



## **Procesamiento de Imágenes y Vision por Computadoras**

**Trabajo Practico Final**

**INFORME**

**Carrera: TUIA**

**Alumno:** Donnarumma, César Julián

**Ciclo:** 2024

## **Ejercicio 01: Creación de un dataset colaborativo.**

Para la primera parte del ejercicio se sacaron, como el pide el enunciado, 30 imágenes (31 en realidad) con entre 5 y 8 cartas y se le hicieron las respectivas anotaciones en formato yolo con la técnica de bounding box de carta parcial que acordaron en clases presenciales los compañeros.

El resultado de mi aporte fue subido en su momento al siguiente link:

[https://drive.google.com/drive/folders/1FTsIU6-eaKGB-qDcZBYKLXg\\_OgVypzp?usp=drive\\_link](https://drive.google.com/drive/folders/1FTsIU6-eaKGB-qDcZBYKLXg_OgVypzp?usp=drive_link).

## **Ejercicio 02: Dataset personal.**

### **Descarga de las imágenes:**

En primer lugar cabe mencionar que del dataset colaborativo compuesto por imágenes + anotaciones confeccionado por la totalidad del curso solo se utilizaron las imágenes.

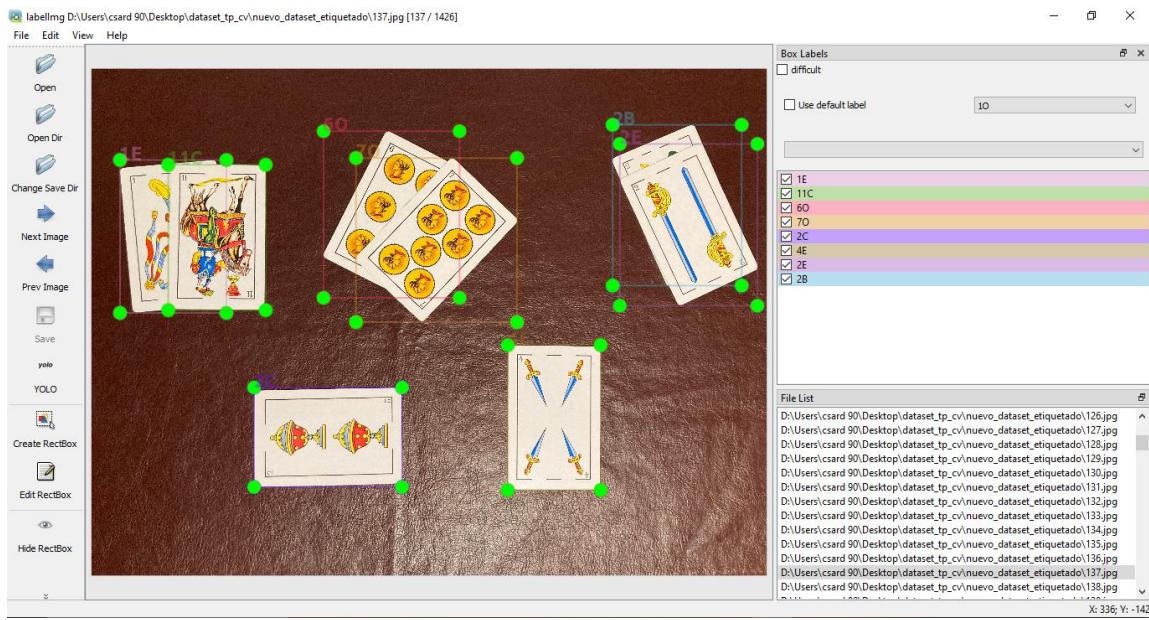
Se procedió a descargar manualmente en local por navegador una a una las carpetas de todos los alumnos desechariendo las anotaciones y renombrando todas las imágenes con números. En total quedaron 1434 imágenes sin etiquetar.

El motivo de lo anterior es que como fue mi primera experiencia entrenando un detector de objetos quería tener control absoluto de todo el proceso, incluso de la técnica de bounding box, y mis compañeros se habían decantado por hacerla a carta parcial, cosa que yo no quería. Sumado a esto al verificar los .txt muchos utilizaron formato yolo para segmentación con 8 coordenadas y otros usaron un orden distinto de clases en el classes.txt. Por supuesto esta decisión fue consultada por el profesor de teoría que dio el visto bueno para que así lo haga.

El siguiente paso fue conformar un grupo de 3 alumnos más que estaban en la misma situación que yo, ponernos de acuerdo en cuanto a criterios de cómo etiquetar y dividir en 3 la totalidad de las imágenes.

### **Etiquetado:**

La anotación de los datos se hizo en local a través **labelImg** instalado en un entorno virtual de **Anaconda**. Cada uno de los 3 compañeros etiqueto su parte y envió al resto su tarea. Luego de esto todos tenían en su totalidad el dataset con las imágenes de todo el curso anotado bajo un mismo criterio, con sus respectivos .txt en el mismo formato (4 coordenadas) y utilizando el mismo orden de classes.txt. Como eran pocas imágenes las revise rápidamente una a una para asegurarme que no haya errores, descarte 4 porque tenían una excesiva cantidad de cartas y así di la tarea de anotación por concluida.



Se subió a Google Drive todo el nuevo dataset y a partir de ese momento se empezó a trabajar en Google Colab en la notebook “*TUIA\_Computer\_Vision-TP\_FINAL\_Ejercicio\_02.ipynb*” conectada a Drive para dejar registro en código del resto del proceso.

### Validación de anotaciones:

Era hora de validar las anotaciones como la consigna pedía verificando que no haya puntos de bounding box fuera de la imagen y que ningún .txt contenga una clase que no corresponda a una carta.

```
[ ] print(f'Cantidad de imágenes movidas: {int(len(os.listdir(ruta_label_errors))/2)}')
→ Cantidad de imágenes movidas: 66
• En total se movieron 66 imágenes cuyo bb estaba fuera de la imagen (en general por muy poco)
```

Como resultado de esto se movieron 66 imágenes a una carpeta llamada “label\_errors” como el enunciado lo pedía.

### Partición de los datos en Train, Validación y Test:

Mediante código se creó en Drive la estructura de directorios necesaria para entrenar en Colab, se partió el dataset en 70-15-15 con *train\_test\_split* de **sklearn** y se movieron las imágenes con **os** y **shutil** a la carpeta correspondiente. El resultado fue: 957 instancias de train, 205 de validación y 206 de test.

```
-----  
train:  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/images/train: 957 archivos  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/labels/train: 957 archivos  
-----  
val:  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/images/val: 205 archivos  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/labels/val: 205 archivos  
-----  
test:  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/images/test: 206 archivos  
/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/data/labels/test: 206 archivos  
-----
```

Para concluir con esta parte de dejar la estructura de dataset lista para entrenar se creó con un script simple el *dataset.yaml* y se lo guarda donde tiene que estar.

```
▶ # Crear el archivo dataset.yaml  
yaml_content = f"""  
path: {ruta_datos_particionados}  
train: images/train  
val: images/val  
test: images/test  
  
nc: {len(clases.split())}  
names: {clases.split()}  
"""  
  
yaml_path = os.path.join(ruta_datos_particionados, "dataset.yaml")  
with open(yaml_path, "w") as f:  
    f.write(yaml_content)
```

## Aumentación:

Para las aumentaciones se utilizó la librería **albumentations**. Previamente hubo que aprender a utilizarla. Para esto se leyó la documentación oficial y se miraron los videos de clase de teoría de la materia donde se la ha utilizado.

La primera aproximación fue generar una instancia aumentada por cada instancia de entrenamiento.

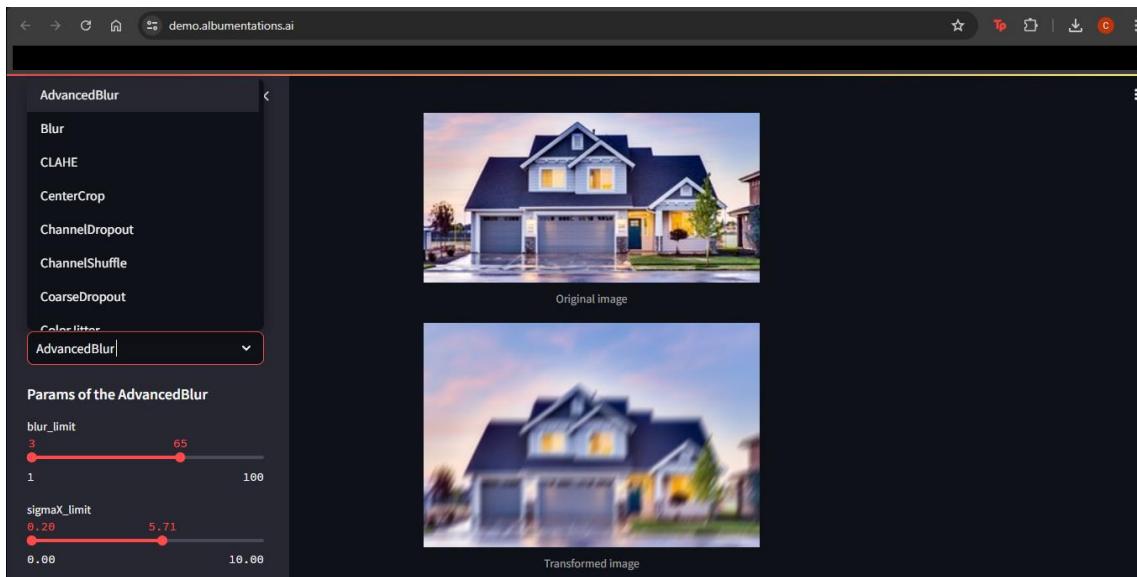
```

# Creamos el pipeline de transformaciones
transform = A.Compose([
    # Transformaciones a nivel pixel
    A.OneOf([
        # Cuando usamos OneOf ponemos una p de OneOf de que se entre en este bloque y las p de cada uno de los elementos del interior en 1 así el que se elije
        A.CLAHE(always_apply=False, p=1.0, clip_limit=(1, 45), tile_grid_size=(47, 52)), # Aplica ecualización de histograma adaptativo limitado a la imagen de entrada
        A.RandomBrightnessContrast(always_apply=False, p=1.0, brightness_limit=(-0.2, 0.48), contrast_limit=(-0.08, 0.51), brightness_by_max=True) # Ajusta aleatoriamente el brillo y el contraste
    ], p=0.5),
    A.OneOf([
        A.Blur(always_apply=False, p=1.0, blur_limit=(3, 50)), # Desenfoca la imagen de entrada utilizando kernel de tamaño aleatorio
        A.Defocus(always_apply=False, p=1.0, radius=(3, 3), alias_blur=(0.0, 0.0)), # Aplica desenfoque
        A.GaussNoise(always_apply=False, p=1.0, var_limit=(271.28, 480.5), per_channel=True, mean=45.39), # Aplica ruido Gaussiano a la imagen de entrada
        A.ISONoise(always_apply=False, p=1.0, intensity=(0.6, 1.06), color_shift=(0.21, 0.33)), # Aplica ruido de sensor de cámara
        A.MotionBlur(always_apply=False, p=1.0, blur_limit=(9, 19), allow_shifted=True), # Aplica desenfoque de movimiento
        A.Sharpen(always_apply=False, p=1.0, alpha=(0.55, 0.86), lightness=(0.32, 2.13)) # Enfoca la imagen
    ], p=0.4),
    A.OneOf([
        A.ChannelDropout(always_apply=False, p=1.0, channel_drop_range=(1, 1), fill_value=117), # Elimina canales aleatoriamente en la imagen de entrada
        A.ChannelShuffle(always_apply=False, p=1.0), # Mezcla aleatoriamente los canales de la imagen de entrada
        A.Downscale(always_apply=False, p=1.0, scale_min=0.31, scale_max=0.39, interpolation=cv2.INTER_LINEAR), # Disminuye la calidad de la imagen
        A.Emboss(always_apply=False, p=1.0, alpha=(0.27, 0.46), strength=(2.55, 3.97)), # Realiza la imagen de entrada y superpone el resultado con la imagen original
        A.MultiplicativeNoise(always_apply=False, p=1.0, multiplier=(1.66, 2.39), per_channel=True, elementwise=True), # Multiplica los pixeles por un número aleatorio
        A.ToGray(always_apply=False, p=1.0) # Pasa la imagen a blanco y negro
    ], p=0.3),
    A.OneOf([
        A.PixelDropout(always_apply=False, p=1.0, dropout_prob=0.12, per_channel=0, drop_value=(0, 0, 0), mask_drop_value=None), # Señala algunos pixeles a 0
        A.Spatter(always_apply=False, p=1.0, mean=(2.55, 2.55), std=(78.72, 78.72), gauss_sigma=(3.9, 3.9), intensity=(-1.28, -1.28), cutout_threshold=(-3.9, -3.9))
    ], p=0.4),
    # Transformaciones a nivel espacial
    A.OneOf([
        A.Flip(always_apply=False, p=0.1), # Da vuelta la imagen vertical u horizontalmente u ambas
        A.Rotate(always_apply=False, p=1.0, limit=(0, 360), interpolation=3, border_mode=0, value=(0, 0, 0), mask_value=None, rotate_method='largest_box', crop_border=0)
    ], p=0.7)
], bbox_params=A.BboxParams(format='yolo', min_area=area_minima_to_numpy([0])))

```

El bloque principal de código es un pipeline `Compose()` compuesto por varios `OneOf()` donde cada uno incluye transformaciones distintas más o menos del mismo tipo para que no se hagan dos transformaciones similares. Cabe destacar que los distintos `OneOf()` tienen probabilidades distintas de ocurrencia, pero una vez que se entra en alguno todas las transformaciones contenidas tienen probabilidad 1 con el objetivo de que la escogida se haga sí o sí.

Para la elección de transformaciones y de parámetros de las mismas se utilizó la página con la demo: <https://demo.albumentations.ai/>.



La dinámica fue básicamente recorrer una a una cada transformación disponible para ver de qué trataba moviendo manualmente los parámetros. Como arriba se vio se armaron grupos de transformaciones similares evitando elegir aquellas que deformen la imagen, ya que se quería evitar deformar las cartas. Luego de cada prueba se copió el bloque de código que la página genera abajo.

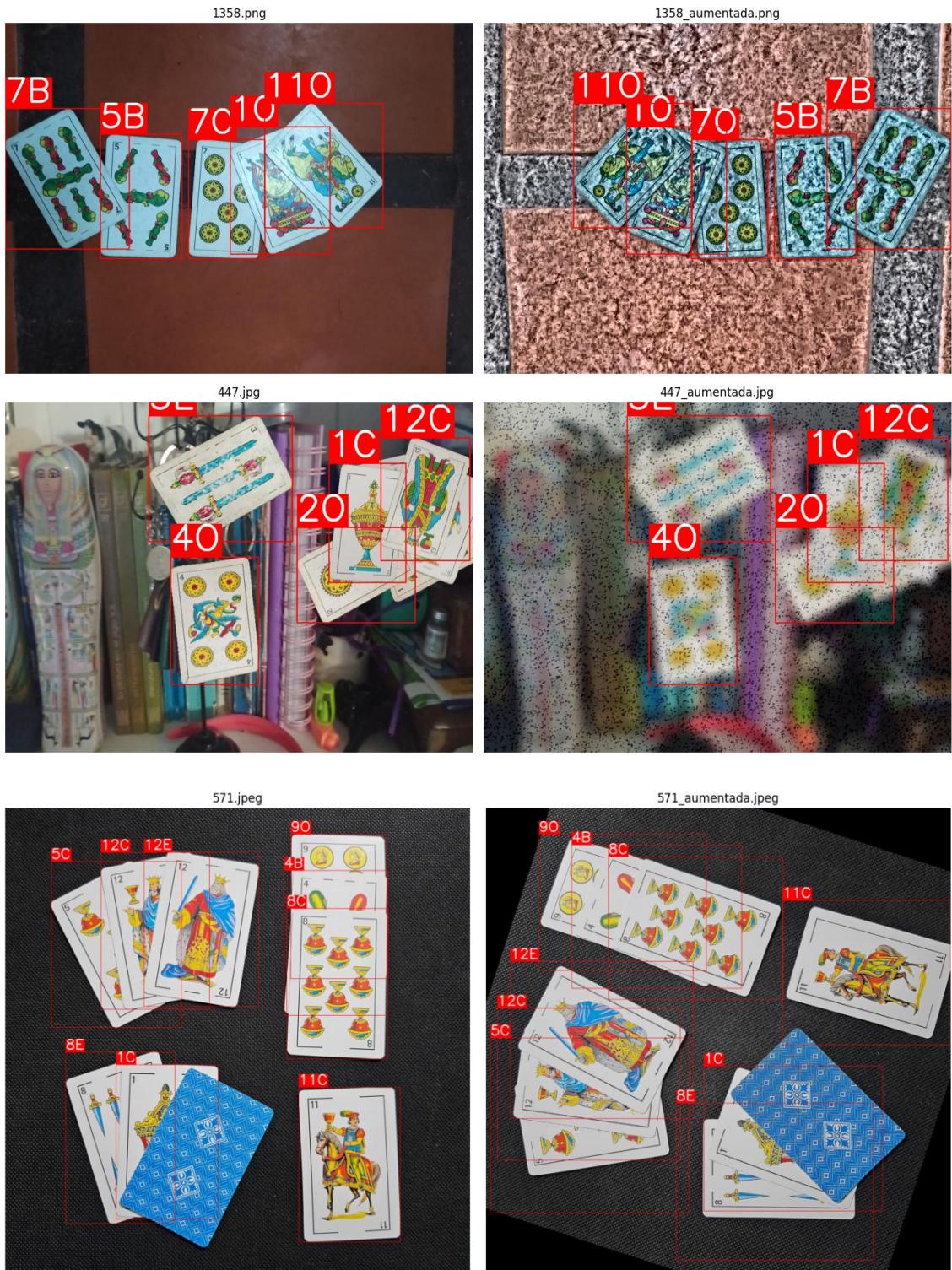
```
uint8, float32
AdvancedBlur(always_apply=False, p=1.0, blur_limit=(3, 65), sigma_x_limit=(0.2, 5.
```

Para la ejecución simplemente se corrió un *for* recorriendo una a una las instancias de train, se cargó la imagen con **OpenCV** en formato RGB, se leyeron las anotaciones y reorganizaron de la manera que Albumentations las necesita para yolo (x\_center, y\_center, w, h, class) y se le paso al pipeline declarado arriba la imagen cargada y el bb reacomodado. Luego solo fue cuestión de reacomodar el nuevo bb de la imagen aumentada a formato yolo de nuevo (class, x\_center, y\_center, w, h), renombrar los nuevos archivos y guardarlos en el directorio correspondiente de entrenamiento.

Al final de todo este proceso contaba con 1914 instancias de entrenamiento.

```
Σ -----  
Train:  
-----  
Cantidad de imagenes: 1914  
Cantidad de labels: 1914  
-----  
  
• Antes teniamos 957 imagenes y labels de train, ahora tenemos 1914, es decir, el doble.
```

Los siguientes son algunos de los ejemplos de las imágenes originales junto a sus imágenes aumentadas.

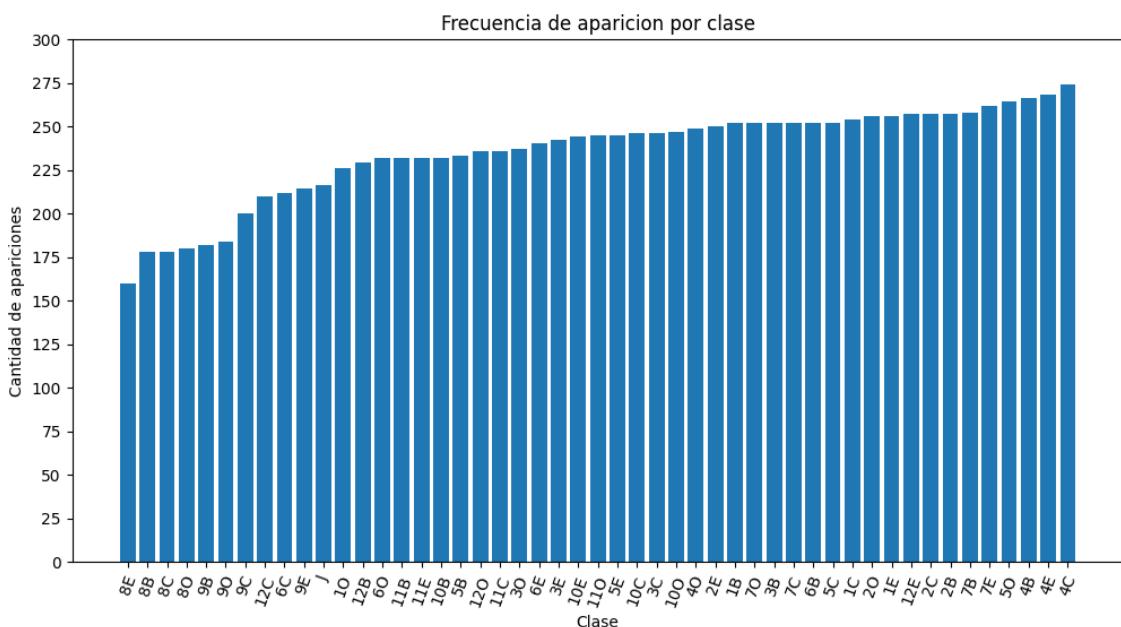


En líneas generales parece que la aumentación funcionó bastante bien, se respetaron los bounding box y las clases. En algunos casos donde hubo rotación quizás los bb no quedaron tan ajustados como estaban los originales pero salvo por ese detalle parecen haber quedado bien.

## Composición del dataset en términos del balanceo de clases:

	frecuencia_absoluta	frecuencia_relativa
count	49.000000	49.000000
mean	235.387755	2.040408
std	27.063056	0.234414
min	160.000000	1.390000
25%	229.000000	1.990000
50%	245.000000	2.120000
75%	252.000000	2.180000
max	274.000000	2.380000

```
[ ] print(f"Cantidad de cartas en total: {tabla_conteo_clases['frecuencia_absoluta'].sum()}")
→ Cantidad de cartas en total: 11534
```



En total quedaron 11534 cartas en todo el dataset.

El dataset quedó bastante balanceado.

En términos absolutos entre 160 y 274 imágenes por clases.

La cartas menos frecuentes son todos los 8s y las más frecuente el 4 de copas.

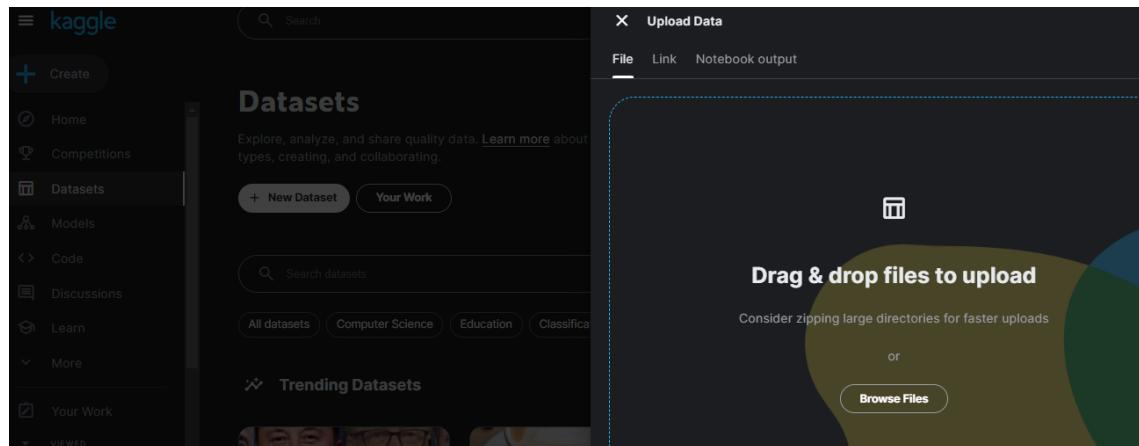
En términos relativos todas las clases tenían entre el 1.39% del total y el 2.38%. Es una proporción bastante parecida. No parece tener demasiado sentido tener que balancear en esta situación.

## Ejercicio 03: Entrenamiento de modelos.

Con todo lo anterior ya realizado primero se intentó entrenar en Colab un yolov8n con una GPU T4, pero Google no te deja estar mucho mas de 1:30hs conectado y en general el modelo solo en levantar las imágenes tardaba 30 minutos, así que entrenarlo de este modo me resulto imposible.

La otra alternativa fue usar **Kaggle** para entrenar, esta plataforma te presta 2 GPU T4s por 30 hs semanales que las administras como vos quieras pudiendo estar de corrido las que quieras. Así que resulto la mejor opción dado el caso.

Previamente hubo que descargar a local el dataset aumentado de Colab, comprimirlo en un .rar y para poder utilizarlo en una notebook fue necesario cargarlo como un Dataset Kaggle desde la página.



Esto debido al gran peso del mismo (12 GB) tomo bastante tiempo. Al igual que el paso previo de la descarga de Colab.

The screenshot shows the Kaggle web interface with the title "Your Datasets (5)". On the left, there's a sidebar with navigation links like Home, Competitions, Datasets, Models, Code, Discussions, Learn, and More. Under "Your Work", there are links for "VIEWED" and "TUIA CV - Trabajo ...". Below these are "View Active Events". The main area displays five datasets:

- 0 selected
- Private - Usability 1.2 - 262 MB
- TUIA\_CV\_TP\_Yolov8n\_Aumentacion\_x2  
csharp90 Updated 9 days ago
- TUIA\_CV\_TP\_Yolov8n\_Aumentacion\_x1  
csharp90 Updated 9 days ago
- Private - Usability 1.2 - 21 MB
- TUIA\_ComputerVision\_Dataset\_Aumentado\_x2  
csharp90 Updated 10 days ago
- Private - Usability 1.3 - 18 GB
- TUIA\_ComputerVision\_Dataset\_Aumentado\_x1  
csharp90 Updated 11 days ago
- Private - Usability 1.3 - 12 GB

El siguiente paso fue aprender a usar la notebook de Kaggle ya que tiene una estructura de directorio distinta y una interfaz algo diferente, pero no fue una tarea muy difícil.

El dataset que cargamos antes se levanta en la parte de los inputs que es un espacio en el que solo tenemos permiso de lectura.

The screenshot shows a Kaggle Notebook interface. At the top, there are buttons for "+", "X", "Run All", "Code", and a dropdown menu. Below this is a code cell with a play button and a "Draft Session off (run a cell to start)" message. To the right is a "Notebook" panel with tabs for "Input" and "Datasets". The "Input" tab shows "+ Add Input" and "+ Upload" buttons. The "Datasets" tab shows a file tree for "dataset-tp":

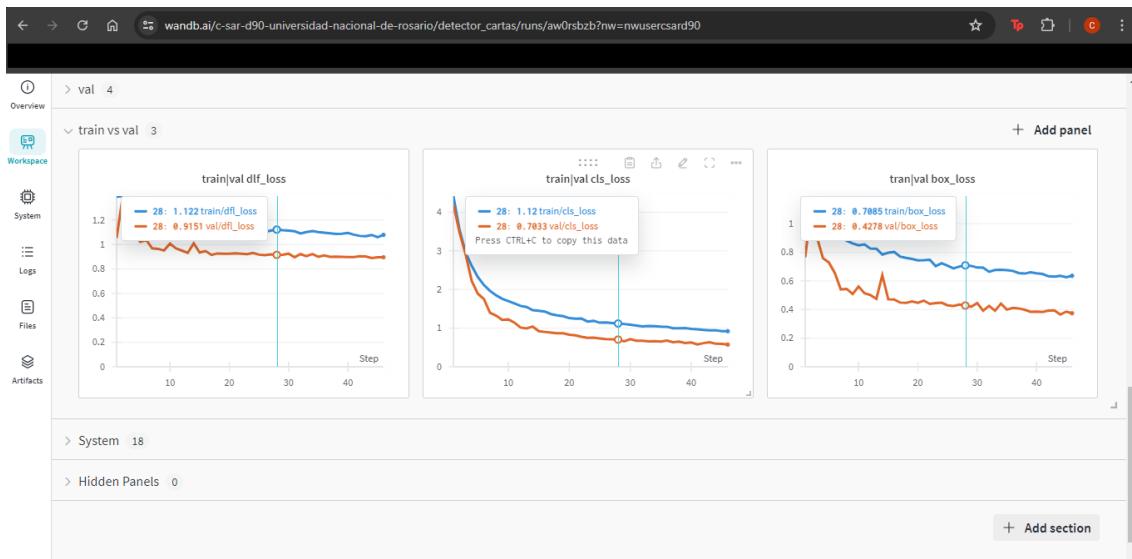
- dataset-tp
  - data
    - images
      - test
      - train
      - val
    - labels
      - test
      - train
      - val
        - train.cache
        - val.cache
  - dataset.yaml

Antes de poder entrenar fue necesario crear un nuevo *dataset.yaml* y ubicarlo en la parte de los outputs la notebook ya que el archivo creado anteriormente se encuentra en el input y como mencione tiene permiso de solo lectura.

```
In [7]:  
# Ruta donde esta guardado el dataset  
ruta_datos_particionados = '/kaggle/input/dataset-final-tp-cv/data'  
  
# Estructura del yaml  
yaml_content = f"""  
path: {ruta_datos_particionados}  
train: images/train  
val: images/val  
test: images/test  
  
nc: {len(clases.split())}  
names: {clases.split()}  
  
cache_dir: /kaggle/working  
"""  
  
# Ruta donde lo vamos a guardar (en nuestra notebook)  
yaml_path = os.path.join('/kaggle/working/', "dataset.yaml")  
  
# Guardamos el yaml  
with open(yaml_path, "w") as f:  
    f.write(yaml_content)
```

```
In [8]:  
# Chequeamos el contenido  
with open(yaml_path, 'r') as f:  
    yaml_content = f.read()  
  
print(yaml_content)  
  
path: /kaggle/input/dataset-final-tp-cv/data  
train: images/train  
val: images/val  
test: images/test  
  
nc: 49  
names: ['10', '1C', '1E', '1B', '20', '2C', '2E', '2B', '30', '3C', '3E', '3B', '40', '4C', '4E', '4B', '50', '5C', '5E', '5B', '60', '6C', '6E', '6B', '70', '7C', '7E', '7B', '80', '8C', '8E', '8B', '90', '9C', '9E', '9B', '100', '10C', '10E', '10B', '110', '11C', '11E', '11B', '120', '12C', '12E', '12B', 'J']  
  
cache_dir: /kaggle/working
```

En todos los entrenamientos siguientes se monitorearon en vivo las métricas en **wandb.ai**.



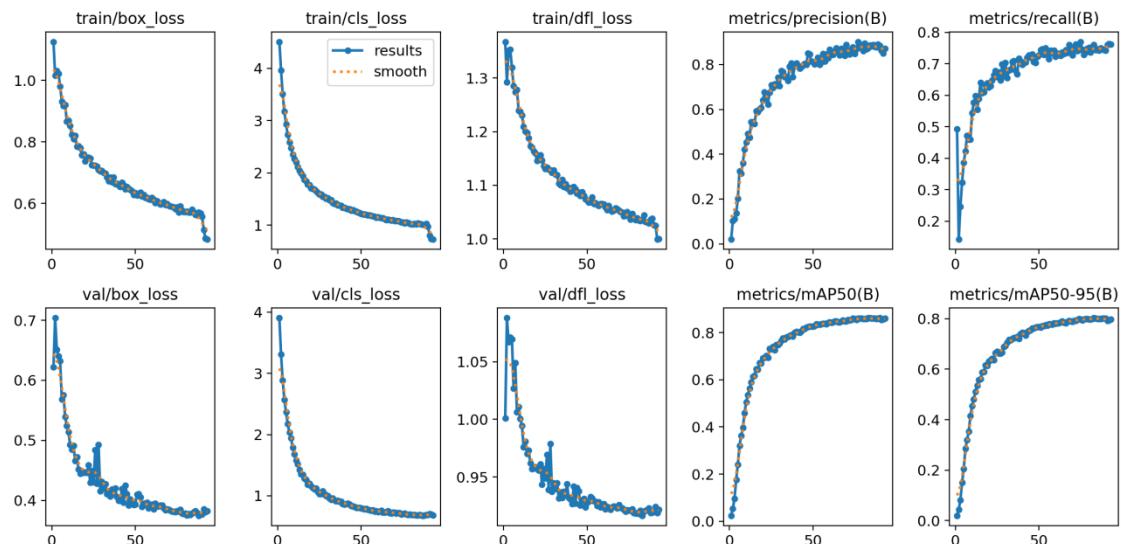
## Primer modelo:

La primer idea para resolver el problema, y a modo de “prueba”, fue la de entrenar un yolov8n con el dataset (con una sola aumentación por instancia) y ver como resultaba, para a partir ver que hacer.

```
In [12]: # Cargamos un modelo pre-entrenado para continua entrenandolo en nuestro caso especifico (recomendado)
# Entrenamos el modelo
model = YOLO("yolov8n.pt")

# Entrenamos el modelo
results = model.train(
    data=yaml_path, # Le pasamos la ruta donde esta nuestro yaml (en la notebook con los datos referenciando al dataset)
    epochs=100,
    imgsz=640,
    patience=10, # earlystopping en 10 epochas
    plots=True,
    project='detector_cartas', # Nombre del proyecto
    name='modelo_01_entrenamiento_01'
)
```

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size
93/100	2.33G	0.4837	0.7312	1	56	640: 100%  ████████
120/120 [02:09<00:00, 1.08s/it]						
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95): 100%
	██████	7/7 [00:08<00:00, 1.23s/it]				
	all	205	1227	0.871	0.762	0.86 0.797
<b>EarlyStopping:</b> Training stopped early as no improvement observed in last 10 epochs. Best results observed at epoch 83, best model saved as best.pt.						
To update EarlyStopping(patience=10) pass a new patience value, i.e. 'patience=300' or use 'patience=0' to disable EarlyStopping.						
93 epochs completed in 3.690 hours.						
Optimizer stripped from detector_cartas/entrenamiento_012/weights/last.pt, 6.3MB						
Optimizer stripped from detector_cartas/entrenamiento_012/weights/best.pt, 6.2MB						



### Observaciones:

- El early stopping detuvo el entrenamiento en la época 93 luego de 3.69 hs.
- De las pérdidas de la última época:

train/box_loss	0.48367
train/cls_loss	0.73124
train/dfl_loss	1.00004
val/box_loss	0.38259
val/cls_loss	0.68348
val/dfl_loss	0.92193

Se puede decir que en todos los casos las de validación fueron inferiores (mejores) a las de entrenamiento lo que sugiere que no hubo overfitting durante el entrenamiento.

- Sobre el final del log de entrenamiento se corrió una validación sobre el mejor modelo `best.pt`:

```

Validating detector_cartas/entrenamiento_012/weights/best.pt...
Ultralytics YOLOv8.2.5.8 🚀 Python-3.10.13 torch-2.1.2 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 168 layers, 3,015,203 parameters, 0 gradients, 8.1 GFLOPs

          Class    Images Instances     Box(P)      R      mAP50    mAP50-95 : 100%
          | 7/7 [00:14<00:00,  2.08s/it]

          all     205     1227   0.873   0.759   0.862   0.803
          10      35      35    0.964   0.766   0.9     0.827
          1C      20      20    0.826   0.85    0.87   0.849
          1E      26      26    0.952   0.923   0.958   0.888
          1B      33      33    0.972   0.788   0.89   0.826
          20      26      26    0.957   0.849   0.968   0.868
          2C      22      22    0.909   0.906   0.92   0.838
          2E      20      20    0.974   0.9     0.969   0.923
          2B      25      25    0.857   0.721   0.877   0.778
          30      24      24    0.954   0.862   0.957   0.866
          3C      27      27    0.926   0.815   0.969   0.924
          3E      26      26    0.852   0.923   0.911   0.854
          3B      27      28    0.918   0.8     0.906   0.868
          40      29      29    0.846   0.758   0.879   0.822
          4C      19      19    0.787   0.684   0.785   0.698
          4E      27      27     1    0.829   0.939   0.873
          4B      28      28     1    0.881   0.95    0.862
          50      30      30    0.882   0.867   0.91    0.849
          5C      25      25    0.829   0.8     0.829   0.771
          5E      41      41    0.979   0.927   0.98    0.939
          5B      29      29    0.991   0.862   0.917   0.82
          60      21      21    0.665   0.663   0.738   0.693
          6C      24      24    0.954   0.583   0.781   0.735
          6E      26      26     1    0.742   0.923   0.859
          6B      25      25    0.981   0.88    0.945   0.895
          70      30      30    0.824   0.6     0.782   0.739
          7C      24      25    0.78    0.76    0.889   0.764
          7E      20      20    0.805   0.75    0.852   0.835
          7B      28      28    0.928   0.786   0.904   0.835
          80      17      17     0.8    0.786   0.796   0.762
          8C      21      21    0.767   0.667   0.776   0.73
          8E      29      30    0.917   0.74    0.922   0.838
          8B      28      28    0.829   0.864   0.914   0.839
          90      26      26    0.908   0.762   0.913   0.866
          9C      14      14    0.864   0.909   0.928   0.884
          9E      14      14    0.729   0.857   0.897   0.826
          9B      27      27    0.928   0.852   0.918   0.841
          100     20      20     0.91    0.75    0.832   0.795
          10C     17      17    0.779   0.588   0.701   0.674
          10E     24      24    0.851   0.458   0.692   0.646
          10B     26      27    0.853   0.647   0.797   0.722
          110     25      25    0.908   0.68    0.844   0.8
          11C     25      25    0.834   0.72    0.817   0.771
          11E     25      25    0.826   0.57    0.784   0.712
          11B     35      35    0.892   0.543   0.762   0.718
          120     16      16    0.539   0.625   0.676   0.581
          12C     25      25    0.666   0.56    0.68    0.638
          12E     21      21    0.828   0.762   0.864   0.809
          12B     30      30    0.835   0.7     0.831   0.779
          J       19      21     1    0.742   0.872   0.83
Speed: 0.2ms preprocess, 2.4ms inference, 0.0ms loss, 1.1ms postprocess per image
Results saved to detector_cartas/entrenamiento_012

```

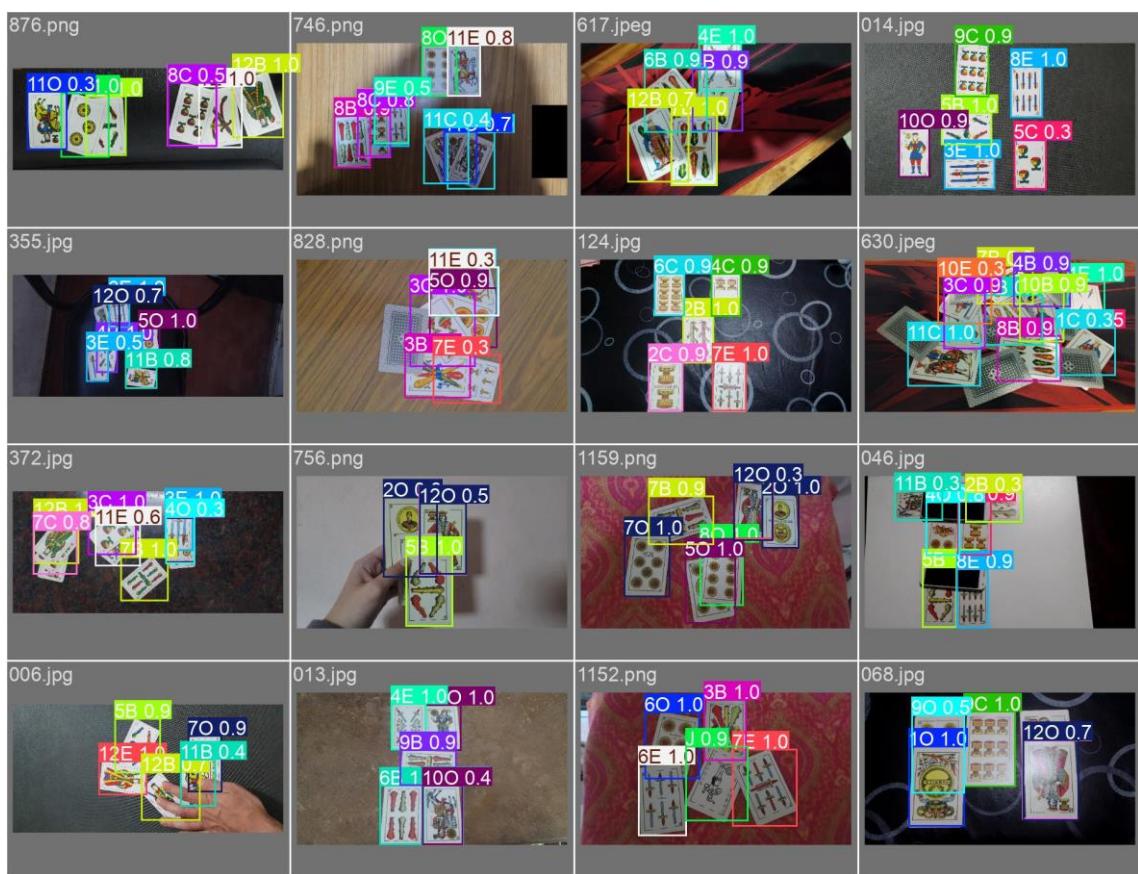
Los valores para todo el conjunto (all) fueron de mAP50=0.862, mAP50-95=0.803.

A simple vista para un primer intento no parecen malas métricas pero se cree que podrían mejorarse.

En la evaluación clase por clase las que peores performan son: 6O, 10C, 10E, 12C, 12O todas con un mAP50 inferior a 0.75 y un mAP50-95 inferior a 0.70.

Observando el grafico de distribución de clases realizado previamente a entrenar llama la atención que ninguna de estas clases sea de las q menos instancias tiene (lo que afirma la teoría de que el pequeño desbalanceo que había era menor).

- También cabe agregar que las gráficas de perdida de validación mostraban lo que parecería ser una estabilización de la curva, sin embargo las gráficas en train tendían a continuar bajando por lo que si seguimos entrenando podríamos terminar en zona de overfitting.
- *Por lo tanto con el objetivo de mejorar un poco las métricas se determinó realizar una aumentación x2 (dos imágenes por cada imagen de train) y re-entrenar un nuevo modelo (pre-entrenado).*



### Nueva aumentación:

Si bien a esta altura del informe nos encontramos explicando el proceso relacionado al entrenamiento de los modelos la historia entre cada uno de los notebooks no fue lineal. Luego del primer entrenamiento como mencione más arriba se determinó realizar una aumentación más por cada una de las 957 instancias de entrenamiento iniciales. Así que se retomó el notebook “TUIA\_Computer\_Vision-TP\_FINAL\_Ejercicio\_02.ipynb” en Colab y se realizó el mismo procedimiento que antes. La única salvedad fue la forma en la que se trabajó, debido a las limitaciones de espacio tenía ocupados 12GB/15GB así que tuve que borrar de Drive las imágenes de las primeras aumentaciones (los labels los dejé para evaluar el balanceo de

nuevo), con código generar las nuevas y sus labels en otra carpeta separada, descargar a local las nuevas aumentaciones y juntarlas con el dataset que use anteriormente. Los nuevos labels los junta en el Drive con los anteriores para analizar la nueva composición del dataset.

```
[ ] print(f'Cantidad de labels: {len(os.listdir(ruta_labels_train))}')
```

→ Cantidad de imagenes: 2871

Ahora contaba con 2871 imágenes de entrenamiento.

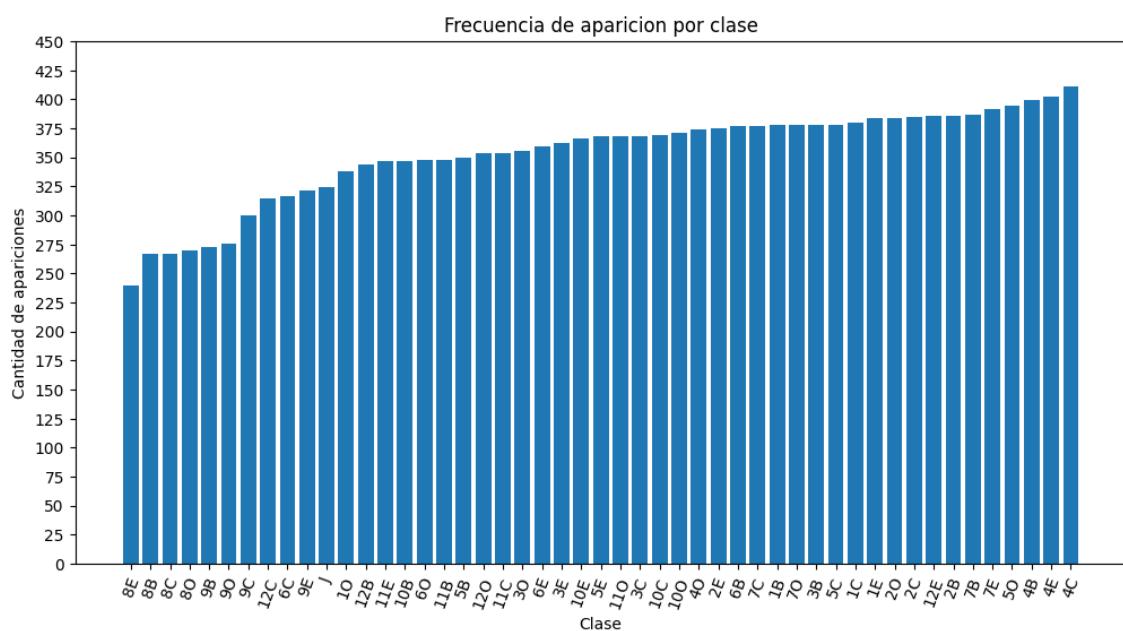
```
[ ] print(f"Cantidad de cartas en total: {tabla_conteo_clases['frecuencia_absoluta'].sum()}")
```

→ Cantidad de cartas en total: 17293

En total en todas las imágenes había 17293 cartas.

	frecuencia_absoluta	frecuencia_relativa
count	49.000000	49.000000
mean	352.918367	2.041224
std	40.551838	0.235174
min	240.000000	1.390000
25%	344.000000	1.990000
50%	368.000000	2.130000
75%	378.000000	2.190000
max	411.000000	2.380000

Las proporciones se mantuvieron igual.



El dataset continuaba relativamente balanceado.

Luego de esto mismo accionar, crear un Dataset Kaggle y ya estaba listo para usarse de nuevo en el notebook de entrenamiento de Kaggle.

### Segundo modelo:

El primer intento con el nuevo conjunto de imágenes seria con el mismo modelo yolov8n. A esta altura de la realización del trabajo ya se tenía mayor familiaridad con la documentación de Yolo por lo que también se aprovechó que Kaggle nos dio las 2 T4 para entrenar con ambas.

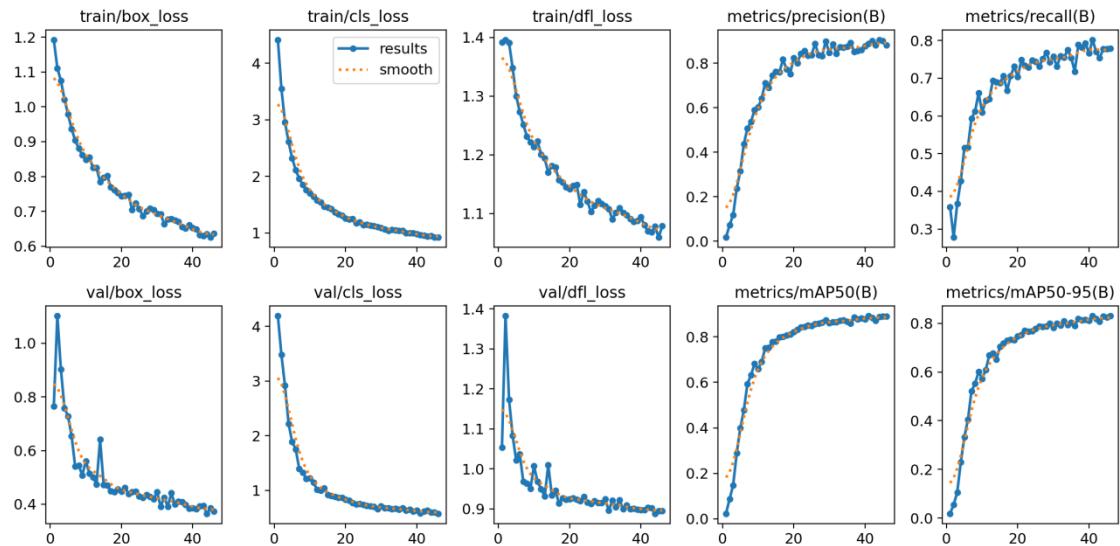
```
# Vamos a probar con el modelo mas grande a ver que resultados da
model = YOLO("yolov8n.pt")

# Entrenamos el modelo
results = model.train(
    data=yaml_path, # Le pasamos la ruta donde esta nuestro yaml (en la notebook con los datos refe
    renciando al dataset)
    epochs=100,
    imgsz=640,
    patience=5, # earlystopping en 10 epochas
    plots=True,
    project='detector_cartas', # Nombre del proyecto
    name='modelo_02_entrenamiento_01', # Nombre del modelo
    device=[0, 1], # Para entrenar con las 2 T4 que nos da Kaggle
    batch=64,
    cache='True'
)
```

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size
46/100	4.75G	0.6357	0.9213	1.079	337	640: 100%  ██████████
45/45 [03:24<00:00, 4.54s/it]						
	Class	Images	Instances	Box(P	R	mAP50 mAP50-95): 100%
	██████████	4/4 [00:14<00:00, 3.55s/it]				
	all	205	1227	0.882	0.779	0.891 0.83

**EarlyStopping:** Training stopped early as no improvement observed in last 5 epochs. Best results observed at epoch 41, best model saved as best.pt.  
To update EarlyStopping(patience=5) pass a new patience value, i.e. 'patience=300' or use 'patience=0' to disable EarlyStopping.

46 epochs completed in 2.875 hours.  
Optimizer stripped from detector\_cartas/modelo\_02\_entrenamiento\_013/weights/last.pt, 6.2MB  
Optimizer stripped from detector\_cartas/modelo\_02\_entrenamiento\_013/weights/best.pt, 6.2MB



### Observaciones:

- El early stopping detuvo el entrenamiento en la época 46 luego de 2.88 hs.
- Perdidas de la última época:

```
wandb :      train/box_loss 0.63566
wandb :      train/cls_loss 0.92134
wandb :      train/dfl_loss 1.0786
wandb :      val/box_loss 0.37379
wandb :      val/cls_loss 0.57498
wandb :      val/dfl_loss 0.89573
```

Nuevamente mejores métricas de perdida en validación que en entrenamiento a lo largo de todo el entrenamiento. Se descarta overfitting.

Además las métricas de perdida de validación de este modelo son un poco más bajas que las del anterior lo que podría ser síntoma de un modelo que performa mejor en datos no vistos.

- Sobre el final del log de entrenamiento se corrió una validación sobre el mejor modelo best.pt:

```

Validating detector_cartas/modelo_02_entrenamiento_013/weights/best.pt...
Ultralytics YOLOv8.2.60 🚀 Python-3.10.13 torch-2.1.2 CUDA:0 (Tesla T4, 15095MiB)
                                         CUDA:1 (Tesla T4, 15095MiB)
Model summary (fused): 168 layers, 3,015,203 parameters, 0 gradients, 8.1 GFLOPs

```

Class	Images	Instances	Box(P)	R	mAP50	mAP50-95): 100%
	4/4 [00:22<00:00, 5.75s/it]					

all	205	1227	0.887	0.803	0.892	0.831
10	35	35	0.899	0.771	0.893	0.804
1C	20	20	0.854	0.8	0.814	0.775
1E	26	26	0.942	0.885	0.952	0.882
1B	33	33	0.915	0.818	0.9	0.823
20	26	26	0.987	0.923	0.966	0.906
2C	22	22	0.898	0.884	0.895	0.802
2E	20	20	0.846	1	0.986	0.947
2B	25	25	0.919	0.8	0.868	0.797
30	24	24	0.947	0.958	0.957	0.868
3C	27	27	0.958	0.845	0.941	0.87
3E	26	26	0.823	0.885	0.912	0.856
3B	27	28	0.847	0.821	0.892	0.85
40	29	29	0.894	0.759	0.845	0.794
4C	19	19	0.775	0.725	0.807	0.71
4E	27	27	0.991	0.815	0.937	0.883
4B	28	28	0.96	0.858	0.945	0.865
50	30	30	0.98	0.867	0.956	0.9
5C	25	25	0.649	0.8	0.85	0.797
5E	41	41	0.97	0.927	0.974	0.922
5B	29	29	1	0.9	0.951	0.853
60	21	21	0.771	0.804	0.854	0.803
6C	24	24	0.902	0.708	0.84	0.781
6E	26	26	1	0.779	0.932	0.824
6B	25	25	0.864	0.88	0.925	0.876
70	30	30	0.879	0.724	0.874	0.818
7C	24	25	0.974	0.76	0.9	0.865
7E	20	20	0.757	0.779	0.873	0.816
7B	28	28	0.926	0.893	0.916	0.845
80	17	17	0.77	0.706	0.797	0.747
8C	21	21	0.87	0.639	0.782	0.738
8E	29	30	0.905	0.8	0.946	0.841
8B	28	28	0.938	0.857	0.937	0.873
90	26	26	0.951	0.743	0.944	0.873
9C	14	14	0.805	1	0.948	0.881
9E	14	14	0.605	0.786	0.867	0.81
9B	27	27	0.924	0.906	0.95	0.89
100	20	20	0.951	0.85	0.916	0.871
10C	17	17	0.915	0.706	0.867	0.807
10E	24	24	0.883	0.629	0.861	0.773
10B	26	27	0.832	0.735	0.846	0.784
110	25	25	1	0.715	0.891	0.857
11C	25	25	0.752	0.8	0.872	0.838
11E	25	25	0.777	0.557	0.758	0.709
11B	35	35	0.826	0.771	0.853	0.807
120	16	16	0.922	0.74	0.897	0.802
12C	25	25	0.871	0.76	0.853	0.798
12E	21	21	0.994	0.762	0.845	0.816
12B	30	30	0.934	0.8	0.874	0.821
J	19	21	0.917	0.81	0.866	0.834

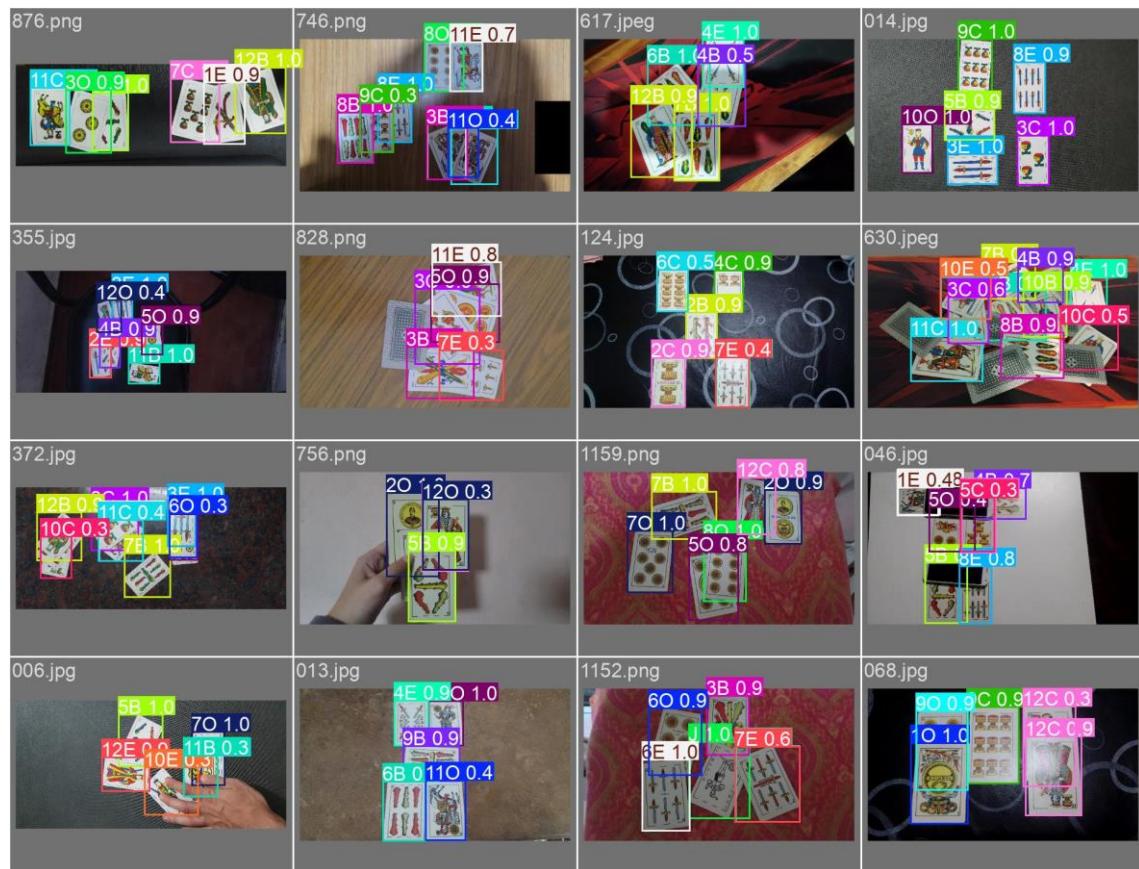
Speed: 0.3ms preprocess, 4.1ms inference, 0.0ms loss, 13.7ms postprocess per image

Results saved to `detector_cartas/modelo_02_entrenamiento_013`

Los valores para todo el conjunto (all) son de mAP50=0.892 y mAP50-95=0.831, superando por poco al modelo anterior.

En la evaluación clase por clase la performance en general ha aumentado también.

- Nuevamente las gráficas de perdida de validación mostraban lo que parecería ser una estabilización de la curva.
- Si bien se tenía conocimiento de que el hecho de aumentar aún más la cantidad de imágenes de entrenamiento podría derivar en una mejora aun mayor de las métricas, debido a que se estaba trabajando con Colab, en local y Kaggle de manera intercalada para cada uno de los pasos, y al tiempo que toma cada paso y llevar los datos de un lado al otro se descartó continuar por ese lado.
- Sin embargo como experimento y para ver si de ese modo mejoran se optó por entrenar con los últimos datos utilizados un modelo yolov8x (el más grande) a sabiendas de que el peso del modelo sería mucho mayor y la velocidad de inferencia mucho menor y también de que quizás en un ambiente de trabajo tomar esta solución puede resultar en algo no tan conveniente dependiendo de las circunstancias.



### Tercer modelo:

Esta fue la última prueba de modelo que se realizó debido al tiempo que quedaba para continuar experimentando con el trabajo práctico.

```

# Vamos a probar con el modelo mas grande a ver que resultados da
model = YOLO("yolov8x.pt")

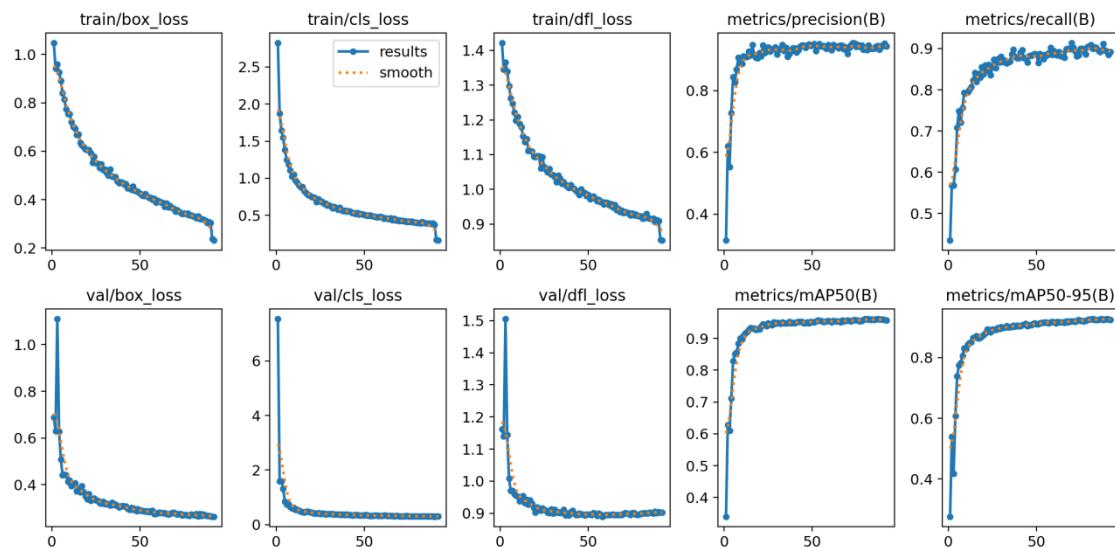
# Entrenamos el modelo
results = model.train(
    data=yaml_path, # Le pasamos la ruta donde esta nuestro yaml (en la notebook con los datos referenciando al dataset)
    epochs=100,
    imgsz=640,
    patience=10, # earlystopping en 10 epochas
    plots=True,
    project='detector_cartas', # Nombre del proyecto
    name='modelo_02_entrenamiento_01', # Nombre del modelo
    device=[0, 1], # Para entrenar con las 2 T4 que nos da Kaggle
    batch=32,
    cache='True'
)

```

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
92/100	14.8G	0.2316	0.1608	0.8528	69	640: 100%  ██████████	
90/90 [03:20<00:00, 2.22s/it]							
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95: 100%	
	███████	7/7 [00:12<00:00, 1.86s/it]					
	all	205	1227	0.944	0.892	0.957	0.924

**EarlyStopping:** Training stopped early as no improvement observed in last 10 epochs. Best results observed at epoch 82, best model saved as best.pt.  
To update EarlyStopping(patience=10) pass a new patience value, i.e. 'patience=300' or use 'patience=0' to disable EarlyStopping.

92 epochs completed in 5.856 hours.  
Optimizer stripped from detector\_cartas/modelo\_02\_entrenamiento\_01/weights/last.pt, 136.8MB  
Optimizer stripped from detector\_cartas/modelo\_02\_entrenamiento\_01/weights/best.pt, 136.8MB



Observaciones:

- El early stopping detuvo el entrenamiento en la época 92 luego de 5.86 hs.
- Fue el modelo que más tarde en entrenar de todos, lo cual es entendible debido a la complejidad del mismo.
- Perdidas de la última época:

```
wandb:      train/box_loss 0.23158
wandb:      train/cls_loss 0.16079
wandb:      train/dfl_loss 0.85284
wandb:      val/box_loss 0.26375
wandb:      val/cls_loss 0.30607
wandb:      val/dfl_loss 0.90283
```

Observando las métricas de perdida a la última época (92) se puede ver claramente un poco de overfitting.

Sin embargo observando en la gráfica de arriba (y también en la página wandb) se puede ver que en la época 82 (best.pt) no había overfitting en ninguna perdida.

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size
82/100	14.8G	0.3247	0.3936	0.9167	158	640: 100%  ██████████
90/90 [03:25<00:00, 2.29s/it]						
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95: 100%
	███████	7/7 [00:11<00:00, 1.66s/it]				
	all	205	1227	0.947	0.9	0.961 0.927

Cabe destacar que es el modelo que pérdidas más bajas tiene de todos.

- Sobre el final del log de entrenamiento se corrió una validación sobre el mejor modelo best.pt:

```

Validating detector_cartas/modelo_02_entrenamiento_01/weights/best.pt...
Ultralytics YOLOv8.2.59 🚀 Python-3.10.13 torch-2.1.2 CUDA:0 (Tesla T4, 15095MiB)
                                         CUDA:1 (Tesla T4, 15095MiB)
Model summary (fused): 268 layers, 68,170,755 parameters, 0 gradients, 257.6 GFLOPs

```

Class	Images	Instances	Box(P)	R	mAP50	mAP50-95): 100%
	7/7 [00:10<00:00, 1.44s/it]					

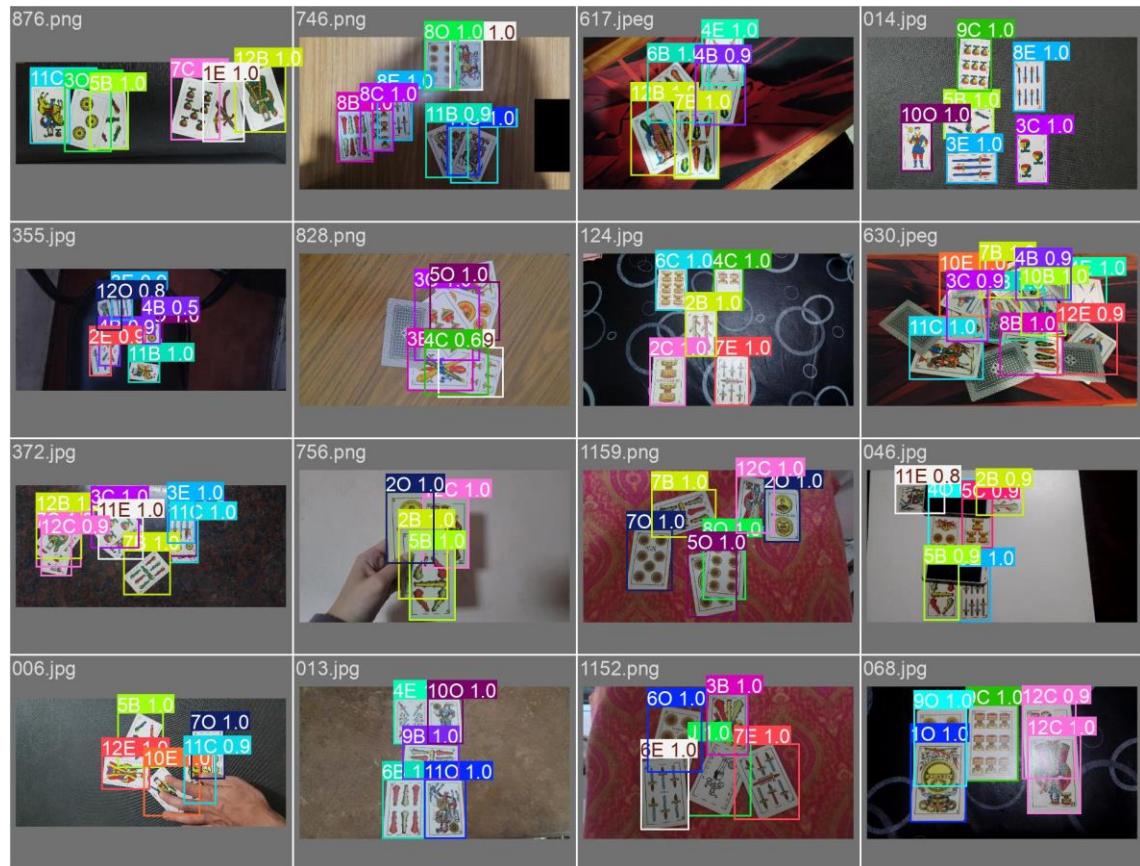
all	205	1227	0.947	0.9	0.961	0.928
10	35	35	1	0.889	0.966	0.941
1C	20	20	0.891	1	0.913	0.884
1E	26	26	0.961	0.956	0.987	0.954
1B	33	33	1	0.928	0.986	0.933
20	26	26	1	0.947	0.989	0.968
2C	22	22	0.95	0.955	0.965	0.921
2E	20	20	0.929	0.95	0.986	0.965
2B	25	25	0.957	0.891	0.975	0.919
30	24	24	0.912	1	0.992	0.935
3C	27	27	0.964	1	0.994	0.951
3E	26	26	0.961	0.959	0.97	0.962
3B	27	28	0.948	0.893	0.972	0.936
40	29	29	0.95	0.793	0.926	0.899
4C	19	19	0.869	0.699	0.863	0.824
4E	27	27	0.979	0.963	0.993	0.97
4B	28	28	0.988	0.964	0.983	0.945
50	30	30	0.929	0.933	0.984	0.946
5C	25	25	0.841	0.88	0.913	0.875
5E	41	41	0.973	0.976	0.993	0.975
5B	29	29	0.992	1	0.995	0.962
60	21	21	0.875	0.81	0.923	0.88
6C	24	24	0.951	0.808	0.926	0.89
6E	26	26	0.889	0.846	0.947	0.918
6B	25	25	0.908	0.96	0.941	0.919
70	30	30	0.839	0.867	0.93	0.898
7C	24	25	0.911	0.92	0.965	0.937
7E	20	20	0.944	0.841	0.93	0.91
7B	28	28	0.963	0.857	0.969	0.902
80	17	17	0.937	0.873	0.95	0.92
8C	21	21	0.943	0.784	0.95	0.93
8E	29	30	1	0.898	0.982	0.905
8B	28	28	0.971	0.893	0.965	0.933
90	26	26	0.979	0.885	0.968	0.937
9C	14	14	0.993	1	0.995	0.962
9E	14	14	0.932	0.981	0.972	0.947
9B	27	27	0.977	0.889	0.981	0.958
100	20	20	0.965	0.85	0.977	0.963
10C	17	17	0.957	0.824	0.963	0.928
10E	24	24	1	0.79	0.94	0.885
10B	26	27	0.934	0.815	0.942	0.908
110	25	25	0.937	0.84	0.951	0.939
11C	25	25	0.959	0.926	0.983	0.971
11E	25	25	0.922	0.96	0.97	0.933
11B	35	35	0.935	0.886	0.959	0.936
120	16	16	0.935	0.895	0.97	0.929
12C	25	25	0.954	0.92	0.926	0.893
12E	21	21	0.918	0.857	0.958	0.921
12B	30	30	0.966	0.961	0.965	0.928
J	19	21	0.993	0.905	0.925	0.905

Speed: 0.2ms preprocess, 19.7ms inference, 0.0ms loss, 2.9ms postprocess per image  
Results saved to `detector_cartas/modelo_02_entrenamiento_01`

Los valores para todo el conjunto (all) de mAP50=0.961, mAP50-95=0.928, superando nuevamente al modelo anterior.

Destaca en la evaluación clase por clase que ambos mAP en casi todas las clases superan el 0.9 lo cual es bastante bueno.

- Nuevamente las gráficas de perdida de validación mostraban lo que parecería ser una estabilización de la curva.
- Debido a las mejores métricas y con conocimiento de las consecuencias en cuanto a tiempo de inferencia y tamaño del modelo es el modelo con el que nos quedaremos.



### Evaluación de tiempos de inferencia de último modelo:

Se realizaron pruebas de tiempos de inferencia tanto en CPU, como en GPU en los formatos de entrenamiento de Yolo así como en formato TensorRT en GPU.

	CPU	GPU	GPU_tensor_rt
<b>Tiempo promedio inferencia</b>	1265.196238	41.732755	48.313904
<b>Tiempo promedio preprocessamiento</b>	2.948122	2.769573	2.978904
<b>Tiempo promedio postprocesamiento</b>	1.312172	1.292048	1.466162
<b>Tiempo promedio total procesamiento</b>	1430.117568	198.357654	198.348375
<b>FPS</b>	0.699243	5.041399	5.041634

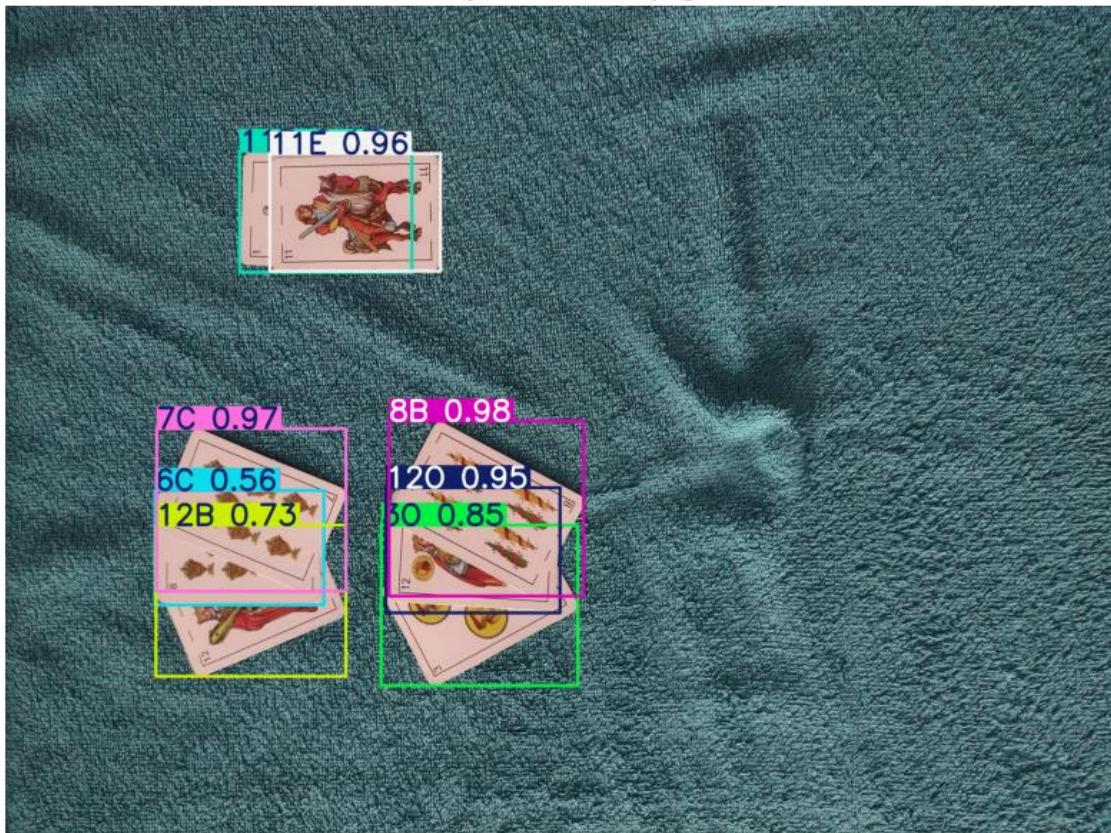
Los tiempos son bastante lentos, pero como para este ejercicio tenemos que hacer inferencia sobre imágenes únicamente, nos podemos permitir usar este modelo pesado que funciona mejor que los otros y tarda un poco más. Mirando la fila que mide a cuantos FPS iría un video haciendo inferencia con nuestro modelo es lógico pensar que habría que intentar hacer muchas más aumentaciones y entrenar con el modelo más pequeño posible para poder correr con este hardware a un framerate decente un video en tiempo real y además hacer más pruebas con la optimización de inferencia (que debido a problemas de tiempo no se pudo experimentar demasiado). Pero como mencione ya varias veces estoy al tanto de las consecuencias de elegir el modelo más pesado y tomo la decisión porque la tarea final lo permite.

Llama la atención como el modelo optimizado con TensorRT con GPU no mejora nada sus tiempos con respecto al modelo no optimizado con GPU.

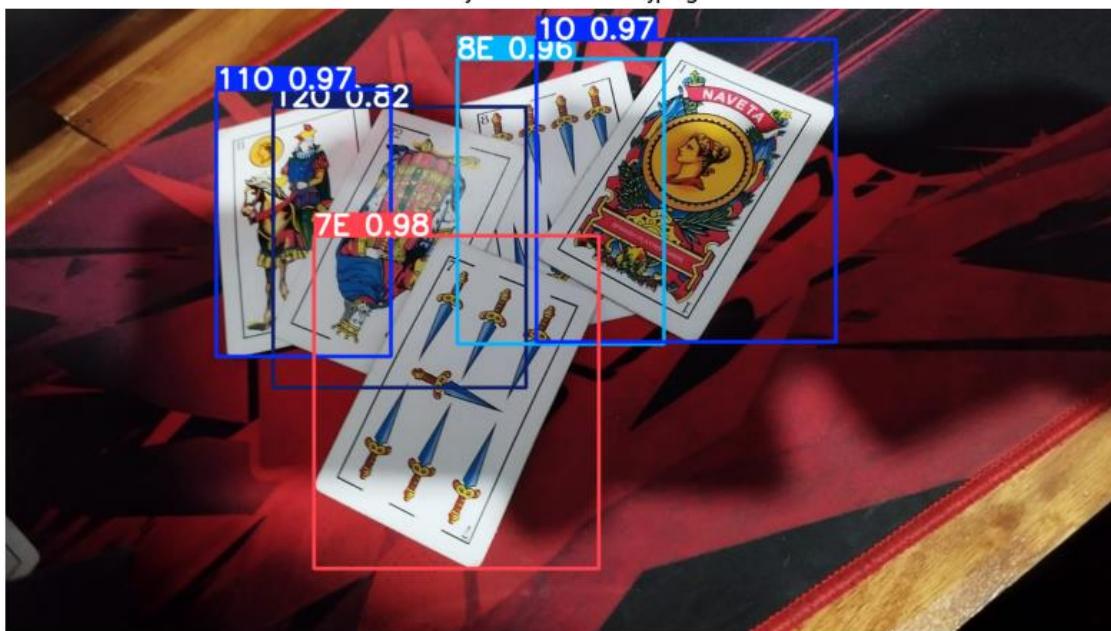
Por lo que finalmente se terminó escogiendo el modelo yolov8x sin optimización de inferencia.

Por último se realizaron algunas detecciones en el conjunto de test a modo de prueba previo a realizar el último ejercicio.

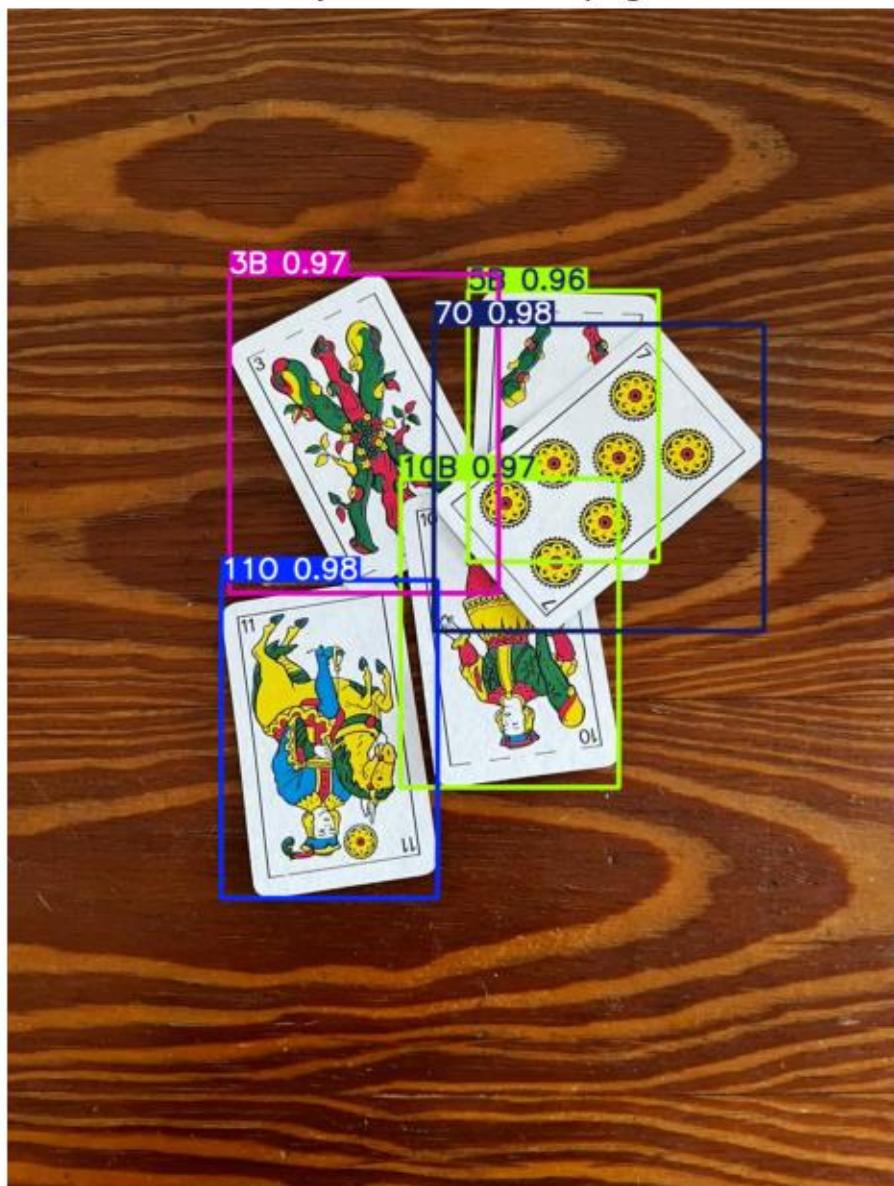
Conjunto test: 676.png



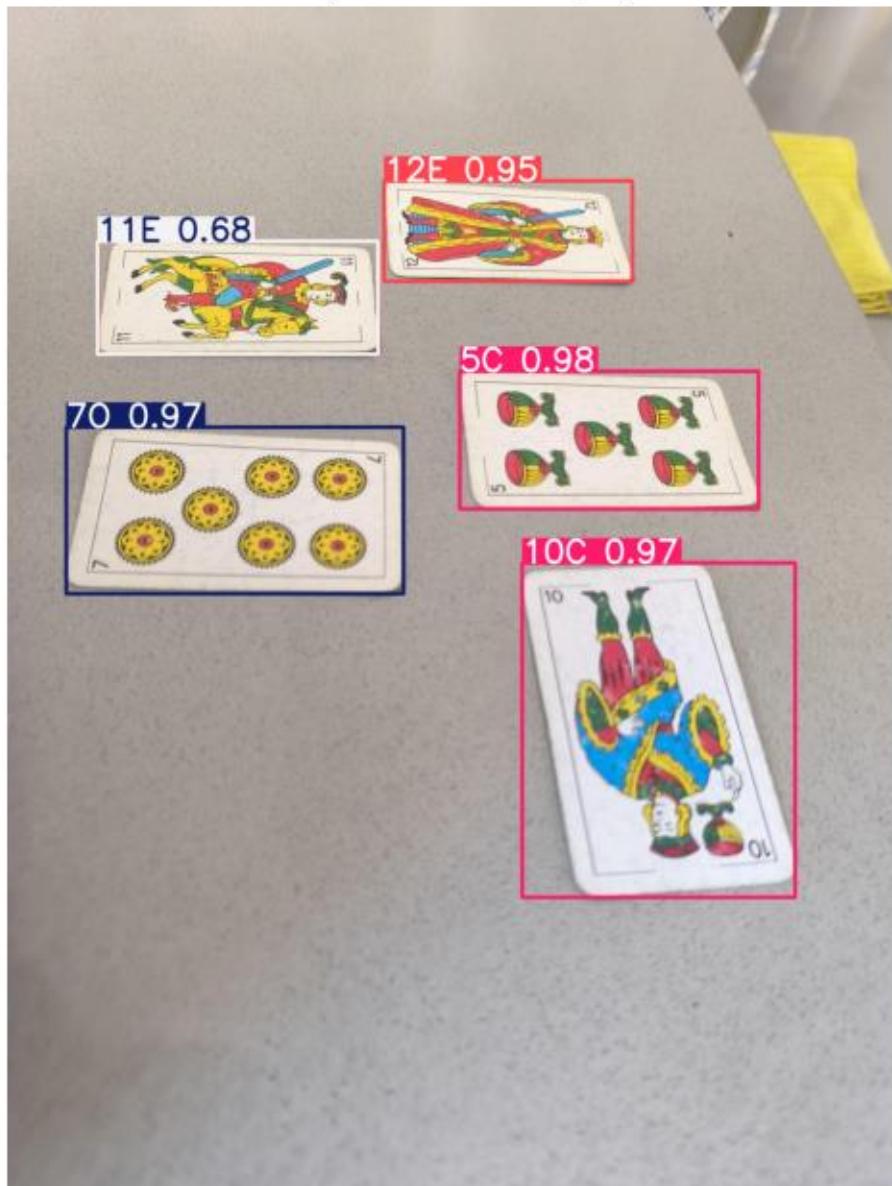
Conjunto test: 609.jpeg



Conjunto test: 1047.png



Conjunto test: 861.png



Conjunto test: 1114.png



En general el modelo parece funcionar bastante bien.

### **Ejercicio 04: Evaluación.**

Para el último ejercicio se volvió a trabajar en Google Colab en la notebook  
“TUIA\_Computer\_Vision-TP\_FINAL\_Ejercicio\_04.ipynb”.

Se subió a Google Drive tanto el modelo elegido como él .zip de evaluación descomprimido que la cátedra dio para probarlo.

Primero se montó el drive y se completaron todas las rutas que el cuaderno pedía.

```
[ ] drive.mount('/content/drive')
→ Mounted at /content/drive
```

```
[ ] # Nombre del alumno
student_name = "cesar_donnarumma"

# Ruta al archivo de pesos
model_path = "/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/modelos/03 - yolov8x_aumentacion_x2/weights/best.pt"

# Ruta al directorio que contiene las imágenes
imgs_dir = "/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/evaluacion/data/eval/images/val"

# Ruta al directorio de destino de las detecciones
base_dir = "/content/drive/MyDrive/TUIA_Computer_Vision/TP_02/evaluacion/data/out"
dets_dir = os.path.join(base_dir, student_name)

[ ] # Reestablecimiento del directorio de destino (eliminación)
if os.path.exists(dets_dir):
    shutil.rmtree(dets_dir)

os.makedirs(dets_dir)
```

Luego se cargó el modelo y junto con las rutas a cada una de las imágenes de evaluación.

Carga del modelo que vamos a utilizar

```
[ ] model = YOLO(model_path)
```

Imagenes de evaluacion

```
[ ] imagenes_evaluacion = [ img for img in os.listdir(imgs_dir) ]
imagenes_evaluacion = sorted(imagenes_evaluacion)
```

Se creó una función que generaba la estructura de json que el enunciado pedía para cada imagen evaluada.

Funcion de json generico

```
[ ] # Funcion que genera estructura de diccionario general para todas las cartas
def generar_carta_generica():

    carta_generica = {}
    carta_generica['total_cards'] = 0
    carta_generica['cards'] = {}
    carta_generica['cards']['E'] = []
    carta_generica['cards']['C'] = []
    carta_generica['cards']['B'] = []
    carta_generica['cards']['O'] = []
    carta_generica['points'] = 0
    carta_generica['figure'] = 'N/A'

    return carta_generica
```

Se creo un diccionario que contendrá el diccionario con la estructura de salida de cada una de las cartas.

```
# Diccionario que sera el json con los resultados
envido_json = {}
```

Luego iteramos una a una las imágenes de evaluación y vamos haciendo inferencia, y guardando los .txt en formato yolo con la confidencia al final en el path que el enunciado nos pide.

```

# Recorremos cada imagen
for imagen in imagenes_evaluacion:
    # Hacemos inferencia una a una (el resultado queda en results[0])
    results = model(os.path.join(imgs_dir, imagen), verbose=True, conf=0.5)
    # Guardamos los resultados de la detección en el .txt que pide el enunciado
    results[0].save_txt(txt_file=os.path.join(dets_dir, os.path.splitext(imagen)[0] + '.txt'), save_conf=True)

```

Generamos el diccionario con la estructura de json con la función de más arriba y lo guardamos en el diccionario que contendrá todas las salidas.

Obtenemos de los resultados las predicciones en formato índice, calculamos la cantidad de cartas detectadas y guardamos este número en ‘total\_cards’ del diccionario de la imagen.

```

# Agregamos al diccionario, futuro json, la sección correspondiente a la carta en cuestión
envido_json[imagen] = generar_carta_generica()
# Creamos una lista con índices correspondientes a cartas detectadas
cartas = list(results[0].boxes.cls.cpu().numpy())
# Guardamos en una variable la cantidad de cartas
total_cartas = len(cartas)
# Guardamos en el json la cantidad de cartas detectadas en la imagen en la sección correspondiente
envido_json[imagen]['total_cards'] = total_cartas

```

Si se detecta alguna carta, buscamos en el diccionario de clases que está guardado en los resultados el nombre que le corresponde a cada uno de los índices y dejamos afuera los comodines si llega a haber.

```

# Si se detectan cartas
if total_cartas > 0:
    # Buscamos el nombre de las cartas según su índice en el diccionario según su índice (filtrando el comodín)
    cartas = [results[0].names[carta] for carta in cartas if results[0].names[carta] != 'J']

```

Por cada detección, que es un string, guardamos en el diccionario de resultados tomando como key el ultimo carácter y como value el resto de caracteres (que serán números) casteados a int. Es decir estamos guardando con key=letra del palo, value=número de carta.

```

for carta in cartas:
    envido_json[imagen]['cards'][carta[-1]].append(int(carta[:-1]))

```

Si hay tres cartas y no hay ningún 8 o 9 entre las cartas podemos considerar que es una mano de truco.

```

# Si hay 3 cartas en la imagen y no hay ni 8 ni 9 podemos considerar que es una mano de truco
if (total_cartas == 3) and not (any(carta.startswith(("8", "9")) for carta in cartas)):

```

La idea ahora será descubrir si algún palo tiene 3 cartas, 2 cartas o no hay palo repetido.

Primero contamos la cantidad de cartas por palo y la guardamos en una lista ordenada.

```
# Calculamos la cantidad de cartas por palos
cantidad_espada = len(envido_json['imagen']['cards'][0])
cantidad_copa = len(envido_json['imagen']['cards'][1])
cantidad_basto = len(envido_json['imagen']['cards'][2])
cantidad_oro = len(envido_json['imagen']['cards'][3])
# Guardamos en una lista
cantidades = [cantidad_espada, cantidad_copa, cantidad_basto, cantidad_oro]
# Ordenamos la lista para quedarnos luego con el mayor
cantidades = sorted(cantidades)
```

Ahora en un diccionario a cada cantidad (clave) le asignamos el palo que le corresponde (valor).

```
# Creamos un diccionario que nos ayudara a encontrar cual es el palo que mas cartas tiene
cantidad_cartas = {}
# Metemos las cantidades como clave y el nombre del palo como valor
cantidad_cartas[cantidad_espada] = 'E'
cantidad_cartas[cantidad_copa] = 'C'
cantidad_cartas[cantidad_basto] = 'B'
cantidad_cartas[cantidad_oro] = 'O'
```

La idea es de las cantidades que antes guardamos en una lista quedarnos con la más grande, y meter esa cantidad en el diccionario de arriba que mapea al palo para saber que palo tuvo la mayor cantidad de cartas.

Creamos una lista con todas las cartas detectadas de ese palo (solo lo usaremos en el caso que sean 2 o 3 detecciones) que sean menores a 8 y las ordenamos de menor a mayor.

```
# De la lista con las cantidades por palo nos quedamos con la mas grande
maximo = cantidades[-1]
# Con el numero maximo como clave buscamos en el diccionario que nos indicara a que palo corresponde. Con dicho
# palo buscamos en el json todos los numeros del palo detectados. Nos quedamos con los menores a o iguales a 7
nuevas = [carta for carta in envido_json['imagen']['cards'][cantidad_cartas[maximo]] if carta <= 7]
# Ordenamos los numeros
nuevas = sorted(nuevas)
```

Si hay 3 cartas del mismo palo y 3 o 2 son menores o iguales a 7 los puntos son 20 sumado a la mayor y a la siguiente. Si hay solo 1 menor o igual a 7 los puntos son 20 + esa carta. Y si no hay ninguna menor o igual a 7 los puntos son 20.

```
nuevas = sorted(nuevas)
# Si hay 3 cartas del mismo palo
if maximo == 3:
    # Si hay 3 o 2 menores a 8
    if (len(nuevas) == 3) or (len(nuevas) == 2):
        puntos = nuevas[-1] + nuevas[-2] + 20 # Los puntos son 20 + la mayor + la siguiente
        envido_json['imagen']['figure'] = cantidad_cartas[maximo] # El palo es el palo que tiene la mayor cantidad de cartas
    elif len(nuevas) == 1: # Si hay 1 carta sola menor a 8
        puntos = nuevas[-1] + 20 # Los puntos son 20 + la unica carta
        envido_json['imagen']['figure'] = cantidad_cartas[maximo] # El palo es el palo que tiene la mayor cantidad de cartas
    else:
        puntos = 20 # Si no hay ninguna carta menor a 8 son 20 puntos fijos
```

Para 2 cartas del mismo palo se aplica la misma lógica de arriba.

```
# Si hay 2 cartas del mismo palo
elif maximo == 2:
    # Si hay 2 cartas menores a 8
    if len(nuevas) == 2:
        puntos = nuevas[-1] + nuevas[-2] + 20 # Los puntos son 20 + las dos cartas
        envido_json[imagen]['figure'] = cantidad_cartas[maximo]
    elif len(nuevas == 1): # Si hay una sola carta menor a 8
        puntos = nuevas[-1] + 20 # Los puntos son 20 mas la unica carta
        envido_json[imagen]['figure'] = cantidad_cartas[maximo]
    else:
        puntos = 20 # 2 cartas del mismo palo mayores a 9 son 20 puntos fijos
```

Si todas las cartas tienen distinto palo nuevamente nos quedamos de la vieja lista con las detecciones con las que sean menores o iguales a 7 y las ordenamos de menor a mayor. Si no hay ninguna menor a 8 los puntos son 0. Sino los puntos son el valor de la más grande.

```
# Si hay todas cartas de distinto palo
else:
    # Nos quedamos con las menores a 8
    nuevas = [ carta for carta in cartas if int(carta[:-1]) <= 7 ]
    # Ordenamos ascendentemente
    nuevas = sorted(nuevas)
    # Si no hay ninguna menor a 8
    if len(nuevas) == 0:
        puntos = 0 # Los puntos son 0 fijos
    else: # Sino
        puntos = int(nuevas[-1][:-1]) # Los puntos son los de la mas grande
        envido_json[imagen]['figure'] = nuevas[-1][-1] # El palo es el ultimo caracter de dicha carta
```

Guardamos los puntos en 'points' sea cual sea el caso.

```
# Asignamos los puntos en el json donde corresponde
envido_json[imagen]['points'] = puntos
```

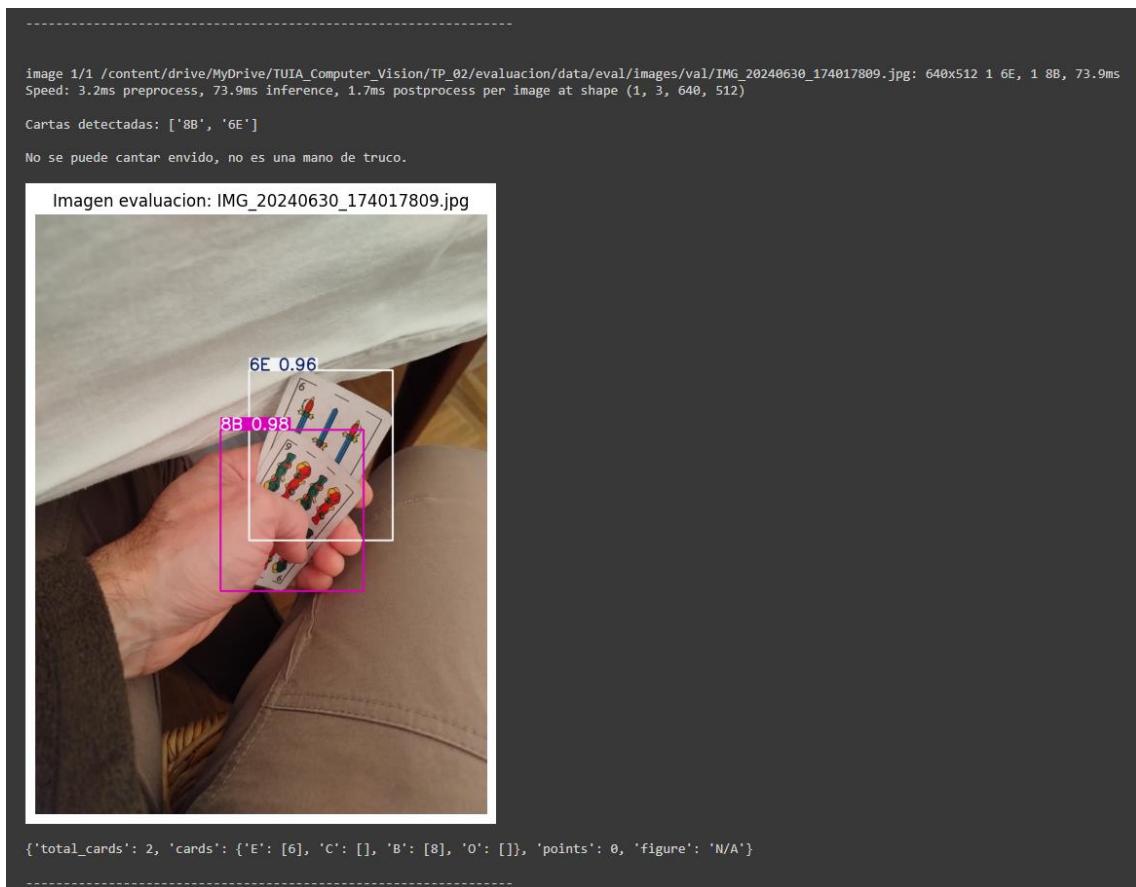
Si hay menos o más de 3 cartas u 8 o 9 entre ellas simplemente dejamos un mensaje de que no es una mano de truco.

```
# Si hay menos o mas de 3 cartas u hay 8 o 9 entre ellas:
else:
    print(f'No se puede cantar envido, no es una mano de truco.\n')
```

Por último se escribe el diccionario con todas las detecciones en un archivo json en la ubicación solicitada por el enunciado.

```
[ ] # Se abre el archivo en la ruta indicada (si no existe se crea) como escritura
with open(os.path.join(dets_dir, "envido.json"), "w") as jf:
    # Se escribe la informacion de envido_json en el archivo abierto con 4 espacio de indentacion
    json.dump(envido_json, jf, indent=4)
```

### Casos en los que las detecciones de evaluación funcionan mal.



Detecta el 9B como 8B.

```
image 1/1 /content/drive/MyDrive/TUIA Computer_Vision/TP_02/evaluacion/data/eval/images/val/IMG_20240630_174223161_HDR.jpg: 640x512 1 40, 1 12B, 73.9ms  
Speed: 3.1ms preprocess, 73.9ms inference, 1.5ms postprocess per image at shape (1, 3, 640, 512)
```

```
Cartas detectadas: ['12B', '40']
```

```
No se puede cantar envido, no es una mano de truco.
```

Imagen evaluacion: IMG\_20240630\_174223161\_HDR.jpg



```
{'total_cards': 2, 'cards': {'E': [], 'C': [], 'B': [12], 'O': [4]}, 'points': 0, 'figure': 'N/A'}
```

No detecta el 10E lo que lo lleva a decir que no es una mano de truco cuando debería haber cantado envido con 4 puntos y figure O.

```
image 1/1 /content/drive/MyDrive/TUIA_Computer_Vision/TP_02/evaluacion/data/eval/images/val/IMG_20240630_174316833.jpg: 640x512 1 20, 1 7C, 74.0ms
Speed: 4.4ms preprocess, 74.0ms inference, 1.9ms postprocess per image at shape (1, 3, 640, 512)
```

Cartas detectadas: ['20', '7C']

No se puede cantar envido, no es una mano de truco.

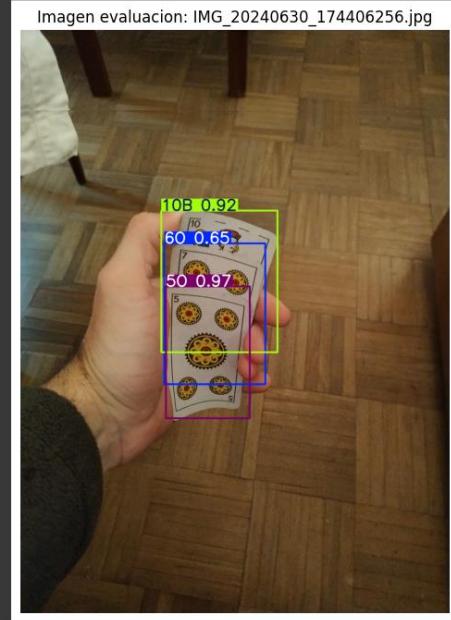


```
{'total_cards': 2, 'cards': {'E': [], 'C': [7], 'B': [], 'O': [2]}, 'points': 0, 'figure': 'N/A'}
```

Detecta el 4C como 7C.

```
image 1/1 /content/drive/MyDrive/TUIA_Computer_Vision/TP_02/evaluacion/data/eval/images/val/IMG_20240630_174406256.jpg: 640x512 1 50, 1 60, 1 10B, 73.9ms
Speed: 3.2ms preprocess, 73.9ms inference, 2.2ms postprocess per image at shape (1, 3, 640, 512)
```

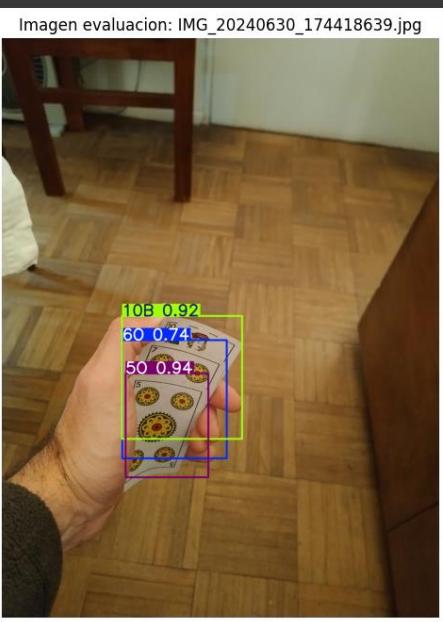
Cartas detectadas: ['50', '10B', '60']



```
{'total_cards': 3, 'cards': {'E': [], 'C': [], 'B': [10], 'O': [5, 6]}, 'points': 31, 'figure': '0'}
```

```
image 1/1 /content/drive/MyDrive/TUIA_Computer_Vision/TP_02/evaluacion/data/eval/images/val/IMG_20240630_174418639.jpg: 640x512 1 50, 1 60, 1 10B, 73.9ms
Speed: 3.2ms preprocess, 73.9ms inference, 1.8ms postprocess per image at shape (1, 3, 640, 512)
```

```
Cartas detectadas: ['50', '10B', '60']
```



```
{'total_cards': 3, 'cards': {'E': [], 'C': [], 'B': [10], 'O': [5, 6]}, 'points': 31, 'figure': '0'}
```

Mismo error en las dos imágenes anteriores, confunde 60 con 70 lo que lo lleva a cantar envido con un punto demás.

## Conclusiones:

Como conclusión más allá de que la experiencia a nivel personal haya sido satisfactoria puedo hacer la autocrítica (y me di cuenta tarde) de que podía haber hecho toda la parte de aumentación y preparación del dataset en local y evitarme largas esperas de subidas y descargas en Google Colab.

Además también podría haber comprimido quizás un poco las imágenes para que en su conjunto el dataset de 17GB con 2 aumentaciones de Kaggle pese menos y subirlo también tarde menos.

Por el lado de los modelos me habría gustado probar con más aumentaciones a ver si podía llegar a las mismas métricas con modelos más pequeños.