DermAI

Pruebas de modelado/entrenamiento

ARMADO DEL DATASET Y TRANSFORMACIóN DE IMAGENES

Se cargó el Dataset original sin eliminar nada). Tampoco se eliminaron imágenes.

Se cargaron y transformaron las imágenes utilizando esta función

```
def load_image(image_name, image_dir):
    image_path = os.path.join(image_dir, image_name)
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, [112, 112])  #ajuste de tamaño….no se xq elegi ese
    img = img / 255.0  # Normalizar
    return img.numpy()  # Convertir a numpy array para guardar
```

Las etiquetas (variable dx en el dataset) se convirtieron usando one-hot encoding

```
etiquetas = pd.get_dummies(df['dx']).values  # Convertir etiquetas a one-hot
encoding
```

se guardaron las imágenes transformadas en tensores y las etiquetas en un archivo h5py

```
with h5py.File('datasetCompleto_lesiones.h5', 'w') as hdf:
    hdf.create_dataset('imagenes', data=imagenes_array)
    hdf.create_dataset('dx_labels', data=etiquetas)
```

Es grande, pero ocupa la mitad que las imágenes en formato jpg (aprox 1.5 Gb)

MODELADO Y ENTRENAMIENTO

Modelo Básico y entrenamientos 1 y 2 están en el archivo: ModeladoyEntrenamiento.py

Modelo básico y entrenamiento 3 en el archivo : ModeladoyEntrenamientoV2.py

Modelo Básico

Antes de entrenar es necesario dividir el dataset en datos para entrenamiento y datos para testeo

```
imagenes_train, imagenes_test, etiquetas_train, etiquetas_test = train_test_split(
imagenes, etiquetas_dx, test_size=0.2, random_state=42 )
```

Para la **definición del modelo** se tomó el ejemplo de la notebook compartida por los profes y se adecuaron algunos parámetros que tienen que ver con el tamaño de las imágenes y la cantidad de etiquetas que tenemos nosotras.

```
model_cnn = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(112, 112, 3)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(7, activation='softmax')
])
```

El modelo tiene la siguiente estructura una vez compilado

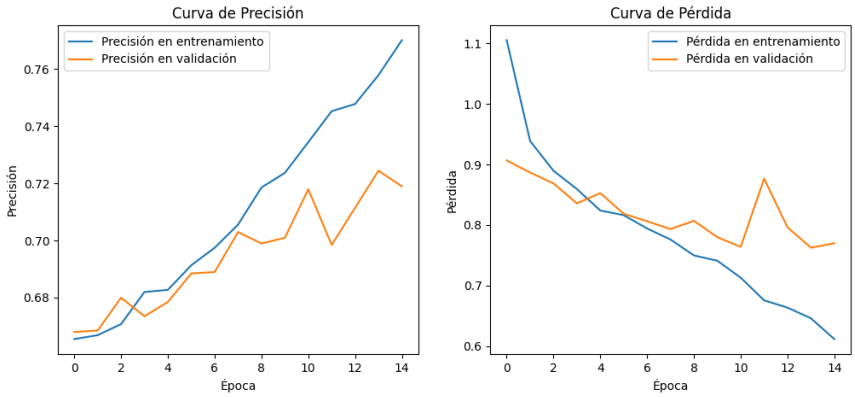
Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 110, 110, 32)	896
max_pooling2d (MaxPooling2D)	(None, 55, 55, 32)	0
conv2d_1 (Conv2D)	(None, 53, 53, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 26, 26, 64)	0
flatten (Flatten)	(None, 43264)	0
dense (Dense)	(None, 128)	5,537,920
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 7)	903

Total params: 5,558,215 (21.20 MB)
Trainable params: 5,558,215 (21.20 MB)
Non-trainable params: 0 (0.00 B)

Entrenamiento 1

Se entrenó con 15 con tamaño del lote =20. Se obtuvieron los siguientes valores de precisión y pérdida



Se observa que

- que la precisión de validación es menor que la de entrenamiento, y
- que la pérdida de validación no sige disminuyendo consistentemente,

Esto puede sugerir que el modelo está comenzando a sobreajustarse a los datos de entrenamiento. El sobreajuste ocurre cuando el modelo aprende los patrones específicos de los datos de entrenamiento, pero no generaliza bien a los datos que no ha visto antes.

Posibles mejoras (según chatgpt)

Aumentar el tamaño del conjunto de datos: Si tienes la posibilidad de agregar más datos de entrenamiento, esto puede ayudar a mejorar la generalización del modelo y evitar el sobreajuste.

Regularización adicional: Puedes añadir más capas de **Dropout** para forzar al modelo a no depender tanto de algunas conexiones, lo que podría ayudar a reducir el sobreajuste. También puedes probar con **regularización L2** en las capas densas para penalizar los pesos altos.

Ajustar el número de épocas: Podrías intentar detener el entrenamiento antes, utilizando **Early Stopping** para evitar que el modelo aprenda en exceso sobre los datos de entrenamiento y comience a sobreajustarse.

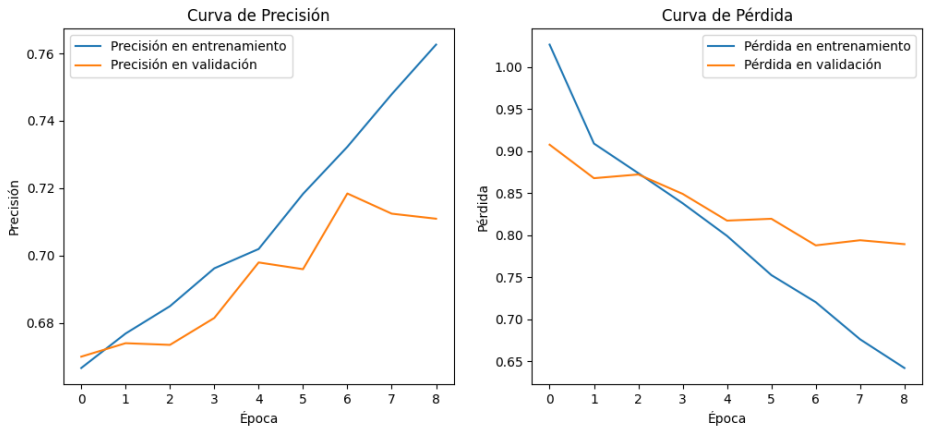
Aumentar la capacidad del modelo: Si sospechas que el modelo es demasiado simple, podrías probar con más capas convolucionales o más filtros en las capas ya existentes para aumentar la capacidad del modelo y permitirle capturar más patrones complejos en los datos.

Entrenamiento 2: Ajuste del número de épocas usando Early Stopping

Se modificó el entrenamiento agregando el early_stopping en el parámetro callbacks

```
early_stopping = EarlyStopping(monitor='val_loss', patience=2,
restore_best_weights=True)

history_cnn = model_cnn.fit(imagenes_train, etiquetas_train, epochs=15,
batch_size=20, validation_data=(imagenes_test,
etiquetas_test),callbacks=[early_stopping])
```



En este caso los resultados que se obtuvieron muestran una mejora constante en la precisión del modelo a lo largo de las épocas, alcanzando una **exactitud del 71.84%** en el conjunto de prueba tras el entrenamiento. El uso de la técnica de **detención temprana** (EarlyStopping) ayudó a evitar un sobreajuste, deteniendo el entrenamiento cuando el rendimiento en los datos de validación dejó de mejorar. Sin embargo, es posible que el modelo aún tenga margen de mejora.

Algunas ideas para seguir mejorado incluyen (según chatGPT):

1. **Aumentar la profundidad del modelo**: Añadir más capas de convolución o aumentar el número de filtros puede ayudar a extraer características más complejas de las imágenes.
2. **Aumentar el tamaño de las imágenes**: Incrementar el tamaño de las imágenes de entrada a algo mayor que 112x112 podría proporcionar más detalles para el modelo, si el poder computacional lo permite.
3. **Data augmentation**: Aplicar técnicas como rotaciones, zoom, o cambios en el brillo a las imágenes de entrenamiento podría ayudar a generar más diversidad en los datos y reducir el riesgo de sobreajuste.
4. **Ajustar hiperparámetros**: Podrías experimentar con la tasa de aprendizaje, el tamaño del lote o probar otros optimizadores, como RMSprop o AdamW.
5. **Balancear las clases**: Si las clases están desbalanceadas, podrías utilizar técnicas como el ajuste del class_weight en la función de ajuste del modelo para dar más peso a las clases menos representadas.

Entrenamiento 3: balanceo de clases

Primero deben calcularse los pesos

```
# Calcular los pesos de las clases
# Calcular los pesos de las clases
# convertir las etiquetas que estan con one-hot encoded a una dimension
etiquetas_train_unidimensional = np.argmax(etiquetas_train, axis=1)

# Calcular los pesos de las clases con el array unidimensional
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(etiquetas_train_unidimensional),  # Etiquetas únicas
    y=etiquetas_train_unidimensional
)
```

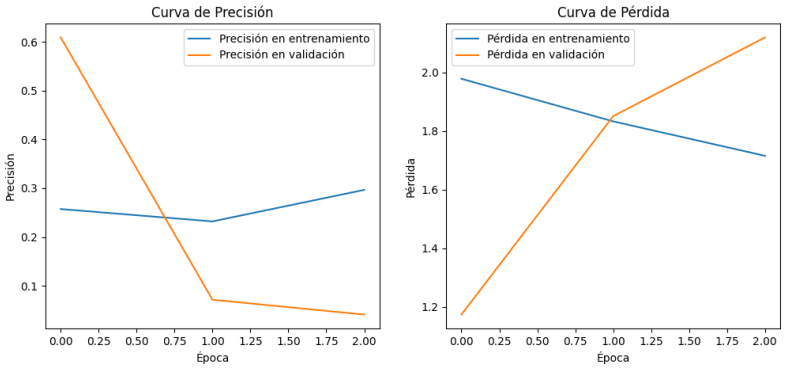
los pesos que se calcularon para las clases son:

```
Pesos de las clases: {0: 4.436323366555925, 1: 2.71869697997964, 2:
1.3140888961784485, 3: 13.155993431855501, 4: 1.2903849251087132, 5:
0.2132609332162155, 6: 9.459268004722551}

# Convertir los pesos en un diccionario que Keras pueda utilizar
class_weight_dict = dict(enumerate(class_weights))
history_cnn = model_cnn.fit(
    imagenes_train,
```

```
    etiquetas_train,
    epochs=15,
    batch_size=20,
    validation_data=(imagenes_test, etiquetas_test),
    class_weight=class_weight_dict,  # Agregar el balanceo de clases
    callbacks=[early_stopping]
)
```

Resultados obtenidos:



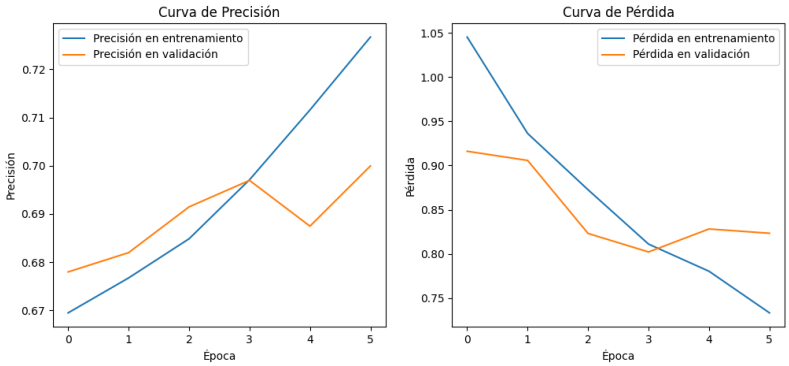
Esto no ha mejorado el entrenamiento.

El problema que estás viendo en los resultados, con una **exactitud baja en las primeras épocas** y una **pérdida elevada en los datos de validación**, sugiere que el modelo está experimentando dificultades para aprender correctamente con el balanceo de clases. La caída de la precisión de validación en la segunda época (0.0714) indica que el modelo puede estar enfrentando uno de estos problemas:

- **Posible problema Problema**: Asegúrate de que las clases estén balanceadas también en el conjunto de validación. De lo contrario, el modelo puede estar aprendiendo en un conjunto de entrenamiento balanceado, pero validando en uno desbalanceado.
- **Solución**: Asegúrate de usar la opción stratify=etiquetas_train en train_test_split, de modo que las proporciones de clases se mantengan en ambos conjuntos.

Volví para atrás lo de los pesos y modifiqué la función que separa el dataset en los subconjuntos para entrenamiento y validación (agregué el parámetro stratify)

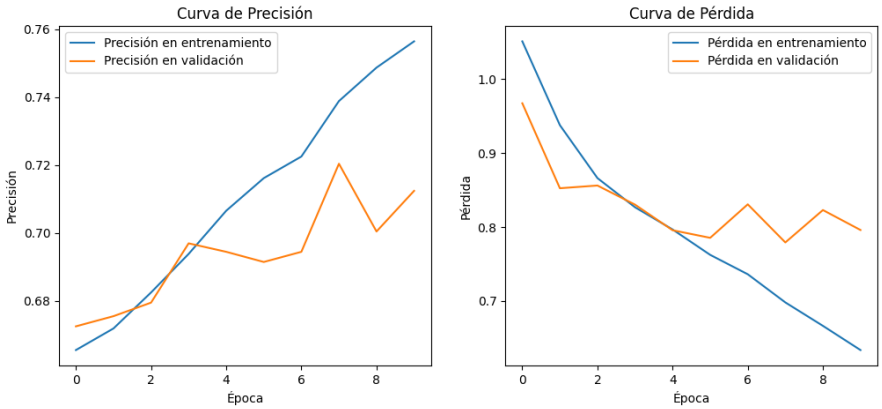
```
imagenes_train, imagenes_test, etiquetas_train, etiquetas_test = train_test_split(
    imagenes, etiquetas_dx, test_size=0.2, random_state=42, stratify=etiquetas_dx
)
```



Accuracy del modelo CNN en el conjunto de prueba: 0.6969545483589172

Se ejecutaron 6 épocas.

Ejecuté nuevamente el modelo para evaluar el rendimiento de tu modelo utilizando métricas adicionales como precisión, recall y F1-score. No se porque pero se ejecutaron más épocas y mejoró un poco la precisión. De todas maneras el modelo necesita ajustes o el dataset necesita ajustes



Métricas calculadas

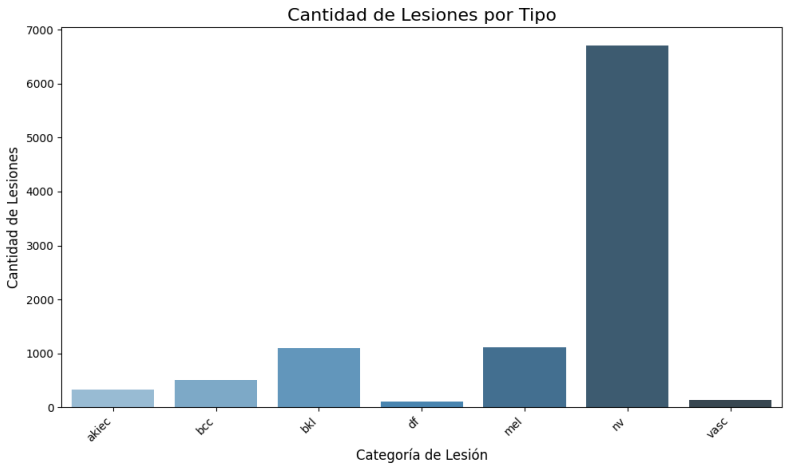
	precision	recall	f1-score	support
0	1.00	0.03	0.06	65
1	0.46	0.35	0.40	103
2	0.40	0.50	0.44	220
3	0.00	0.00	0.00	23
4	0.47	0.21	0.29	223
5	0.81	0.92	0.86	1341
6	0.80	0.43	0.56	28
accuracy			0.72	2003
macro avg	0.56	0.35	0.37	2003
weighted avg	0.70	0.72	0.69	2003

clase	etiqueta
0	bkl
1	nv
2	df
3	mel
4	vasc
5	bcc
6	akiec

La clase 5 es la que tiene mejor rendimiento. El modelo está identificando correctamente la mayoría de las instancias de esta clase, lo que sugiere que tiene suficientes datos y características discriminativas para esta categoría.

el 72% de todas las predicciones realizadas son correctas. Sin embargo, dado el desbalance en las clases, esta cifra puede ser engañosa.

La distribución de imágenes por tipo de lesión es la siguiente:

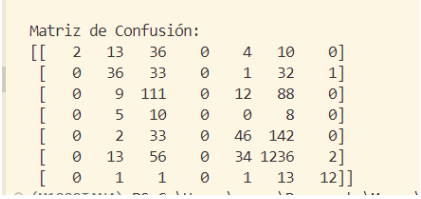


Esto muestra que la clase 5 (nv) tiene muchísimos más imágenes que el resto.

Conclusiones y Recomendaciones (según chatGPT)

1. **Problemas de Desbalance**: Las clases 0, 3, y 4 presentan problemas significativos. Es posible que necesites más datos para estas clases o considerar técnicas de aumento de datos.
2. **Mejorar la Detección**: Considera ajustar el modelo, como cambiar la arquitectura de la red, probar diferentes hiperparámetros, o aplicar técnicas de regularización.
3. **Uso de Métodos de Aumento de Datos**: Para clases con bajo rendimiento, el aumento de datos podría ayudar a mejorar la generalización del modelo.
4. **Revisar el Preprocesamiento**: Asegúrate de que el preprocesamiento de los datos esté bien alineado con las necesidades de tu modelo. Podría ser útil realizar una nueva ingeniería de características.
5. **Experimentar con Otras Architecturas**: Si es posible, prueba diferentes modelos o técnicas, como redes neuronales convolucionales más profundas o transfer learning, si cuentas con un dataset limitado.

Análisis Detallado de la Matriz de Confusión



```
Matriz de Confusión:
[[   2   13   36    0    4   10    0]
 [   0   36   33    0    1   32    1]
 [   0    9  111    0   12   88    0]
 [   0    5   10    0    0    8    0]
 [   0    2   33    0   46  142    0]
 [   0   13   56    0   34 1236    2]
 [   0    1    1    0    1   13   12]]
```

1. **Clases Confundidas**:
    - **Clase 0**: La mayoría de las predicciones incorrectas se asignan a la clase 1 (13 predicciones incorrectas) y la clase 2 (36 predicciones incorrectas). Esto sugiere que el modelo podría estar confundiendo las características de las clases 0, 1 y 2, lo que indica la necesidad de mejorar la diferenciación entre estas clases.
    - **Clase 1**: Similarmente, la clase 1 confunde más con la clase 2 (33 predicciones incorrectas) y tiene algunas confusiones con la clase 5 (32 predicciones incorrectas).
    - **Clase 2**: Presenta una confusión notable con la clase 1 (9 predicciones incorrectas) y la clase 5 (88 predicciones incorrectas). Esto sugiere que estas clases pueden compartir características similares, lo que complica la tarea de clasificación.
    - **Clase 3**: El modelo no clasifica correctamente instancias de esta clase, lo que indica que puede no tener suficientes datos representativos de esta clase o que sus características no son bien definidas en el modelo.
    - **Clase 4**: La clase 4 muestra confusión con las clases 2 y 5. Especialmente con la clase 5, donde se producen 34 errores, indicando que estas clases pueden tener características similares que el modelo no logra distinguir claramente.
    - **Clase 5**: Aunque tiene un buen rendimiento en general, confunde con la clase 1 (13 errores) y con la clase 4 (56 errores), lo que podría ser una preocupación a considerar en futuros ajustes.
    - **Clase 6**: Tiene confusiones con la clase 5, sugiriendo que estas dos clases podrían compartir algunas características comunes que el modelo no captura adecuadamente.
2. **Desempeño General por Clase**:
    - **Alto rendimiento en la clase 5**: El modelo está logrando un rendimiento notable en la clase 5, que tiene un número considerable de instancias (1341), lo que sugiere que esta clase es suficientemente representativa en el conjunto de entrenamiento.
    - **Bajo rendimiento en clases minoritarias**: Las clases como la 3 y 6 tienen pocos ejemplos y muestran una incapacidad del modelo para generalizar en esos casos, lo que puede ser un problema crítico si se requiere una clasificación confiable para todas las clases.
3. **Dificultades en la Clasificación Multiclase**:

- En problemas de clasificación multiclase con desequilibrios significativos, el modelo tiende a favorecer las clases con más instancias (en este caso, la clase 5), lo que puede resultar en un desempeño inadecuado para las clases menos representadas.
- La matriz de confusión refleja esto claramente, donde el modelo parece "predecir" la clase mayoritaria con frecuencia, afectando así las métricas de precisión y recall para las clases menos representadas.

4. Impacto en las Métricas:
    - Los errores de clasificación en clases críticas pueden afectar las métricas globales del modelo, como el F1-score y el recall. La clase 3, al no ser identificada, disminuye el rendimiento general y provoca que las métricas promedias se vean afectadas negativamente.
    - Las métricas de precisión y recall son una consecuencia directa de estos errores en la matriz de confusión, donde la precisión puede ser alta pero el recall bajo, reflejando la capacidad del modelo para detectar correctamente instancias en clases menos frecuentes.