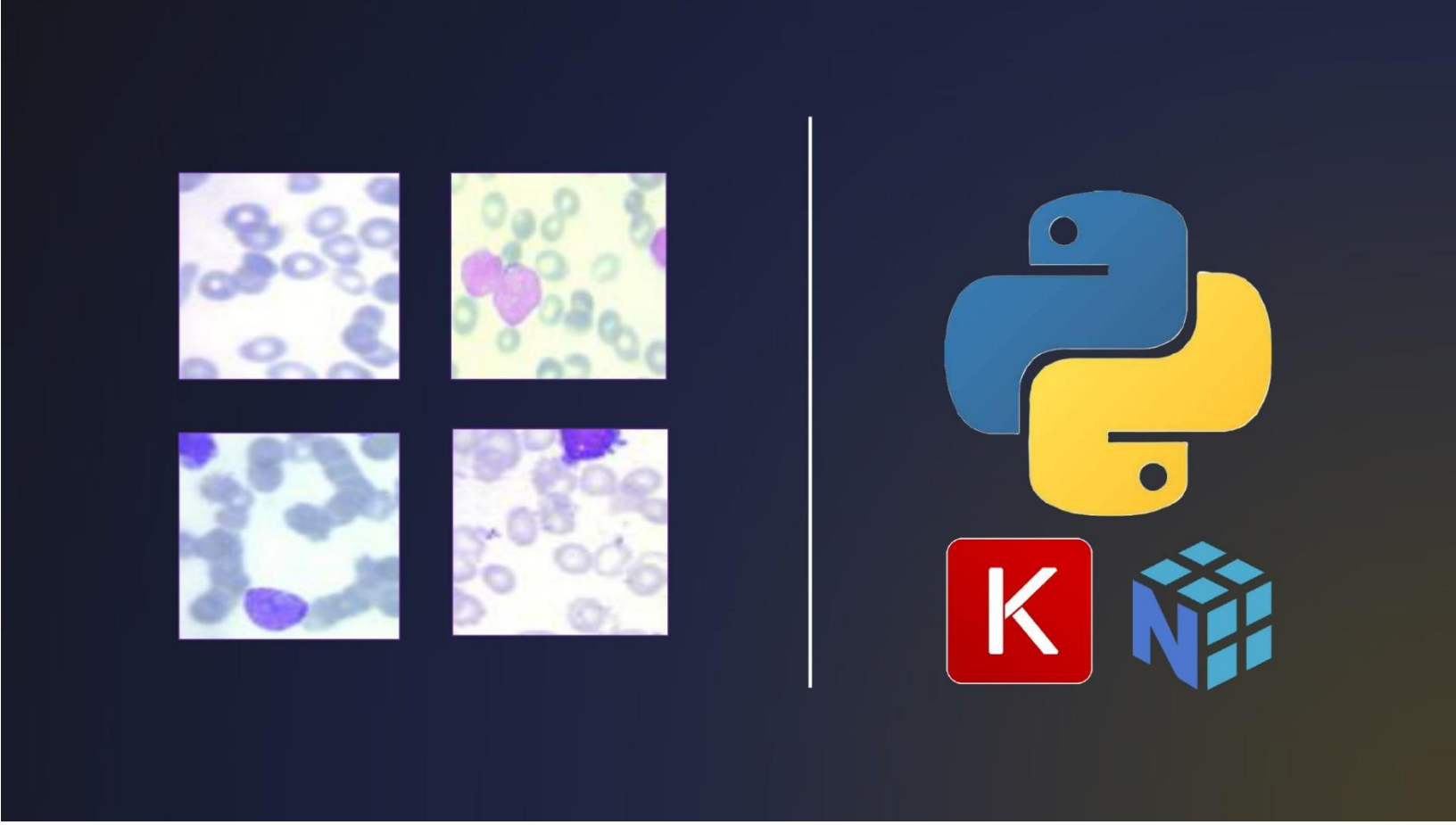


Detección de Leucemia con Python

Abril de 2023

Christian Amauri Amador Ortega

INDICE

1. ¿Qué es?

Este proyecto consiste en un análisis de datos para detectar cuatro clases de sangre relacionadas con la Leucemia Linfoblástica Aguda (LLA), usando Python para el procesamiento de los datos, su librería Keras para el diseño de la red neuronal convolucional, y Sklearn con Matplotlib para la visualización e interpretación de resultados.

El conjunto de datos está dividido en dos clases: benigna y maligna. La primera comprende muestras hematógenas, y la segunda corresponde al grupo de LLA con tres subtipos de linfoblastos malignos: LLA Pro-B, Pre-B y Pre-B Temprana.... [1].

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** (*leukemia.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, validación y prueba.
3. **Diseño y entrenamiento de la red neuronal convolucional** (*leukemia.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** (*leukemia.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 2.17GB en total).

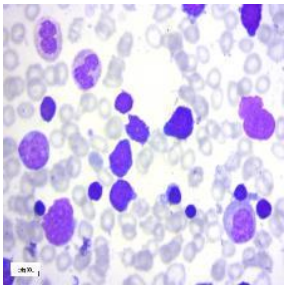
El tiempo de ejecución del proyecto en estas condiciones, con 25 épocas de entrenamiento fue de aproximadamente una hora, con 40 épocas fue de aproximadamente una hora y media.

3. Dataset

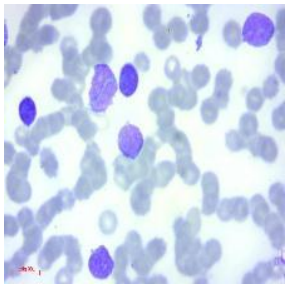
El dataset original está disponible este enlace [1]. Junto con la siguiente información que describe brevemente el proceso de obtención y clasificación de este conjunto de datos:

Las imágenes de este conjunto de datos se prepararon en el laboratorio de médula ósea del Hospital Taleqani (Teherán, Irán). Este conjunto de datos consistió en 3242 imágenes de PBS de 89 pacientes sospechosos de LLA, cuyas muestras de sangre fueron preparadas y teñidas por personal de laboratorio capacitado. Todas las imágenes se tomaron utilizando una cámara Zeiss en un microscopio con un aumento de 100x y se guardaron como archivos JPG. Un especialista, utilizando la herramienta de citometría de flujo, realizó la determinación definitiva de los tipos y subtipos de estas células... [1].

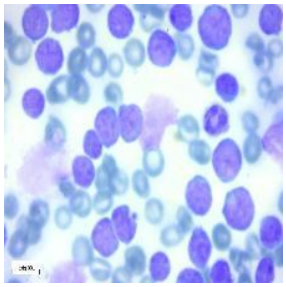
Muestras originales de cada clase:



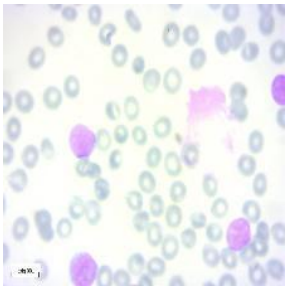
Benign



Early



Pre



Pro

4. Data augmentation (data_augmentation.py)

El dataset original está desbalanceado, ya que tiene 504 imágenes de la clase Benign, 985 de la clase Early, 963 de la clase Pre, y 804 de la clase Pro. Por lo que, como primer paso de este análisis de datos, tendremos que hacer un data augmentation, el cual llevamos a cabo de la siguiente forma:

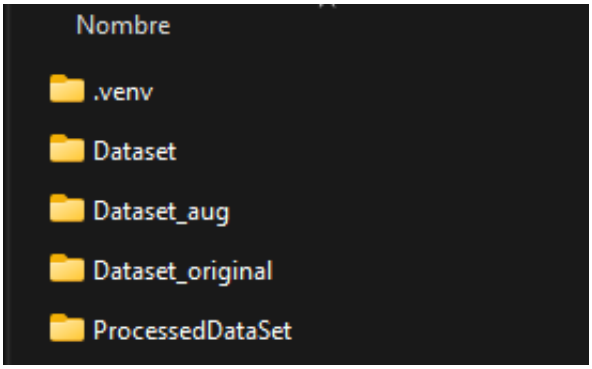
El script data_augmentation.py se encarga de realizar las copias adicionales que cada clase requiera para llegar a las mil instancias, y guarda dichas copias (también por clase) en un directorio llamado Dataset_aug. Esto lo hace la función “generate_image_copies” quien recibe un directorio a una clase (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 trabaja una clase por ejecución) esto se manejó así por si deseamos controlar de forma más manual/personalizable el data augmentation.

Finalmente, desde el administrador de archivos de Windows, juntamos el dataset original junto con el aumentado en un nuevo directorio que simplemente se llamará “Dataset” el cual es el que está listo para su uso en el script (leukemia.py) que contiene el preprocesamiento de los datos y la red neuronal convolucional.

(Nota: Me tomé la libertad de hacerlo así para ahorrar un poco de tiempo de codificación, ya que, al ser un análisis sobre únicamente cuatro clases, no necesito automatizar demasiadas cosas, además de que el uso de almacenamiento no es crítico, el dataset original usa 75MB de almacenamiento, por lo que el uso de recursos y poder computacional es completamente despreciable en este caso).

Finalmente, el script que compila la red neuronal convolucional (leukemia.py) crea un nuevo directorio con copias de las imágenes en “Dataset” 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, (de nuevo) no ocupan demasiado almacenamiento.

Resumen de la estructura de los directorios:



+ *Dataset_original:* contiene el data set descargado directo de la página, sin modificaciones.

+ *Dataset_aug:* contiene las imágenes extra, generadas con el script gen.py para nivelar el número de instancias por clase (en este caso, todas a mil imágenes).

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

+ *Processed data:* 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:

- ✓ La imagen **es cargada** desde el archivo especificado en la ruta image_path.
- ✓ Se asegura de que esté en **modo RGB** usando Image.open(image_path).convert('RGB').
- ✓ La imagen **se redimensiona a 224x224 píxeles** utilizando image.resize((224, 224)).
- ✓ La imagen **se convierte a un arreglo NumPy** mediante np.array(image) para realizar operaciones numéricas.
- ✓ La imagen **se convierte de RGB a escala de grises** usando OpenCV: cv2.cvtColor(image, cv2.COLOR_RGB2GRAY).
- ✓ Los valores de **los píxeles se normalizan** a un rango entre **0 y 1**: image = image / 255.0.
- ✓ **Se aplica un desenfoque gaussiano** para suavizar la imagen y reducir el ruido: blurred = cv2.GaussianBlur(image, (5, 5), 0).
- ✓ La imagen **se realza** combinando la imagen original con el desenfoque gaussiano **mediante una fórmula de nitidez**: sharpened = cv2.addWeighted(image, 1.5, blurred, -0.2, 0).
- ✓ Los valores resultantes se limitan al rango [0, 1] con: image = np.clip(sharpened, 0, 1).
- ✓ Las dimensiones de la imagen se expanden para agregar un canal adicional, pasando de (224, 224) a (224, 224, 1) para adaptarse a modelos que esperan este formato: image = np.expand_dims(image, axis=-1).

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

Benign *Early* *Pre* *Pro*

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

```
76
77  # Recorrer cada clase y cargar las imágenes correspondientes
78  for class_id, class_name in enumerate(class_names):
79      class_path = os.path.join(dataset_dir, class_name)
80
81      # Asegurarnos de que la carpeta existe
82      if os.path.isdir(class_path):
83          # Recorrer todas las imágenes en la carpeta de la clase
84          for image_name in os.listdir(class_path):
85              image_path = os.path.join(class_path, image_name)
86
87              # Verificar que sea un archivo de imagen
88              if image_path.lower().endswith(('png', 'jpg', 'jpeg')):
89                  # Cargar y preprocesar la imagen
90                  image = load_and_preprocess_image(image_path)
91                  # Añadir la imagen y la etiqueta a las listas
92                  images.append(image)
93                  labels.append(class_id)  # Guardar el ID de la clase (Benign=0, Early=1, Pre=2, Pro=3)
94
```

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 4 neuronas con activación Softmax para la clasificación en 4 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.

```
149
150  # Crear el modelo
151  model = keras.models.Sequential()
152
153  # Capas convolucionales + capa de max pooling
154  model.add(keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 1)))  # Para imágenes de 224x224 con 1 canal (gris)
155  model.add(keras.layers.MaxPooling2D((2, 2)))
156  model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
157  model.add(keras.layers.MaxPooling2D((2, 2)))
158  model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
159  model.add(keras.layers.MaxPooling2D((2, 2)))
160  model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
161  model.add(keras.layers.MaxPooling2D((2, 2)))
162  model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
163  model.add(keras.layers.MaxPooling2D((2, 2)))
164  model.add(keras.layers.Flatten()) # Aplanar la imagen para pasar a la capa densa
165  model.add(keras.layers.Dense(128, activation='relu',kernel_regularizer=keras.regularizers.l2(0.01))) # Capa densa con 128 neuronas y Dropout
166  model.add(keras.layers.Dropout(0.5))  # Dropout para evitar sobreajuste
167  model.add(keras.layers.Dense(4, activation='softmax'))  # Capa de salida (4 clases en total), Usamos softmax porque tenemos múltiples clases
168
169  # Compilar el modelo
170  model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
171  model.summary()
172
```

El modelo se entrena durante 40 épocas con un tamaño de lote de 32, y luego se guarda en un archivo HDF5 llamado 'leukemia_detection_x_epochs.h5'.

```
173  # Entrenar al modelo
174  history = model.fit(X_train, y_train, epochs=40, batch_size=32, validation_data=(X_val, y_val))
175  model.save('leukemia_detection_v8_40_epochs.h5')
```

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, 4)	516

Resumen de la red neuronal (tiempo de ejecución).

```
Dimensiones de las imágenes: (4000, 224, 224, 3)
Dimensiones de las etiquetas: (4000,)

Dimensiones del conjunto de entrenamiento: (3200, 224, 224, 1)
Dimensiones del conjunto de validación: (400, 224, 224, 1)
Dimensiones del conjunto de prueba: (400, 224, 224, 1)
```

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

6. Resultados

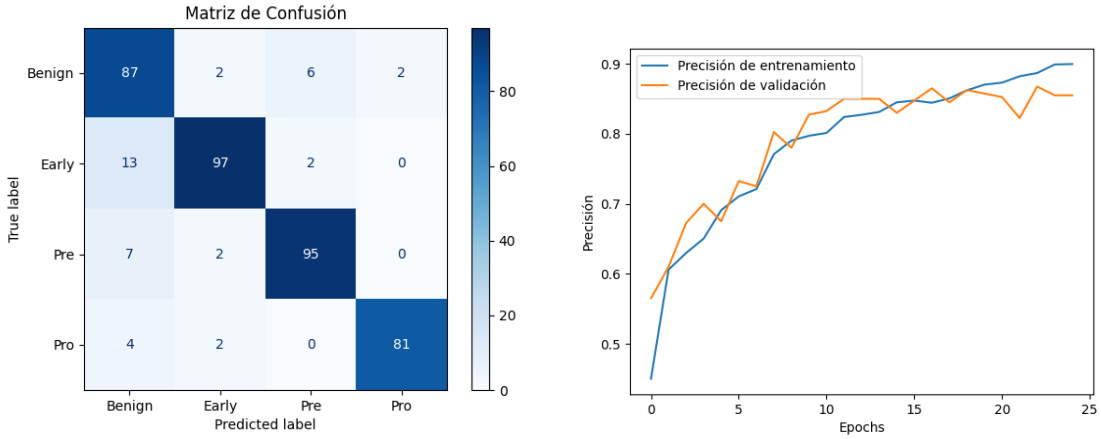
en el proceso de búsqueda de *hiperparámetros*, se experimentó manualmente añadiendo más capas densas, restando porcentaje al conjunto de entrenamiento hasta en un 70%, usando un *learning rate* de 2, 5 y 10 veces más grande, entrenando con menos épocas, y usando técnicas diferente de preprocesamiento de los datos (diferentes grados de contraste, entrenar con los tres canales de colores, omitir la obtención de la *Region Of Interest*, etc.) pero los parámetros que hasta el momento resultaron en una precisión más alta, fueron los que se han dejado en la presente versión del código fuente.

Resultados con **25** épocas de entrenamiento:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER 100                    Python

100/100 ━━━━━━━━━━━━━━━━━━━━ 128s 1s/step - accuracy: 0.8537 - loss: 0.5565 - val_accuracy: 0.8450 - val_loss: 0.5969
Epoch 19/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 127s 1s/step - accuracy: 0.8678 - loss: 0.5246 - val_accuracy: 0.8625 - val_loss: 0.5236
Epoch 20/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 113s 1s/step - accuracy: 0.8637 - loss: 0.5117 - val_accuracy: 0.8575 - val_loss: 0.5160
Epoch 21/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 98s 980ms/step - accuracy: 0.8763 - loss: 0.4681 - val_accuracy: 0.8525 - val_loss: 0.5079
Epoch 22/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 98s 981ms/step - accuracy: 0.8984 - loss: 0.4382 - val_accuracy: 0.8225 - val_loss: 0.5545
Epoch 23/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 98s 981ms/step - accuracy: 0.8804 - loss: 0.4514 - val_accuracy: 0.8675 - val_loss: 0.5212
Epoch 24/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 98s 981ms/step - accuracy: 0.9126 - loss: 0.4073 - val_accuracy: 0.8550 - val_loss: 0.5181
Epoch 25/25
100/100 ━━━━━━━━━━━━━━━━━━━━ 99s 992ms/step - accuracy: 0.9080 - loss: 0.4057 - val_accuracy: 0.8550 - val_loss: 0.4843
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')`.
13/13 ━━━━━━━━━━━━━━━━━━━━ 3s 202ms/step - accuracy: 0.8988 - loss: 0.5171
Pérdida en el conjunto de prueba: 0.5195881724357605
Precisión en el conjunto de prueba: 0.8999999761581421
13/13 ━━━━━━━━━━━━━━━━━━━━ 3s 196ms/step
End
```

Métricas finales en terminal.



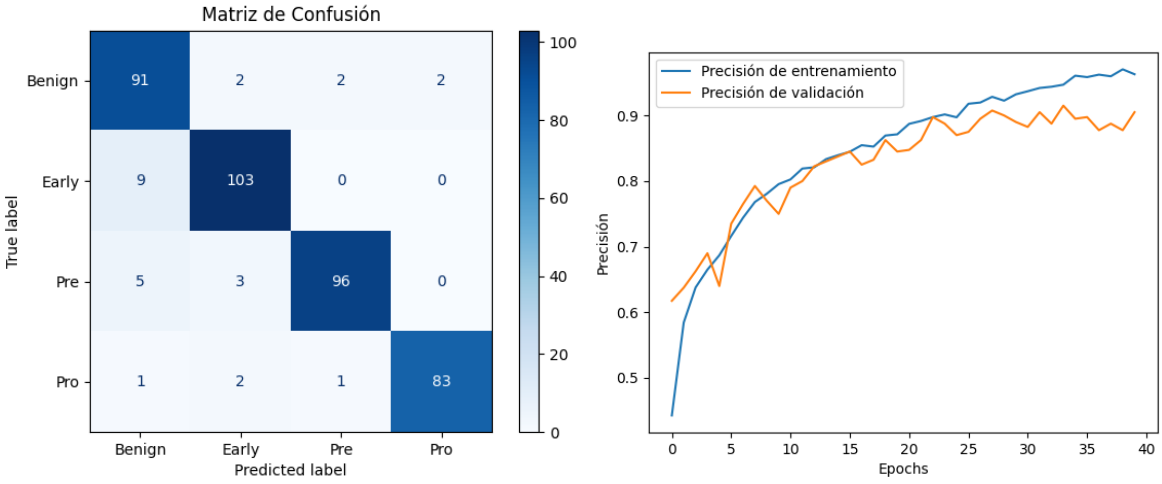
Matriz de confusión, pérdida y precisión en el conjunto de prueba, graficados con Pyplot.

Resultados con **40** épocas de entrenamiento:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER 100                    Python

100/100 ━━━━━━━━━━━━━━━━━━━━ 124s 1s/step - accuracy: 0.9221 - loss: 0.3313 - val_accuracy: 0.9000 - val_loss: 0.4075
Epoch 30/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 115s 1s/step - accuracy: 0.9319 - loss: 0.3192 - val_accuracy: 0.8900 - val_loss: 0.3997
Epoch 31/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 99s 985ms/step - accuracy: 0.9438 - loss: 0.3011 - val_accuracy: 0.8825 - val_loss: 0.4386
Epoch 32/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 146s 1s/step - accuracy: 0.9379 - loss: 0.2979 - val_accuracy: 0.9050 - val_loss: 0.3964
Epoch 33/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 171s 2s/step - accuracy: 0.9474 - loss: 0.2742 - val_accuracy: 0.8875 - val_loss: 0.4177
Epoch 34/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 115s 1s/step - accuracy: 0.9497 - loss: 0.2630 - val_accuracy: 0.9150 - val_loss: 0.3709
Epoch 35/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 138s 1s/step - accuracy: 0.9679 - loss: 0.2248 - val_accuracy: 0.8950 - val_loss: 0.4067
Epoch 36/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 122s 1s/step - accuracy: 0.9569 - loss: 0.2427 - val_accuracy: 0.8975 - val_loss: 0.3994
Epoch 37/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 168s 2s/step - accuracy: 0.9658 - loss: 0.2181 - val_accuracy: 0.8775 - val_loss: 0.4612
Epoch 38/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 112s 1s/step - accuracy: 0.9611 - loss: 0.2199 - val_accuracy: 0.8875 - val_loss: 0.4032
Epoch 39/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 112s 1s/step - accuracy: 0.9679 - loss: 0.2073 - val_accuracy: 0.8775 - val_loss: 0.4968
Epoch 40/40
100/100 ━━━━━━━━━━━━━━━━━━━━ 131s 1s/step - accuracy: 0.9653 - loss: 0.2116 - val_accuracy: 0.9050 - val_loss: 0.3346
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')`.
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 300ms/step - accuracy: 0.9362 - loss: 0.3723
Pérdida en el conjunto de prueba: 0.3949871063232422
Precisión en el conjunto de prueba: 0.9325000047683716
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 299ms/step
End
```

Métricas finales en terminal.



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

7. Conclusiones

90% de precisión junto con una pérdida de 0.37 no está nada mal para un modelo que solo usa Keras y una arquitectura personalizada cuyo tiempo de ejecución para 40 épocas es de máximo dos horas, 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.

8. Referencias

1. Amir Eshraghi, M. (2021). *Blood Cells Cancer (ALL) dataset* [Data set]. Recuperado de: https://www.kaggle.com/datasets/mohammadamireshraghi/blood-cell-cancer-all-4class

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