Detección de incendios forestales con Python

Marzo de 2023

Christian Amauri Amador Ortega

INDICE

(Este proyecto está parcialmente basado en otro [1] que es a su vez una versión mejorada de sí misma, hecha en 2023).

1. ¿Qué es?

Este proyecto consiste en un modelo de red neuronal convolucional para clasificar imágenes de bosques en dos clases: hay fuego o no hay fuego, usando Python para el procesamiento de los datos, su librería Keras para el diseño de la red neuronal, y Sklearn con Matplotlib para la visualización e interpretación de resultados.

El conjunto de datos está dividido en dos clases: fire y nofire. Las imágenes fueron recolectadas tras algunos minutos de búsqueda en internet. No pertenecen a alguna colección en especial. Varias partes de este proyecto están inspiradas o directamente basadas en otro proyecto personal llamado "Detección de leucemia con Python" cuya documentación se encuentra disponible en: Detección de leucemia con Python [1]. Sobre todo en la parte del data augmentation, y la visualización de resultados. La parte más representativa de este proyecto es el preprocesamiento y tratamiento de las imágenes de incendios, el cual difiere bastante del original ya que es probablemente el paso más importante en todo proyecto semejante.

El flujo de operaciones del proyecto, en general, es el siguiente:

1. **Aumento de datos** (*data_augmentation.py*): Aplicar zoom (20%), invertir en el eje X, invertir en el eje Y, o rotar entre 90, 180 y 270 grados.
2. **Preprocesamiento de datos** (*fire_detection.py*): Cargar las imágenes usando la librería *os*, convertirlas a escala de grises, normalizarlas, realzar su contraste, detectar bordes, detectar contornos, y extraer la Region Of Interest (ROI) en cada una. Finalmente dividir las imágenes en conjuntos de entrenamiento (80%), validación (10%) y prueba (10%).
3. **Diseño y entrenamiento de la red neuronal convolucional** (*fire_detection.py*): Usando Keras: crearla con *keras.models.Sequential*, diseñarla con *keras.layers*, compilarla con *.compile*, entrenarla con *.fit* y guardarla en formato HDF5 con *.save*.
4. **Visualización de resultados** (*fire_ detection.py*) (graficar la precisión en el entrenamiento y en la validación con Pyplot, luego imprimir los valores de la precisión y de la pérdida en consola, y finalmente mostrar la matriz de confusión del modelo con Sklearn)

2. Requerimientos

Las librerías necesarias para la ejecución del script y sus versiones están disponibles en la hoja de requirements.txt en la carpeta fuente del proyecto.

Los requerimientos técnicos que fueron usados fueron los siguientes:

- Entorno virtual de Python 3.10.0 (.venv).
- Windows 11 (arquitectura x64) (64 bits).
- Visual Studio Code 1.96.4.
- CPU: Intel Core i3 (i3-8145U) (2.1GHz).
- 8 GB de RAM.
- Unidad de almacenamiento M.2 (1TB de almacenamiento) (la carpeta fuente original, incluyendo entorno virtual, datasets, y códigos fuente, usa 2.11GB en total).

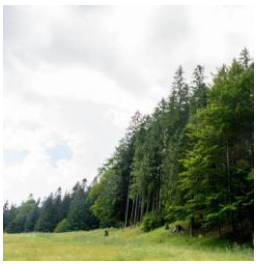
El tiempo de ejecución del proyecto en estas condiciones, con 20 épocas de entrenamiento fue de aproximadamente siete minutos, con 35 épocas fue de aproximadamente quince minutos.

3. Dataset

Como ya se mencionó, el data set no fue sacado de ninguna colección o base de datos específica, simplemente se recolectaron tras algunos minutos de búsqueda en internet. Tienen diferentes tamaños, pero el script se encarga de redimensionar cada imagen a un tamaño de 224 x 224 pixeles. Se procuró que todas las imágenes fueran semejantes en cuanto al tipo de bosque: bosque de coníferas.

Muestras originales de cada clase:


fire

nofire

4. Data augmentation (data_augmentation.py)

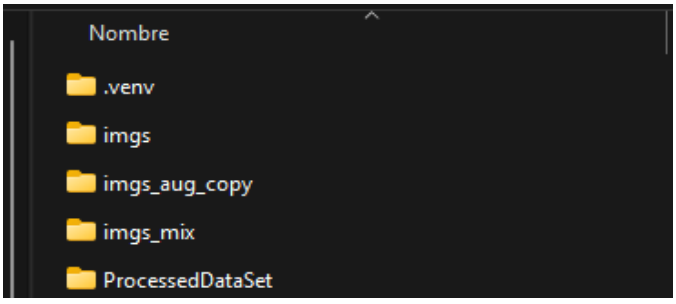
El dataset original está desbalanceado, ya que tiene 222 imágenes de la clase fire y 96 de la clase nofire. Por lo que, como primer paso de este proyecto, tendremos que hacer un data augmentation, el cual llevamos a cabo de la siguiente forma:

El script data_augmentation.py se encarga de realizar las copias adicionales que cada clase requiera para llegar a 230 y 191 respectivamente, y guarda dichas copias (también por clase) en un directorio llamado imgs_aug_copy. Esto lo hace la función “generate_image_copies” quien recibe un directorio a una clase (clase a aumentar), un directorio de salida (para las copias de esa clase), y el número de copias que dicha clase requiere. En este caso, tenemos que ingresar manualmente cada clase junto con su número de imágenes a aumentar, por cada ejecución (se trabajan todas las clases por ejecución, pero tenemos que definir manualmente cada valor para cada clase antes de ejecutar el script) esto se manejó así por si deseamos variar el número de imágenes resultantes del data augmentation.

Finalmente, el script data_augmentation.py crea una carpeta con las imágenes aumentadas creadas a partir de las originales “imgs_aug_copy”, y una carpeta en la que se copia el contenido de la carpeta original “imgs” junto con las imágenes resultantes en “imgs_aug_copy” (las imágenes aumentadas) (data set original + copias adicionales del data augmentation) dicha carpeta final se llama “imgs_mix” (esto se manejó así para tener más control sobre las carpetas individuales, sobre todo la original. Además podemos usar la carpeta “imgs_aug_copy”, que es mas pequeña que el data set original, para hacer pruebas de procesamiento de imágenes más veloces, en caso de ser necesario).

El script que compila la red neuronal convolucional (fire_detection.py) crea un nuevo directorio con copias de las imágenes en “imgs_mix” tras su correspondiente procesamiento (operaciones como blur, filtros de escala de grises y obtención de ROI). Realmente no necesitamos guardarlas en disco, pero las guardamos como parte de la documentación y porque no ocupan demasiado almacenamiento. El directorio se llama “ProcessedDataSet”

Resumen de la estructura de los directorios:



+ *imgs:* contiene el data set original, sin modificaciones.

+ *imgs_aug_copy:* contiene las imágenes extra, generadas con el script data_augmentation.py para nivelar el número de instancias por clase (principalmente a “nofire”).

+ *imgs_mix:* contiene la unión entre el dataset original y el aumentado.

+ *ProcessedDataSet:* guarda copias de las imágenes después de procesarlas.

5. Código principal

5.1 preprocesamiento

las dos funciones encargadas de realizar el procesamiento de las imágenes son *load_and_preprocess_image* y *detect_roi*. El flujo de operaciones que sigue cada imagen para ser procesada es el siguiente:

Función load_and_preprocess_image:

- ✓ La imagen se carga desde la ruta especificada usando Image.open(image_path).convert('RGB') para asegurar que esté en modo RGB.
- ✓ Se redimensiona la imagen a 224x224 píxeles con image.resize((224, 224)).
- ✓ La imagen se convierte en un arreglo NumPy para poder aplicar operaciones: np.array(image).
- ✓ Se convierte la imagen a HSV con cv2.cvtColor(image_np, cv2.COLOR_RGB2HSV) para facilitar la detección de colores relacionados con fuego.
- ✓ Se definen rangos de color en HSV para identificar fuego: rojo bajo ([0–10]), rojo alto ([160–180]) y amarillo-naranja ([11–35]).
- ✓ Se generan máscaras de fuego usando cv2.inRange para cada uno de estos rangos y se combinan todas en una máscara final fire_mask.
- ✓ Se crea una máscara inversa non_fire_mask para aislar las zonas que **no** contienen fuego.
- ✓ Se separan las áreas de fuego y no-fuego usando cv2.bitwise_and.
- ✓ En las zonas con fuego, se aumenta la saturación para resaltar los colores intensos con fire_area_hsv[:,:,1] *= 1.5 (luego se recorta a 255).
- ✓ Las zonas sin fuego se desaturan (proceso omitido parcialmente en tu código, pero se intuye que se desea convertirlas en grises o menos intensas).

Función detect_roi:

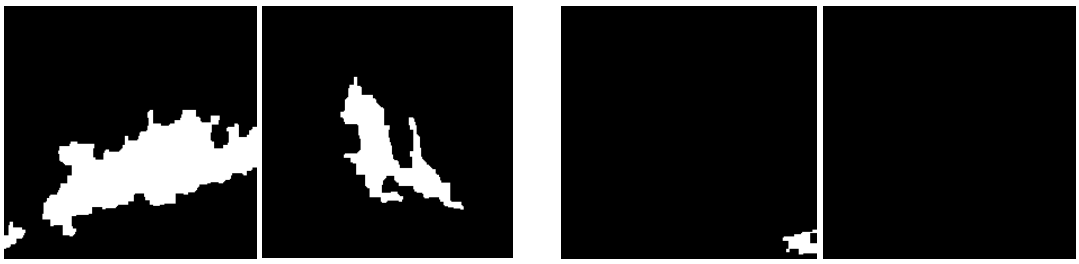
- ✓ Se verifica si la imagen está en escala de grises (2D); si es así, se convierte a BGR usando cv2.cvtColor(image, cv2.COLOR_GRAY2BGR).
- ✓ Si los valores de la imagen están normalizados en [0, 1], se escalan a [0, 255] y se convierten a tipo uint8.
- ✓ Se convierte la imagen a HSV (cv2.COLOR_BGR2HSV) y LAB (cv2.COLOR_BGR2LAB) para aprovechar distintas propiedades del color.
- ✓ Se crean máscaras en HSV para detectar colores característicos del fuego: rojo bajo, rojo alto, naranja, y amarillo, con cv2.inRange.

- ✓ Se combinan las máscaras HSV en una sola fire_mask_hsv mediante operaciones bitwise_or.
- ✓ En LAB, se separan los canales a y b, y se aplican umbrales para detectar valores altos (asociados a fuego).
- ✓ Se combinan los umbrales de a y b para formar una máscara lab_mask que indica regiones con colores cálidos.
- ✓ Se extraen los canales s (saturación) y v (brillo) del HSV y se umbralizan para detectar zonas brillantes y saturadas (fuego).
- ✓ Se combinan estas máscaras (lab_mask, fire_mask_hsv, y brillo/saturación) con operaciones bitwise_and.
- ✓ Se aplican operaciones morfológicas:
    - MORPH_OPEN elimina ruido.
    - MORPH_CLOSE rellena huecos.

- ✓ Se detectan contornos en la máscara resultante.
- ✓ Se filtran contornos según:
    - Área (mínima de 100 píxeles),
    - Relación de aspecto (preferentemente alto y delgado),
    - Circularidad (se evita lo demasiado redondo).

- ✓ Solo los contornos que cumplen con los criterios anteriores se rellenan en una máscara final.
- ✓ La máscara final se normaliza a valores entre [0, 1] para ser usada por modelos de aprendizaje automático.

Después de aplicar este procesamiento, las imágenes se ven de esta forma...



fire *nofire*

Básicamente la idea es resaltar los tonos propios del fuego (amarillo, rojo, naranja) mientras se penalizan los otros tonos (se baja la saturación a los tonos verdes, azules y otros) y luego aplicar la máscara binaria que resulta en una mancha blanca pronunciada correspondiente a la forma del fuego, y manchas blancas diminutas en donde no hay fuego, o ni siquiera una sola mancha.

Se usa el siguiente bucle for para llamar dichas funciones y procesar cada imagen, clasificándola según lo que le corresponde:

```
207  # Recorrer cada clase y cargar las imágenes correspondientes
208  print("[+] Cargando dataset...")
209  for class_id, class_name in enumerate(class_names):
210      class_path = os.path.join(dataset_dir, class_name)
211
212      # Asegurarnos de que la carpeta existe
213      if os.path.isdir(class_path):
214          # Recorrer todas las imágenes en la carpeta de la clase
215          for image_name in os.listdir(class_path):
216              image_path = os.path.join(class_path, image_name)
217
218              # Verificar que sea un archivo de imagen
219              if image_path.lower().endswith(('png', 'jpg', 'jpeg')):
220                  # Cargar y preprocesar la imagen
221                  image = load_and_preprocess_image(image_path)
222                  # Añadir la imagen y la etiqueta a las listas
223                  images.append(image)
224                  labels.append(class_id)  # Guardar el ID de la clase (fire=0, nofire=1)
225
```

El bucle recorre las clases definidas en *class_names*, y para cada clase, procesa todas las imágenes dentro de su carpeta correspondiente.

1. **Recorrido de Clases y Directorios**: Para cada clase, se obtiene su ruta y se verifica que el directorio exista.
2. **Procesamiento de Imágenes**: Dentro de cada clase, el bucle interno recorre los archivos de imagen válidos y llama a *load_and_preprocess_image(image_path)* para cargar y preprocesar cada imagen.
3. **Almacenamiento de Resultados**: La imagen preprocesada se agrega a la lista *images* y su etiqueta de clase (*class_id*) a la lista *labels*.

5.2 Modelo

El modelo comienza con una capa convolucional de 32 filtros 3x3 y ReLU, seguida de un *max pooling* 2x2, luego una capa convolucional de 64 filtros 3x3, *max pooling* 2x2, otra capa convolucional de 64 filtros 3x3 con *max pooling* 2x2, una capa convolucional de 128 filtros 3x3 con *max pooling* 2x2, otra capa convolucional de 128 filtros 3x3 con *max pooling* 2x2, luego se aplana la salida y se pasa a una capa densa de 128 neuronas con ReLU y regularización L2, seguida de una capa de *dropout* 50%, y finalmente una capa densa de 2 neuronas con activación Softmax para la clasificación en 2 clases.

El modelo se compila con el optimizador Adam y una tasa de aprendizaje de 0.0001, usando la función de pérdida *sparse categorical crossentropy* y evaluando la precisión como métrica.

```python
# Crear el modelo
print("[+] Creando modelo...")
model = keras.models.Sequential()

# Capas convolucionales + capa de max pooling
model.add(keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 1)))  # Para imagenes de 224x224 con 1 canal (gris)
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Flatten()) # Aplanar la imagen para pasar a la capa densa
model.add(keras.layers.Dense(128, activation='relu',kernel_regularizer=keras.regularizers.l2(0.01))) # Capa densa con 128 neuronas y Dropout
model.add(keras.layers.Dropout(0.5))  # Dropout para evitar sobreajuste
model.add(keras.layers.Dense(2, activation='softmax'))  # Capa de salida (2 clases en total)

# (Usamos softmax con 2 neuronas en la ultima capa porque aparentemente reacciona mejor que con sigmoid y 1 neurona)

# Compilar el modelo
# (Usamos sparse_categorical_crossentropy porque usamos softmax en la ultima capa)
print("[+] Compilando modelo...")
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

El modelo se entrena durante 35 épocas con un tamaño de lote de 32, y luego se guarda en un archivo HDF5 llamado 'fire_detection_v2_35_epochs.h5'.

```python
# !!! Entrenar al modelo !!!
print("[+] Entrenando modelo...")
history = model.fit(X_train, y_train, epochs=35, batch_size=32, validation_data=(X_val, y_val))
nombre = 'fire_detection_v2_35_epochs.h5'
print(f"[+] Guardando modelo como: {nombre}...")
model.save(nombre)
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	320
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 64)	0
conv2d_3 (Conv2D)	(None, 24, 24, 128)	73,856
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 128)	0
conv2d_4 (Conv2D)	(None, 10, 10, 128)	147,584
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 128)	0
flatten (Flatten)	(None, 3200)	0
dense (Dense)	(None, 128)	409,728
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 2)	258

Total params: 687,170 (2.62 MB)
Trainable params: 687,170 (2.62 MB)
Non-trainable params: 0 (0.00 B)

Resumen de la red neuronal (tiempo de ejecución).

```
[+] Estableciendo ruta del dataset y etiquetas...
[+] Cargando dataset...
    Dimensiones de las imágenes: (431, 224, 224)
    Dimensiones de las etiquetas: (431,)

[+] Creando copia del data set procesado...
[+] Separando en conjunto de entrenamiento, test y validación (80%,10%,10%)...
    Dimensiones del conjunto de entrenamiento: (344, 224, 224, 1)
    Dimensiones del conjunto de validación: (44, 224, 224, 1)
    Dimensiones del conjunto de prueba: (43, 224, 224, 1)

[+] Creando modelo...
```

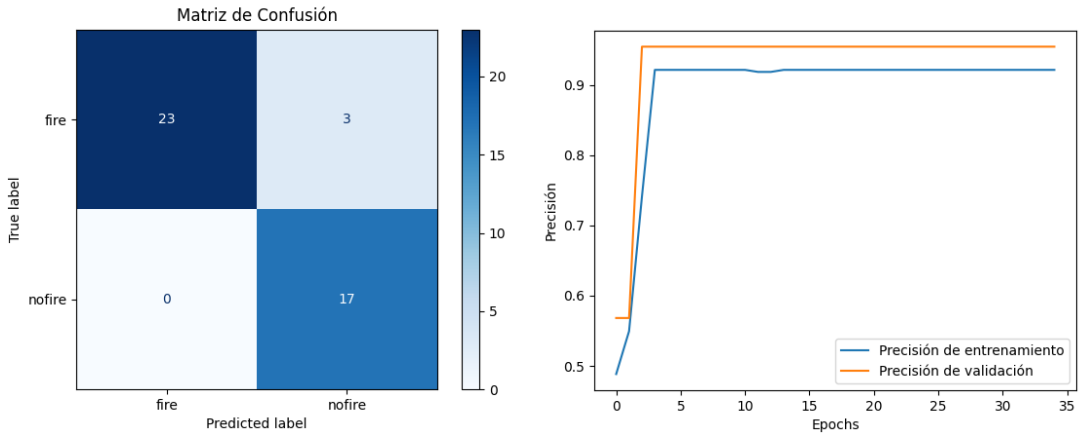
Parámetros de los datos de entrenamiento, validación y prueba (tiempo de ejecución).

6. Resultados

Resultados con **35** épocas de entrenamiento:

```
11/11 ━━━━━━━━━━━━━━━━━━━━ 11s 991ms/step - accuracy: 0.9014 - loss: 0.4417 - val_accuracy: 0.9545 - val_loss: 0.3291
[+] Guardando modelo como: fire_detection_v2_35_epochs.h5...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.s
ave_model(model, 'my_model.keras')`.
[+] Evaluando modelo...
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 112ms/step - accuracy: 0.9327 - loss: 0.3619
    Pérdida en el conjunto de prueba: 0.36930322647094727
    Precisión en el conjunto de prueba: 0.930232584476471
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 193ms/step
End
```

Métricas finales en terminal.



Matriz de confusión, pérdida y precisión en el conjunto de prueba, graficados con Pyplot.

7. Conclusiones

93% de precisión junto con una pérdida de 0.36 no está nada mal para un modelo que solo usa Keras y una arquitectura personalizada cuyo tiempo de ejecución para 35 épocas es de máximo veinte minutos, sobre todo en un equipo con parámetros técnicos tan básicos como este.

Si le agregáramos más capas convolucionales o si la combináramos con otra red previamente entrenada como VGG16 o si simplemente le agregáramos más épocas, podría ser incluso mejor, y sin invertir demasiado esfuerzo en ello, incluso si tuviéramos un mejor equipo podríamos facilitar mucho dichas pruebas. Pero por ahora, el modelo no está nada mal. Probablemente lo que benefició mucho al aprendizaje de la red, no fue tanto su arquitectura sino el preprocesamiento de los datos, que permitió que los patrones fueran más fáciles de analizar para ella (básicamente las imágenes sin fuego eran casi o completamente negras).

8. Referencias

1. *Portafolio/Detección de leucemia con Python at main · ChristianOrtegaFCC/Portafolio · GitHub*

2. *Keras: Deep learning for humans*. (s.f.). Keras.Io. Recuperado de: https://keras.io/

3. *Scikit-learn*. (s.f.). Scikit-learn.org. Recuperado de: https://scikit-learn.org/stable/index.html

4. *NumPy*. (s.f.). Numpy.org. Recuperado de: https://numpy.org/

5. Python, R. (9 de julio de 2014). *Instalar PIL / Pillow y aplicar efectos visuales*. Recursos Python. Recuperado de: https://recursospython.com/guias-y-manuales/instalar-pil-pillow-efectos/

6. *OpenCV: OpenCV-python tutorials*. (s.f.). Opencv.org. Recuperado de: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html