos creados tras la aplicación de rotaciones y otros.
)

# validation
val_datagen = ImageDataGenerator(
    rescale = 1./255
)

# test
test_datagen = ImageDataGenerator(
    rescale = 1./255
)

# para continuar con el proceso de augmentación de datos, vamos a crear nuestros generadores de imagen utilizando los directorios
# creados en el proceso de transformación de los datos en celdas anteriores

train_generator = train_datagen.flow_from_directory(
    train_dir,          # directorio train creado anteriormente
    target_size = (150,150), # redimension de imágenes de entrada a 150x150 píxeles
    batch_size = 75,     # nº de imágenes que se cargaran y procesaran
    color_mode='rgb',    # modo de color de las imágenes
    class_mode='categorical' # tipo de etiquetas que se generaran, en este caso en formato one-hot con valores (0 ó 1)
)

val_generator = val_datagen.flow_from_directory(
    validation_dir,
    target_size = (150,150),
    batch_size = 75,
    color_mode='rgb',
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size = (150,150),
    batch_size = 75,
    color_mode='rgb',
    class_mode='categorical'
)

Found 3999 images belonging to 8 classes.
Found 800 images belonging to 8 classes.
Found 800 images belonging to 8 classes.
```

Una vez que acaba el proceso de augmentación de datos de datos, procedemos a establecer nuestras capas que conformarán la red neuronal. Estas estarán formadas inicialmente por:

- Valores de entrada = 150 x150 (píxeles en input size) y 3 n_colors que serían RGB
- 4 capas convolucionales y 4 capas de pooling, las capas convolucionales tendrán una función de activación relu
- 1 capa de Dropout (regularización) = 0.25
- 1 capa de aplanamiento Flatten
- 1 capa de neuronas profundas o densas (Dense), con 500 neuronas y función de activación relu
- 1 capa de salida con 8 neuronas (según las clases que tengo) y función de activación softmax, para la variabilidad probabilística del resultado

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import Dropout # importacion para la capa de regularización

# definimos los datos de entrada y construimos nuestras capas del modelo

input_size = 150 # tamaño de píxeles por imagen
n_colors = 3     # colores de rgb (red, green, blue)

layers = [
    keras.layers.Input(shape=(input_size, input_size, n_colors)), # capa entrada
    keras.layers.Conv2D(16, (3,3), activation = 'relu'), # capa de convolución con 16 filtros de tamaño 3x3 y funcion de act. relu
    keras.layers.MaxPool2D(2,2), # capa de pooling tamaño 2x2 con un stride de 1 incluido por defecto
    keras.layers.Conv2D(32, (3,3), activation = 'relu'), # capa de convolucion con 32 filtros
    keras.layers.MaxPool2D(2,2),
    keras.layers.Conv2D(64, (3,3), activation = 'relu'), # capa de convolucion con 64 filtros
    keras.layers.MaxPool2D(2,2),
    keras.layers.Conv2D(128, (3,3), activation = 'relu'), # capa de convolucion con 128 filtros
    keras.layers.MaxPool2D(2,2),
    Dropout(0.25), # capa de regularizacion al 25% (0.25)
    keras.layers.Flatten(), # capa de aplanamiento, transformamos en vector los datos
    keras.layers.Dense(500, activation = 'relu'), # capa de neuronas densa/oculta y funcion de activacion relu.
    keras.layers.Dense(8, activation = 'softmax') # capa de salida con 8 neuronas (1 por cada clase) y funcion de act. relu
]
```

Una vez definidas nuestras capas, compilamos el modelo de forma secuencial aplicando:

- Una función de pérdida (loss) = categorical_crossentropy
- Un optimizador Adam con un learning rate inicial de 0.001

- Y como métricas a tener en cuenta la exactitud (accuracy)

```
# establecemos modelo secuencial, según las capas generadas anteriormente

model = keras.Sequential(layers, name = "Modelo_CNN_Proyecto_Final")

# compilamos el modelo creado, estableciendo el optimizador segun indicamos en la descripcion

optimizer = keras.optimizers.Adam(learning_rate=0.001)

model.compile(
    loss = 'categorical_crossentropy',
    optimizer = optimizer,
    metrics = ['accuracy']
)

# Vemos la informacion de nuestro modelo CNN generado
model.summary()
```

Para finalizar mostramos la información general de nuestra red con [.summary()]

Model: "Modelo_CNN_Proyecto_Final"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 148, 148, 16)	448
max_pooling2d_4 (MaxPooling2D)	(None, 74, 74, 16)	0
conv2d_5 (Conv2D)	(None, 72, 72, 32)	4640
max_pooling2d_5 (MaxPooling2D)	(None, 36, 36, 32)	0
conv2d_6 (Conv2D)	(None, 34, 34, 64)	18496
max_pooling2d_6 (MaxPooling2D)	(None, 17, 17, 64)	0
conv2d_7 (Conv2D)	(None, 15, 15, 128)	73856
max_pooling2d_7 (MaxPooling2D)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_2 (Dense)	(None, 500)	3136500
dense_3 (Dense)	(None, 8)	4008

=====
Total params: 3237948 (12.35 MB)
Trainable params: 3237948 (12.35 MB)
Non-trainable params: 0 (0.00 Byte)

Tenemos una red neuronal tipo convolucional, con un total de más de 3 millones de parámetros, lista para proceder al entrenamiento con el que vamos a continuar más adelante.



2.1 Entrenamiento del modelo

Este punto es crucial para nuestro proyecto, ya que debemos considerar una mejoría escalable de la tasa de exactitud de nuestro modelo, hasta obtener el resultado óptimo. Para ello debemos de probar variaciones de hiperparámetros o modificar la construcción en la misma red, consiguiendo mejorar el resultado con el que empezamos.

Intento 1

Comenzamos el entrenamiento de nuestro modelo con los siguientes hiperparámetros en la red:

- 4 capas convolucionales:
 - o 1ª con 16 filtros y tamaño 3x3
 - o 2ª con 32 filtros y tamaño 3x3
 - o 3ª con 64 filtros y tamaño 3x3
 - o 4ª con 128 filtros y tamaño 3x3
 - o * Cada una con una función de activación relu *
- 4 capas de Pooling de tamaño 2x2 y stride = 1, activado por defecto en keras
- 1 capa de Dropout con valor de 0,25 (25% de desactivar procesos)
- 1 capa de neuronas densa con 500 neuronas y activación relu
- 1 capa de salida con 8 neuronas y función de activación softmax (no será cambiada)
- * un learning rate en nuestro optimizador de (0.001)*

A la hora de entrenar tenemos los siguientes hiperparámetros:

- Train_generator
- Steps_per_epoch = 40
- Epochs = 20
- Validation_data, que será nuestro val_generator.
- Validation_steps = 3



Entrenamos el modelo y obtenemos:

Epoch 20/20

40/40 - 52s - loss: 0.6524 - accuracy: 0.7652 - val_loss: 0.9285 - val_accuracy: 0.6700 - 52s/epoch - 1s/step

En nuestro primer entrenamiento del modelo, con los hiperparámetros indicados anteriormente, obtenemos una tasa de acierto del 76.5% en nuestros datos de entrenamiento y un 67% en los datos de validación.

Como primer intento es un resultado bastante bueno, pero vamos a probar algunas modificaciones de cara a mejorar el resultado.

Intento 2

En este entrenamiento vamos a hacer cambios en la estructura de capas de la red:

- Añadimos una capa convolucional más con 256 filtros, para conseguir extraer más características por imagen
- Ponemos un learning rate aún más pequeño pasando de 0.001 a 0.0001, para reducir aún más el ajuste de los pesos.

```
keras.layers.MaxPool2D(2,2),
keras.layers.Conv2D(128, (3,3), activation = 'relu'), # capa de convolucion con 128 filtros
keras.layers.MaxPool2D(2,2),
keras.layers.Conv2D(256, (3,3), activation = 'relu'), # capa de convolucion con 256 filtros
keras.layers.MaxPool2D(2,2),
Dropout(0.25), # capa de regularizacion al 25%
keras.layers.Flatten(), # capa de aplanamiento, transformamos en vector los datos
keras.layers.Dense(500, activation = 'relu'), # capa de neuronas densa/oculta y funcion de activacion relu.
keras.layers.Dense(8, activation = 'softmax') # capa de salida con 8 neuronas (1 por cada clase) y funcion de c
]

# establecemos modelo secuencial, según las capas generadas anteriormente

model = keras.Sequential(layers, name = "Modelo_CNN_Proyecto_Final")

# compilamos el modelo creado, estableciendo el optimizador segun indicamos en la descripcion

optimizer = keras.optimizers.Adam(learning_rate=0.0001)
```

Y volvemos a entrenar:

Epoch 20/20

40/40 - 52s - loss: 0.8800 - accuracy: 0.6947 - val_loss: 0.9432 - val_accuracy: 0.6600 - 52s/epoch - 1s/step



Obtenemos un peor rendimiento que con los valores aplicados inicialmente; consiguiendo un porcentaje de acierto en el conjunto de entrenamiento del 69.4% y de 66% en el conjunto de validación, por lo que restauramos los cambios realizados a los valores iniciales y volvemos a probar otras modificaciones.

Intento 3

Esta vez los cambios los vamos a efectuar sobre los hiperparámetros de la red, pero dejando su construcción inicial intacta:

- Modificamos la capa de dropout de 0.25 a 0.35, para reducir más el sobreajuste
- Añadimos a la capa densa 100 neuronas más pasando de 500 a 600, obteniendo mayores conexiones entre sí
- Modificamos los epoch de 20 a 25, aumentando el proceso computacional, pero obteniendo mayor mejor respuesta de entrenamiento.

```
Dropout(0.35), # capa de regularizacion al 35%
keras.layers.Flatten(), # capa de aplanamiento, transformamos en vector los datos
keras.layers.Dense(600, activation = 'relu'), # capa de neuronas densa/oculta y funcion de activacion

history = model.fit(
    train_generator,
    steps_per_epoch = 40,      # calculado al dividir nº de datos(train=4000) entre el batch_size
    epochs = 25,              # tendremos 20 repases al conjunto de datos
    validation_data = val_generator,
```

Y comprobamos el resultado del entrenamiento:

Epoch 25/25
40/40 - 52s - loss: 0.5566 - accuracy: 0.8037 - val_loss: 0.9846 - val_accuracy: 0.6533 - 52s/epoch - 1s/step

Hemos conseguido mejorar el rendimiento un 5% con respecto al modelo inicial, llegando hasta un 80.3% de tasa de acierto en el conjunto de entrenamiento y un 65.3% en el conjunto de validación, siendo este último menor que el entrenamiento inicial.



Intento 4

Hemos alcanzado una buena tasa de acierto, pero vamos a realizar un último intento de mejorar la red:

- Añadimos 100 neuronas más a la capa densa, pasando de 600 a 700
- Reducimos la capa de Dropout de 0.35 a 0.3
- Aumentamos el número de épocas de 25 a 30
- Aumentamos los steps en validación de 3 a 4

```
keras.layers.MaxPool2D(2,2),
Dropout(0.3), # capa de regularizacion al 30%
keras.layers.Flatten(), # capa de aplanamiento, transformamos en vector los datos
keras.layers.Dense(700, activation = 'relu'), # capa de neuronas densa/oculta y funcion de activacion

history = model.fit(
    train_generator,
    steps_per_epoch = 40,      # calculado al dividir nº de datos(train=4000) entre el batch_size
    epochs = 30,              # tendremos 30 repasos al conjunto de datos
    validation_data = val_generator,
    validation_steps = 4,      # calculado al dividir el nº de datos (validation=100) entre el batch_size
    verbose = 2               # con verbose = 2 obtenemos informacion de los datos importantes
```

Mostramos el resultado del entrenamiento:

Epoch 30/30

40/40 - 54s - loss: 0.3094 - accuracy: 0.8882 - val_loss: 1.2792 - val_accuracy: 0.7050 - 54s/epoch - 1s/step

Hemos obtenido una mejora del 8%, alcanzando finalmente una tasa de acierto del 88.8% en el conjunto de entrenamiento y del 70% en el de validación. También hemos obtenido la pérdida (loss) más baja de todos los intentos.

Tras realizar estas modificaciones doy por finalizado el proceso de mejora de la red. Para mejorar más el rendimiento habría que obtener un mayor número de datos, ya que, tras añadir modificaciones en los hiperparámetros, no distaría mucho más la tasa de acierto alcanzada.

Una vez cerrado el proceso de modificaciones y mejoras, vamos a pasar a la frase previa a la inferencia. Para ello vamos a evaluar el modelo, guardarlo y cargarlo para comprobar que funciona correctamente.

Para evaluar nuestro modelo utilizamos la función `model.evaluate()` de keras:

```
loss, accuracy = model.evaluate(test_generator)

# Mostramos La perdida y La tasa de acierto con nuestros datos de test
print(f"\nPerdida en la evaluación en el conjunto de prueba {loss}")
print(f"\nTasa de acierto en la evaluación del conjunto de prueba {accuracy}")
```

8/8 [=====] - 8s 1s/step - loss: 1.0795 - accuracy: 0.7138

Perdida en la evaluación en el conjunto de prueba 1.0795360803604126








Tasa de acierto en la evaluación del conjunto de prueba 0.7137500047683716

Tras realizar la evaluación obtenemos una tasa de acierto del 71%, menor que en el proceso de entrenamiento.

En la misma carpeta del proyecto guardamos nuestro modelo, para poder cargarlo con la API REST más tarde, para esto utilizamos una función de keras como es `model.save()`, y generamos un archivo **h5** con nuestro modelo entrenado.

```
# Para guardar nuestro modelo utilizamos la función de keras [.save()]

model.save("proyecto_final_cnn.h5")
```

	snow	23/04/2024 17:59	Carpeta de archivos	
	test	23/04/2024 18:20	Carpeta de archivos	
	train	23/04/2024 18:20	Carpeta de archivos	
	validation	23/04/2024 18:20	Carpeta de archivos	
	archive.zip	18/04/2024 1:42	Archivo WinRAR ZIP	711.710 KB
	Proyecto_DL_Rafa_Rodriguez_Vazquez.ip...	30/04/2024 0:03	Archivo IPYNB	6.986 KB
	proyecto_final_cnn.h5	29/04/2024 22:52	Archivo H5	52.726 KB

Ya con nuestro modelo guardado, lo cargamos para probarlo y realizamos una predicción, para comprobar que se cargó correctamente y se puede usar en la API.

```
# cargamos el modelo con la función de keras [.load_model()], y lo guardamos en una variable llamada proyect

project = keras.models.load_model("proyecto_final_cnn.h5")
```

```
# Establecemos una predicción de sobre el primer valor del conjunto -> index = 0, para ello primero vamos
# Las etiquetas reales del conjunto de datos y de este modo comparar para ver si la predicción acertó

index = 500 # el valor de la muestra que usare de ejemplo, en este caso hay 800 muestras y elijo la nº 500

true_labels = test_generator.classes

# Realizamos la predicción

predictions = project.predict(test_generator)

print("\nValor realizado por la predicción en concepto probabilístico:\n", [round(prob, 4) for prob in pr
print("\nValor Real:", true_labels[index])

8/8 [=====] - 8s 991ms/step

Valor realizado por la predicción en concepto probabilístico:
[0.0014, 0.7929, 0.0492, 0.0, 0.0998, 0.0396, 0.0156, 0.0013]

Valor Real: 5
```

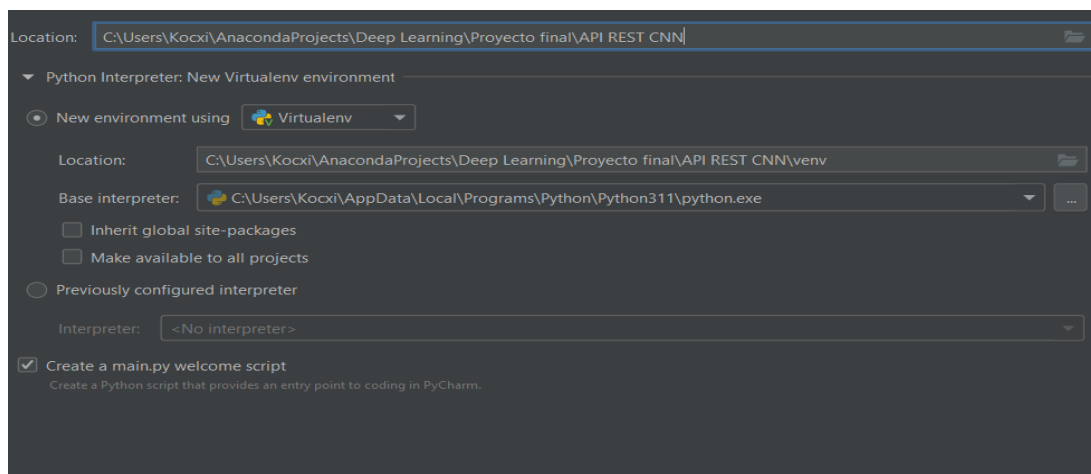
Como podemos comprobar el modelo se carga correctamente y con esto funcionará nuestra predicción a la hora de cargarlo en la API REST.

API REST

1. Construcción

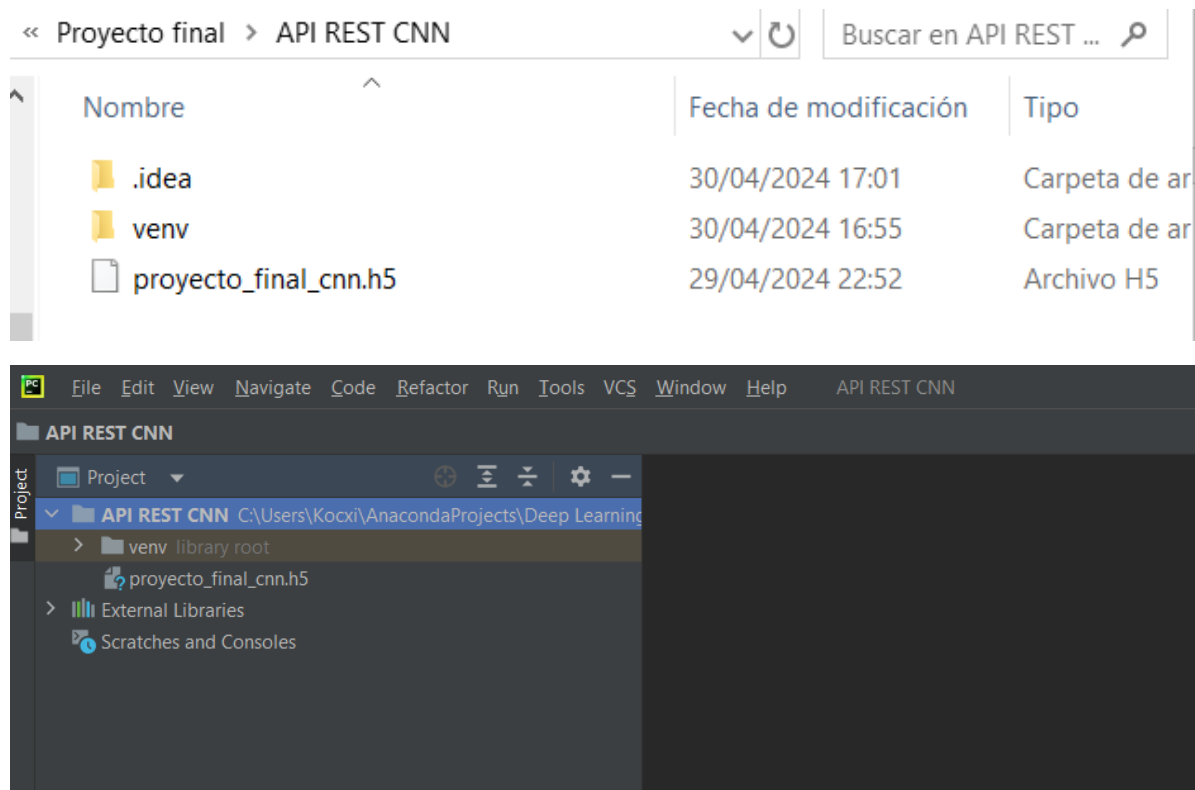
Para establecer el proceso de inferencia del modelo creado, vamos a usar el entorno de desarrollo Pycharm junto con la librería Flask y el módulo Swagger.

Primero creamos un entorno de desarrollo en Pycharm, en la misma carpeta de nuestro proyecto, llamado API REST CNN



Con esto se nos genera una carpeta donde dentro del entorno virtual se añadirán las dependencias, archivos python y demás para trabajar y ejecutar la API.

Movemos nuestro modelo guardado dentro de esta carpeta para que al instanciarlo el modelo no de error.



Con nuestro entorno de desarrollo creado, generamos los ficheros app.py y functions.py, quienes contendrán la ejecución y los procesos necesarios para que la API funcione correctamente.

Primero instalamos una dependencia de flask como es: REST-X, que nos ayudaran a generar automáticamente la documentación swagger teniendo en cuenta las definiciones que pongamos en la API.

Luego lo que nos permitirá, utilizar las funciones de la imagen como son tensorflow / keras y numpy, deberemos instalarlo también

```
(venv) PS C:\Users\Kocxi\AnacondaProjects\Deep Learning\Proyecto final\API REST CNN> pip install flask-restx flask tensorflow keras numpy
```

En el archivo app.py, incorporamos:

- Creación de una instancia para la aplicación de flask (app=Flask(__name__))
- Creación de una instancia para la clase Api dentro de Flask Rest-x, para configurar la api
- El modelo de predicción definido en el archivo functions.py
- El endpoint que contiene la url hacia la predicción (@api.route('/predict')); dentro de este la solicitud post que enviara la petición de la predicción al servidor que obtendremos de nuestra aplicación gracias a la función request de flask y en la que captara la imagen.
- Finalmente devuelve la predicción

```
from flask import Flask, request
from flask_restx import Api, Resource, fields
from functions import predict_image

app = Flask(__name__)
api = Api(app, version='1.0', title='Proyecto Final API REST para red CNN', description='API para realizar predicciones sobre imágenes')

prediction_model = api.model('Prediction',
                             {'image': fields.String(
                                 required=True,
                                 description='Imagen a predecir',
                                 location='form',
                             )})

@api.route('/predict')
class Prediction(Resource):
    @api.expect(prediction_model)
    def post(self):
        image_file = request.files.get('image')

        # Procesamos la imagen y realizamos la predicción
        prediction = predict_image(image_file)

        return {'prediction': prediction}

if __name__ == '__main__':
    app.run(debug=True)
```

En nuestro archivo functions.py, incorporamos:

- La carga de nuestro modelo entrenado que utilizaremos para realizar la predicción
- La función para realizar la predicción de la imagen, que tendrá varias líneas de código para procesar la imagen y la convierten en un array para poder transformarla en un tensor con el que trabaje la red.
- La predicción de la imagen seleccionada
- La distribución probabilística de que sea alguna de las clases de nuestros datos

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

# Cargamos el modelo previamente entrenado
model = load_model('proyecto_final_cnn.h5')

def predict_image(image_file):

    # Realizamos la predicción sobre una imagen y devolvemos el resultado.
    img = image.load_img(image_file, target_size=(150, 150))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img /= 255.0 # Normalizamos los valores de píxeles entre (0 ~ 1)

    # Predecimos
    prediction = model.predict(img)
    class_names = ['dew', 'fogsmog', 'frost', 'hail', 'lightning', 'rain', 'sandstorm', 'snow']
    predicted_class = class_names[np.argmax(prediction)]

    return predicted_class
```

Con todo nuestro código de la api preparado solo tenemos que ejecutar el programa, y nos daría la url de nuestra documentación de swagger

```
(venv) PS C:\Users\Kocxi\AnacondaProjects\Deep Learning\Proyecto final\API REST CNN> python app.py
2024-05-01 00:55:39.777039: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see slightly different numeric
from different computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
2024-05-01 00:55:40.994544: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see slightly different numeric
from different computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
2024-05-01 00:55:43.893739: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available C
S.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. 'model.compile_metrics' will be empty until you
WARNING:absl:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer.
* Serving Flask app 'app'
* Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with stat
```

Y comprobamos como quedaría nuestra web api

Proyecto Final API REST para red CNN ^{1.0}

[Base URL: /]
/swagger.json

API para realizar predicciones sobre imágenes meteorológicas

default Default namespace

POST /predict

Parameters

Try it out

Name	Description
------	-------------

payload * required
object
(body)

Example Value | Model

```
{
  "image": "string"
}
```

Parameter content type

application/json

Responses

Response content type application/json

Code	Description
------	-------------

200	Success
-----	---------

Models

Prediction {
 image*
}

Con esto finalizaría nuestro proyecto.



Cabe destacar que, en swagger ui no podemos cargar directamente una imagen para hacer la predicción, ya que este está diseñado principalmente para interactuar con endpoints que toman datos en formato JSON o similar (en el caso de nuestro código es a través de un json), y no proporciona una forma de cargar archivos directamente desde la interfaz de usuario.

De todos modos, aunque no podamos cargar directamente las imágenes desde swagger, la funcionalidad para procesar imágenes es válida para nuestra API y puede ser utilizada a través de otras herramientas o aplicaciones.

CONCLUSIONES Y POSIBLES MEJORAS

1. Hemos conseguido establecer un modelo de red neuronal convolucional que detecta las imágenes meteorológicas entrenadas correctamente, aunque no se pueda productivizar mediante la api de swagger, cumplimos con el objetivo inicial.
2. Hemos obtenido finalmente una tasa de acierto en el entrenamiento del modelo del 88% y en la evaluación del 71%, que, aunque puede ser mejorable, es razonablemente alta teniendo en cuenta la naturaleza del proyecto.
3. Para mejorar el rendimiento del modelo, podríamos probar otro tipo de configuración, ya sea en las capas de la red neuronal, en la división de los conjuntos de datos o en otra parte influyente del proyecto como es el entrenamiento.
4. Pudiendo ampliar el conjunto de datos con nuevas imágenes sobre las categorías existentes o incluso creando nuevas carpetas con clases diferentes, obtendríamos un mejor rendimiento del modelo y también más variable.



5. Podríamos utilizar otras herramientas externas como Postman (según he visto buscando información), para probar la ejecución de nuestra API con el envío de imágenes, que permite el envío de solicitudes http con archivos adjuntos como pueden ser las imágenes.
6. Podríamos utilizar arquitecturas de red más complejas o incluso añadir técnicas como la transferencia de aprendizaje, o incluso utilizar técnica de aumento de datos más avanzadas.