



Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
T.U.I.A
Computer Vision

Trabajo Práctico

Modelo detección cartas españolas

2024

Problema

Descripción:

El objetivo de este trabajo es desarrollar un modelo de visión por computadora el cual cumpla la tarea de detección y clasificación de cartas españolas, con la finalidad de aplicarlo en el juego del Truco Argentino.

Dataset:

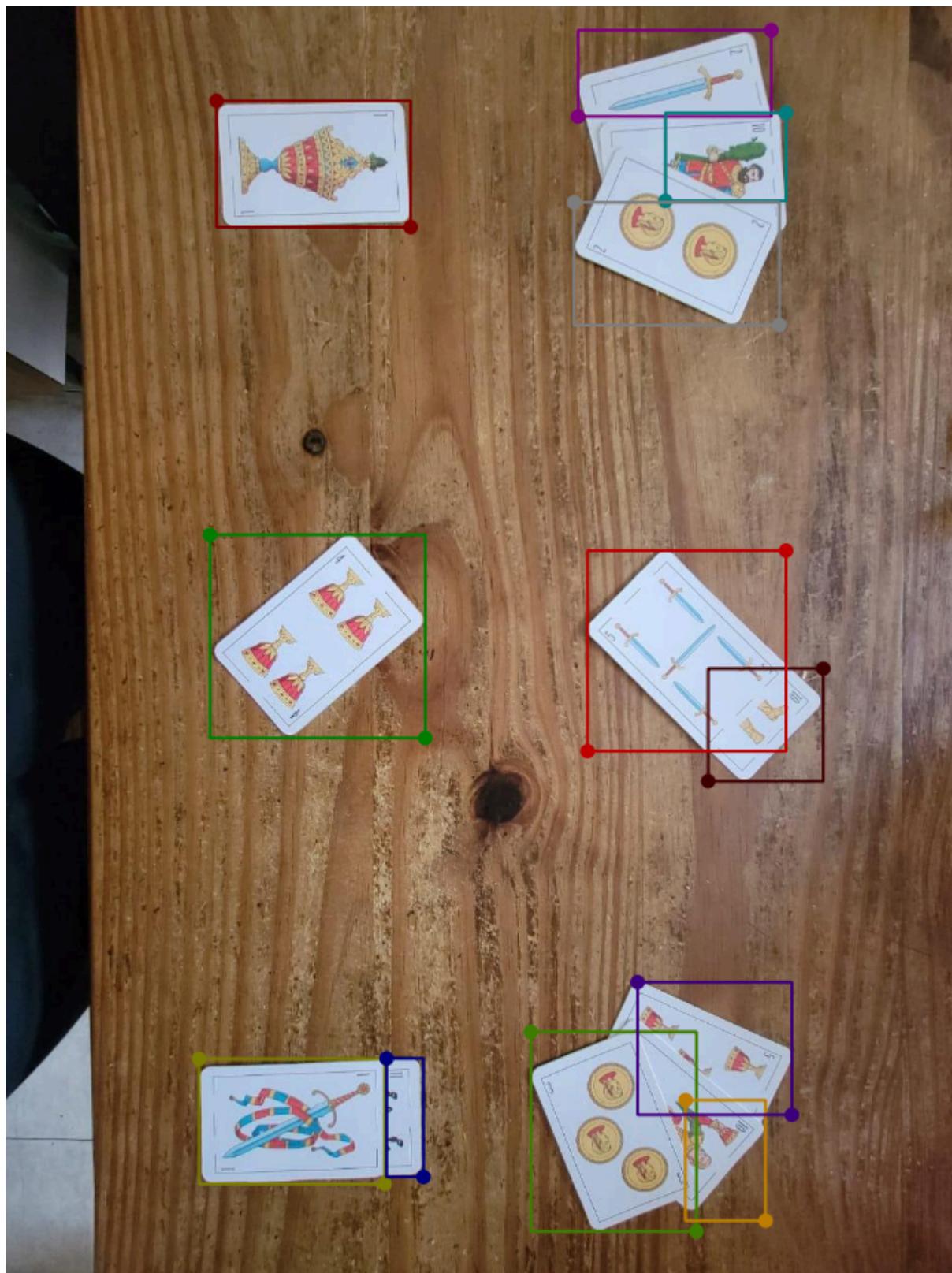
Para este proyecto, el formato de anotación propuesto es el de [YOLO](#), formato de vanguardia para los modelos de detección y clasificación de objetos.

En un principio, los profesores propusieron que los alumnos categoricen las fotos usando una única clase *carta*, lo que va en contra de la finalidad del formato YOLO y terminaría dificultando la tarea a los alumnos. Por esto, propuse a los profesores definir un estándar para anotar los datos, esto será explicado a continuación.

Para poder presentar y dar ejemplo al resto de los alumnos, se utilizó la herramienta [LabelMe](#). Esta es una herramienta muy potente la cual permite una fácil instalación y uso. Además, cuenta con detecciones automáticas para la segmentación mediante polígonos y máscaras, las cuales pueden llegar a ser útiles en el caso de querer aumentar datos.

Para ponernos de acuerdo, utilicé una imagen que según mi criterio engloba la mayoría de dificultades con la que nos podíamos encontrar.

Detección:



De esta manera, todos deberían seguir estas anotaciones.

Clasificación:

Para la clasificación, se utiliza en el label el número y la carta. De esta manera, el ancho de basto sería 1B, el 10 de copa 10C, etc. El joker o comodín se categoriza con la J. Para que todos podamos usar el mismo dataset, debemos formatear las anotaciones usando la misma lista de labels:

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

Oclusión:

En la foto anterior se pueden observar los distintos tipos de oclusión propuestos como desafío para nuestro modelo. El objetivo es que el modelo pueda no solo detectar cartas completas aisladas, sino que también pueda detectarlas, por ejemplo, viendo solo la sección superior, pues la cantidad de espacios en el superior e inferior de la carta determina el palo de esta. Se deben incluir en el dataset la mayor cantidad de tipos de oclusiones posible para que el modelo logre ser lo más robusto posible.

Recolección de datos:

Se compartieron entre los alumnos las imágenes y anotaciones que realizó cada uno. Cada uno luego tendrá que evaluar cuáles imágenes y/o anotaciones le servirán o no, y armar así su dataset. Cabe mencionar que todas las anotaciones, o su gran mayoría están en formato segment. Se debe transformarlas en formato bbox antes de entrenar

Resultado de la recolección de datos:

Luego de que todos los alumnos hayan enviado sus imágenes con anotaciones, se debió considerar cuáles elegir y cuáles no. Se dejaron múltiples imágenes fuera, tomando los siguientes criterios:

- Anotaciones defectuosas: Algunas anotaciones simplemente no tenían ningún tipo de sentido y debían ser descartadas para evitar problemas en el entrenamiento.
- Mal criterio para anotar: Ya sea por falso negativo o por anotaciones incorrectas.
- Dimensiones de imagen no adecuadas: Algunas imágenes estaban muy “estiradas”. Esto puede llegar a causar problemas al momento de hacer resizing, pues los modelos suelen tomar como entrada una imagen cuadrada.

Otra observación sobre los resultados obtenidos es que muy pocas personas consideraron el nivel de oclusión en el cual solo se ve la parte superior (o inferior) de la carta, por lo que se espera que las detecciones fallen en estos casos.



Como resultado, luego de detectar las malas anotaciones, terminamos con un dataset de alrededor de 700 imágenes. Con la intención de balancear el último tipo de oclusión mencionada, decidí incluir fotos propias para mitigar este problema. Se hablará de las aumentaciones más adelante.

Modelo:

YOLO no es solo un formato de anotación, sino un [modelo](#) muy capaz para la tarea de detección y clasificación de objetos. Consiste en un sistema de detección de objetos que realiza la tarea de detección y clasificación mediante una sola red neuronal. En un principio propuso un sistema de detección de clases y bounding boxes dividiendo a la imagen en un numero fijo de grillas, donde cada grilla tendrá también un numero fijo de bounding boxes, cada uno de los cuales tiene una probabilidad de que haya un objeto allí y además probabilidades condicionales de que haya una determinada clase dado que hay un objeto. De esta manera, cada grilla entrenará para que un único bounding box sea responsable de detectar un objeto, asignando un valor alto de probabilidad a este.

En particular, se utilizará la última versión de YOLO a la fecha: [YOLOV10](#). Este framework es muy similar al de YOLOV8, por lo que a pesar de ser nuevo no es difícil usarlo. Esta versión trae mejoras en la optimización de la arquitectura, lo que trae también mejoras en la latencia y tamaño del modelo, lo cual resulta aún más útil todavía en nuestro caso.

Métodos propuestos y problemáticas:

Nuestro problema consiste en la detección de cartas, estas consisten en un número y un palo, por ende, contamos con una tarea de detección y clasificación multilabel. YOLO no soporta en sí clasificación multilabel, a no ser que uno modifique la última capa y redefina su función de pérdida, lo que se decide no hacer en este trabajo debido a su complejidad, por lo que debemos considerar los siguientes:

- Como primer approach, podemos usar las clases que nos dan en bruto: 49 clases donde cada una corresponde a cada carta del mazo. Esto trae problemas respecto a la cantidad de datos presentes, los cuales son escasos, y de calidad cuestionable.
- Similar al anterior, se podrían modificar los labels para que a cada detección de carta se le asignen dos bboxes: una de número y otra de palo. Sin embargo, esto trae problemas respecto a cómo funciona YOLO: como cada grilla tiene una cantidad finita de bboxes encargados de detectar, cuando hay varias cartas juntas la detección falla, además que los resultados son malos.
- Otro approach es el de entrenar un modelo para detectar el número de la carta y otro para detectar el palo y luego ensamblar las detecciones, utilizando técnicas como NMS para mergear el bbox de palo y número en uno. De esta manera, cada modelo se encarga de su tarea, y podemos manejar la complejidad de estas de forma predecible.
- Otro approach es el de utilizar YOLO para que detecte únicamente la carta. Luego, por otro lado se entrenaría un clasificador para que tome el recorte de la carta y luego clasifique cada recorte. Esto trae notables ventajas y desventajas:

Ventajas:

- Primero, mitigamos el problema de la pérdida de información cuando YOLO hace resizing en la detección (hablaremos de esto más adelante). De esta manera, nos aseguramos de que siempre se pueda aprovechar una alta resolución para la clasificación de la carta.
- También, facilitamos la tarea de detección de YOLO, haciendo que esta sea mucho más precisa que si además tuviese que clasificar.

Desventajas:

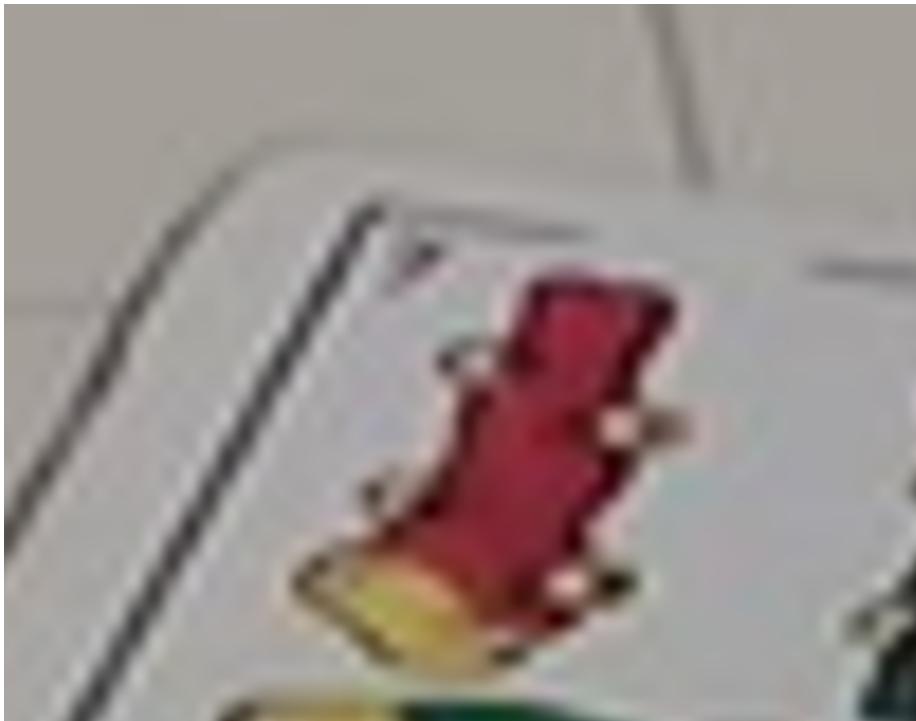
- A pesar de contar con detecciones más precisas, el hecho de contar con dos modelos hace que la velocidad de detección disminuya considerablemente.

Se decide trabajar con los últimos dos métodos. Sin embargo, antes de seguir, debemos tener en cuenta las dificultades con las cuales nos encontramos.

Principales problemas:

Una de las mayores complicaciones respecto al modelo de YOLO es el resizing: como para detectar números necesitamos en muchas ocasiones detalle de los dígitos o de las líneas para los palos, al cambiar la resolución perdemos información. Esto hace que falle la detección en los casos donde las cartas están lejos o que no generalize.

Ejemplo: recorte de imagen luego de hacer resizing a 640x640



(esto en teoría es un 7)

Además, esto también trae problemas a la hora de la aumentación de datos, pues es muy probable que al realizar alteraciones se pierda aún más información.

Es por esto que se opta por el approach del detector + clasificador, aunque también se intentará con el approach de dos YOLO, uno para cada tarea.

Antes de reportar el desempeño de los modelos, veamos la aumentación de datos.

Aumentación de datos:

A pesar de que casi cualquier tipo de aumentación genera pérdida de información, se detallan las técnicas utilizadas. Se utilizó la librería [augly](#) para realizar aumentaciones:

- Brillo, contraste y saturación.
- [Filtros de PIL](#): ver link.
- Ruido aleatorio.
- Emoji overlay: Controlando la opacidad, se espera causar pequeñas varianzas en las cartas, esperando que el modelo capte las características que la definen y no overfitee.

Sobre cada uno de estos se utilizó la función beta para manipular las probabilidades de la magnitud de cada aumentación. Los extremos de la función beta fueron ajustados según se tratase de aumentación sobre el dataset de detección o el de clasificación, pues la resolución y perdida de información cambian entre ambos datasets.



Imagen antes y después de ser aumentada.

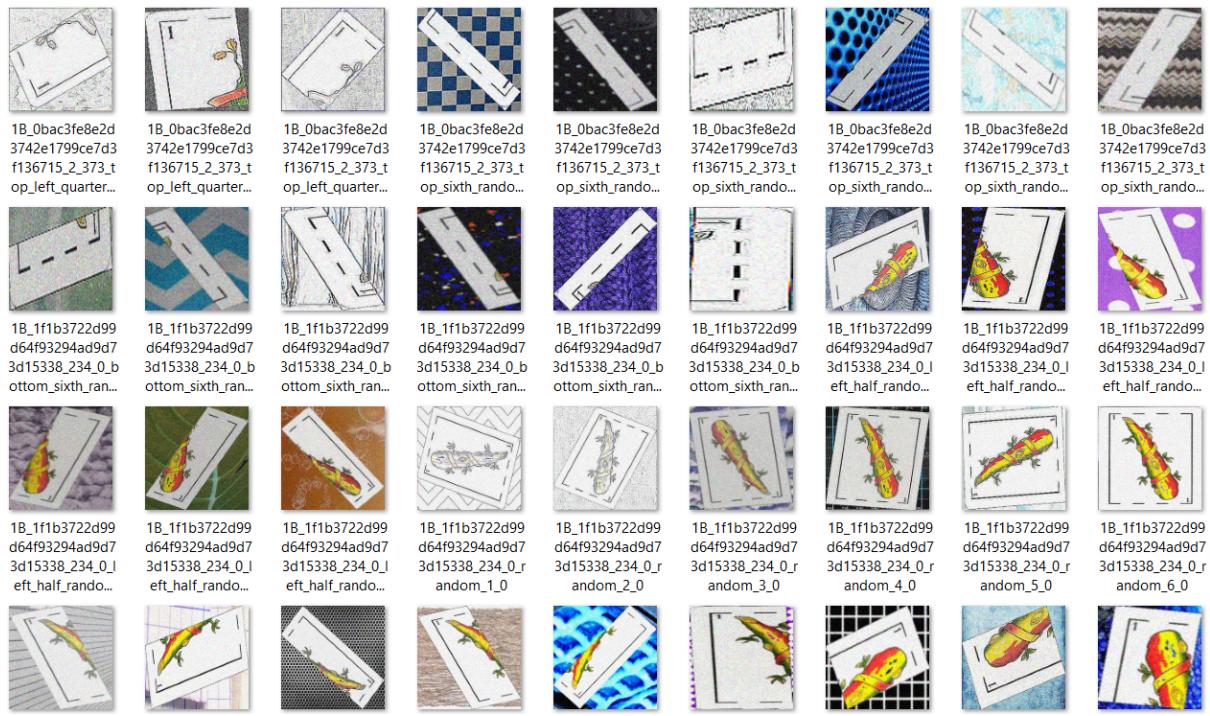
Otro tipo de aumentación que se consideró fue la de cambiar el background. Esto resulta muy llamativo, sin embargo no se profundizó sobre este método pues se necesitan conseguir las máscaras, cosa que resulta muy costoso.



Imagen antes y después de cambiar su background

Dataset para clasificador de recortes

Como necesitamos entrenar un clasificador para el approach detección con YOLO + clasificador, necesitamos crear un dataset acorde. Para esto, se consiguieron en un [blog](#) imágenes de alta calidad de distintas barajas de cartas españolas. Luego, se hicieron recortes de estas (asegurándose de tener la información necesaria para clasificar) y se rotaron, rellenando el background de la misma manera que antes. Además, se usó augly para realizar las mismas aumentaciones mencionadas anteriormente.



Diseñar el modelo para esta tarea fue algo que creí imposible, sin embargo el resultado final fue el siguiente:

Para el diseño de la arquitectura del clasificador, se utilizó transfer learning de las capas convolucionales del modelo MobileNetV3Large (Small no era lo suficientemente complejo). Para las capas densas, se definieron dos ramas: una para el número y otra para el palo. De esta manera, los palos y números comparten las mismas capas convolucionales pero ajustan sus propias capas densas

```
x = base_model(i, training=True)
x_n = GlobalMaxPooling2D()(x)
x_n = Dropout(0.5)(x_n) # Add dropout to prevent overfitting
x_n = Dense(30, activation='relu', kernel_regularizer=l2(1e-4))(x_n) # Add
x_n = BatchNormalization()(x_n) # Add batch normalization
x_n = Dropout(0.5)(x_n) # Add another dropout layer
x_n = Dense(15, activation='relu', kernel_regularizer=l2(1e-4))(x_n) # Add
x_n = BatchNormalization()(x_n) # Add batch normalization
x_n = Dropout(0.4)(x_n) # Add another dropout layer
x_n = Dense(15, activation='relu', kernel_regularizer=l2(1e-4))(x_n) # Add
x_n = BatchNormalization()(x_n) # Add batch normalization

x_s = GlobalMaxPooling2D()(x)
x_s = Dropout(0.5)(x_s) # Add dropout to prevent overfitting
x_s = Dense(20, activation='relu', kernel_regularizer=l2(1e-4))(x_s) # Add
x_s = BatchNormalization()(x_s) # Add batch normalization
x_s = Dropout(0.5)(x_s) # Add another dropout layer
x_s = Dense(15, activation='relu', kernel_regularizer=l2(1e-4))(x_s) # Add
x_s = BatchNormalization()(x_s) # Add batch normalization
x_s = Dropout(0.5)(x_s) # Add another dropout layer
x_s = Dense(15, activation='relu', kernel_regularizer=l2(1e-4))(x_s) # Add
x_s = BatchNormalization()(x_s) # Add batch normalization

# Output for numbers
x_number = Dropout(0.4)(x_n) # Add another dropout layer
num_output = Dense(12, activation='softmax', name='num_output')(x_number)

# Output for suits (including Joker)
x_suit = Dropout(0.5)(x_s) # Add another dropout layer
suit_output = Dense(5, activation='softmax', name='suit_output')(x_suit)

model = tf.keras.Model(inputs=i, outputs=[num_output, suit_output])
return model
```

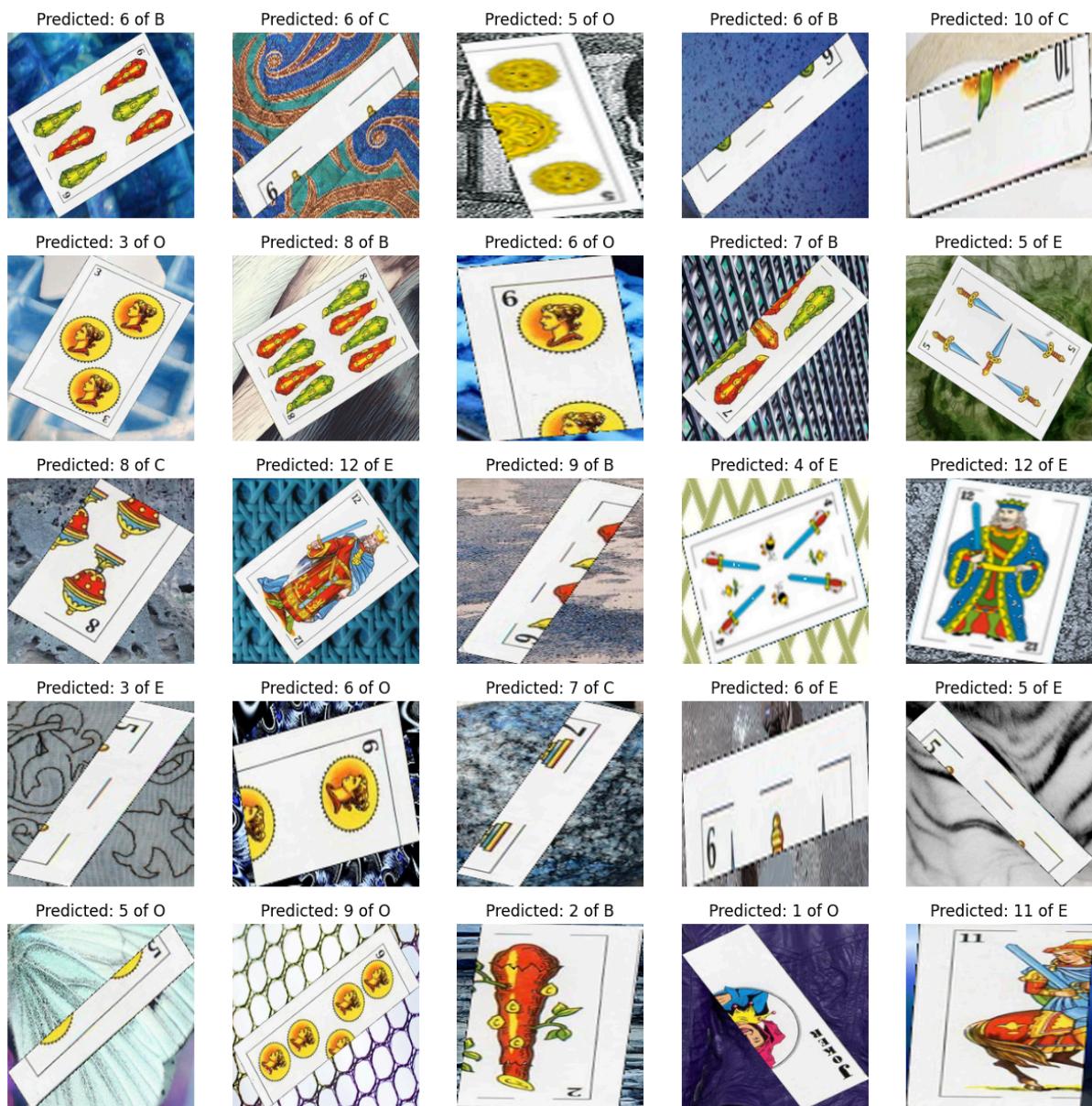
La función de pérdida debe definirse acorde a esta salida. Para esto, se utiliza la perdida categorical_crossentropy sobre cada capa de salida y se suman las pérdidas para la pérdida final. Tensorflow permite hacer esto en el compilador

```
model.compile(optimizer=optimizer,
              loss={'num_output': 'categorical_crossentropy', 'suit_output': 'categorical_crossentropy'},
              loss_weights={'num_output': 1.0, 'suit_output': 0.15},
              metrics={'num_output': ['accuracy', 'recall'], 'suit_output': ['accuracy', 'recall']})
```

Como el problema de clasificar palos es más simple que el de detectar números, sufrimos del problema de la pérdida dominante, lo que causa que primero se entrene la clasificación de palo y la de número quede estancada. Es por esto que se deben definir pesos sobre las perdidas. En este caso, el entrenamiento se vio parejo con un peso de 0.15 para los palos.

Los resultados fueron excepcionales:

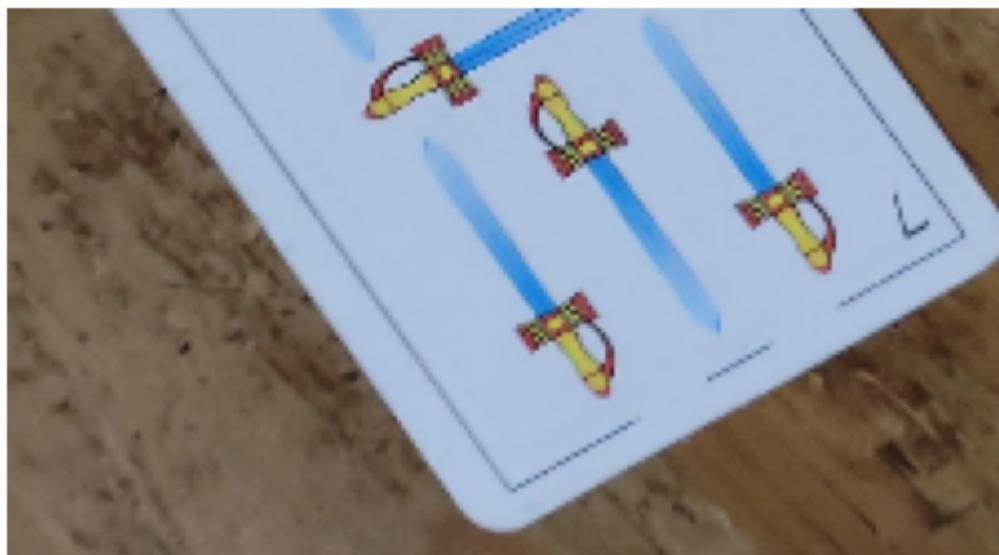
```
152/152 ————— 41s 267ms/step - loss: 0.2731 - num_output_accuracy: 0.9032 - num_output_recall: 0.8848 - suit_output_accuracy: 0.9752 - suit_output_recall: 0.9712 - val_loss: 0.6586 - val_num_output_accuracy: 0.9061 - val_num_output_recall: 0.9147 - val_suit_output_accuracy: 0.9445 - val_suit_output_recall: 0.9439
```



Sin embargo, se cuenta con un problema severo: el data drift.

Como estas imágenes fueron creadas artificialmente, cuando es llevado a un caso real, no predice como se espera:

Predicted: 3 of E

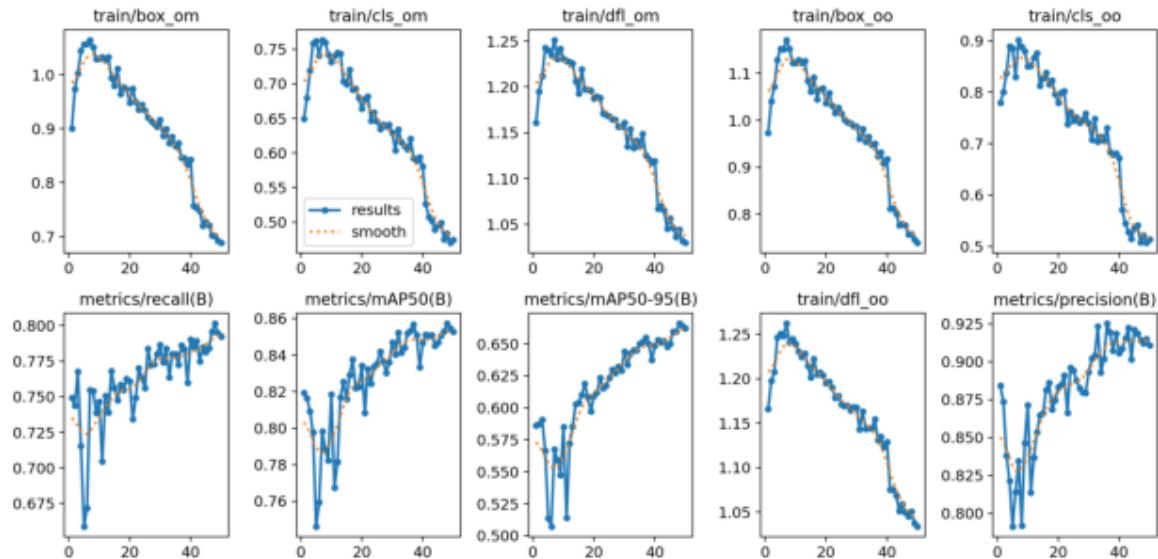


Es probable que esto se deba a los recortes que se hicieron. Solo se recortaron mitades, cuartos y extremos. En ningún momento se consideraron cortes diagonales como este.

Es posible que esto se pueda mitigar incluyendo recortes reales.

Los resultados del modelo detector YOLO son los siguientes:



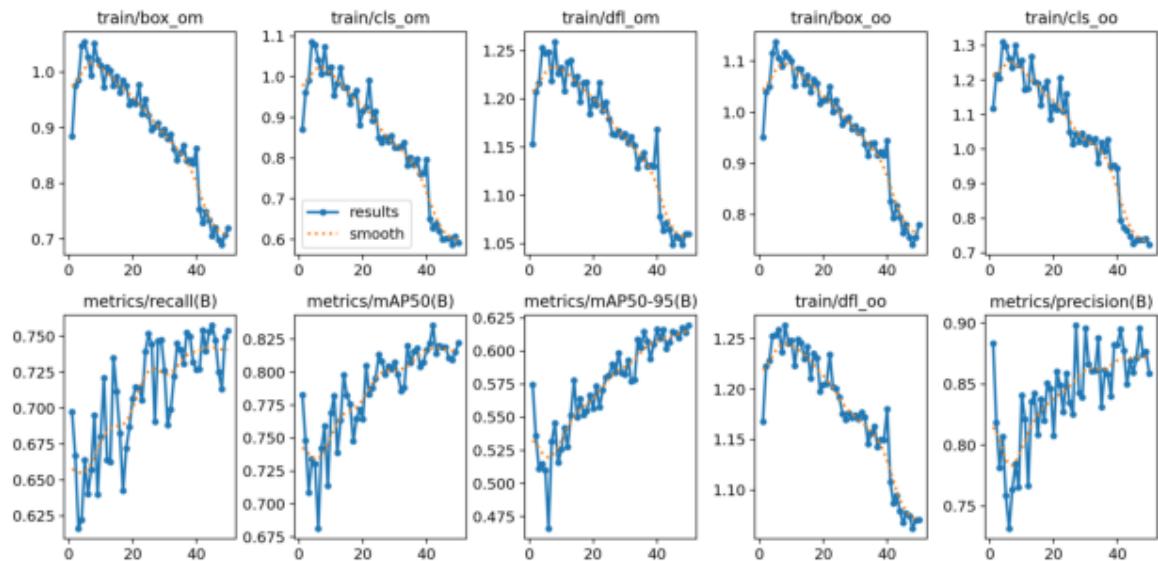


(este modelo se hizo con un dataset no validado, por lo que podría mejorar aún más)

Modelo YOLO de números y palos por separado

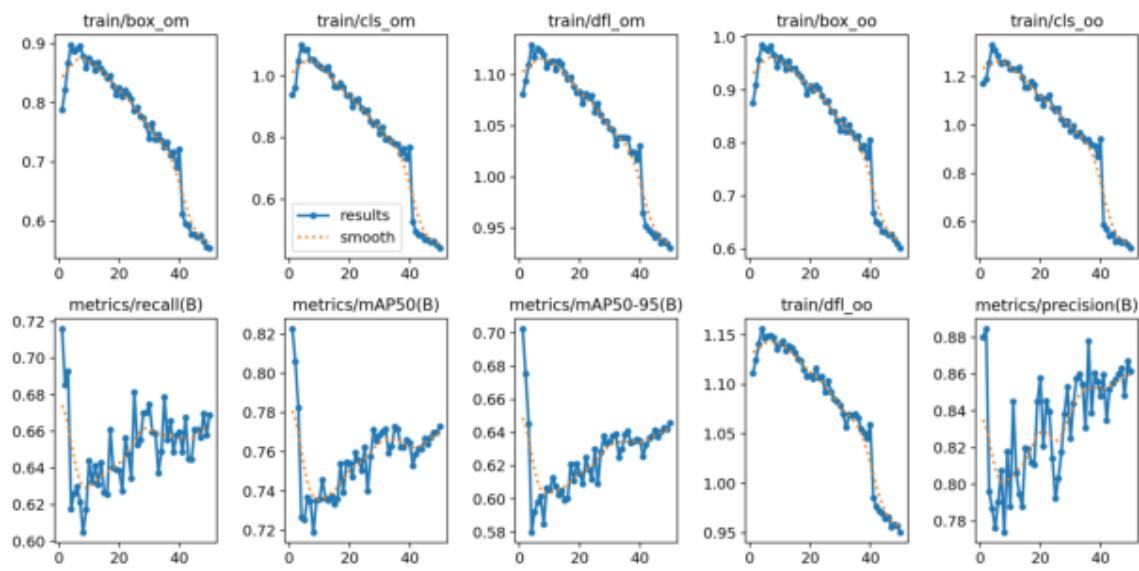
Aquí se incluyen los resultados de los entrenamientos de los modelos de YOLO. Como se mencionó anteriormente, los resultados no fueron los esperados, sin embargo, se muestran igual.

Palos





Números



Cuantización

Para el modelo YOLO se observaron técnicas de cuantización. En particular, se convirtió al formato onnx y tflite. Las instrucciones se encuentran en los notebooks del repositorio.

Conclusiones

A pesar de no haber podido dar con un modelo que cumpla la tarea solicitada, se han podido ver y reforzar múltiples conceptos desde la colección y validación de datos hasta el desarrollo y análisis de distintos tipos de modelos.

Queda ver la parte de corregir el dataset del clasificador, y técnicas de aprendizaje no supervisado para definir cuál es la mano de cada jugador.

Reentrega

Corrección del dataset y reentrenamiento del modelo

Modelo Detector

Para el modelo detector YOLO, se revisaron las anotaciones realizadas por los alumnos mediante un script y se descartaron aquellas anotaciones defectuosas. Además, se aplicaron las aumentaciones mencionadas anteriormente con intensidades leves ya que deja de ser crucial el nivel de detalle para el detector. También se aplicaron rotaciones de 90, 180 y 270 grados modificando también los labels.

Los resultados fueron los siguientes

Epoch	GPU_mem	box_om	cls_om	dfl_om	box_oo	cls_oo	dfl_oo	Instances	Size
34/50	5.68G	0.7424	0.394	1.035	0.8066	0.3739			
Class	Images	Instances		Box(P)	R	mAP50	m		
all	231	1461		0.964	0.957	0.983	0.637		

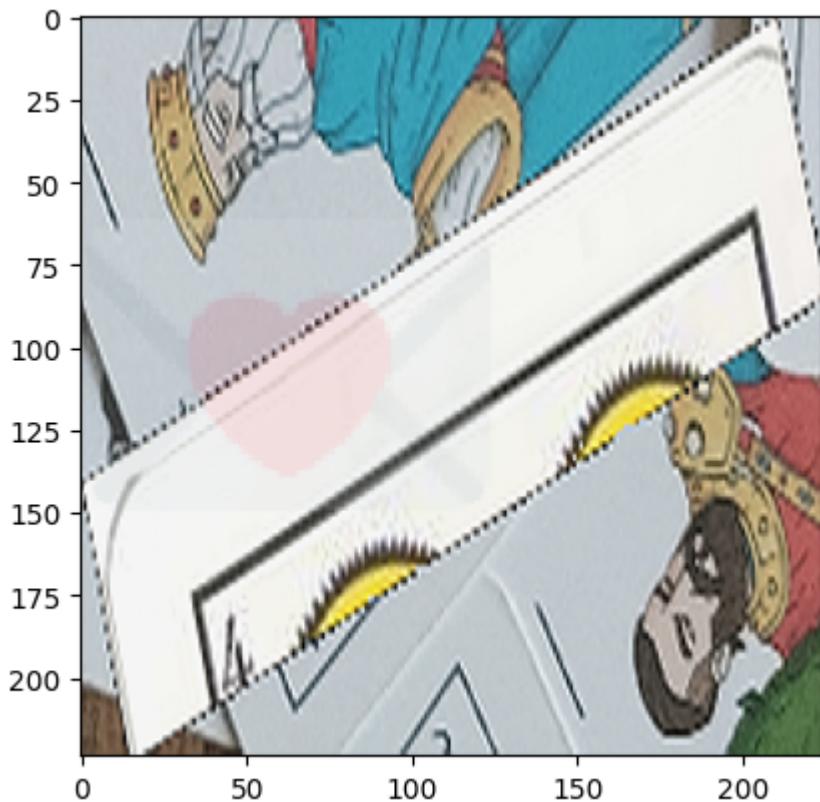
Notar que estas métricas son únicamente de la detección de la clase carta, por lo que el mAP sería únicamente un AP de la clase carta. Se revisarán estas métricas una vez se ensamblen los modelos

Modelo Clasificador

Como hemos visto en la entrega anterior, el dataset del modelo clasificador tenía problemas debido al data drift. Para enmendar esto, se incluyen recortes reales provenientes de las anotaciones originales mediante un script. Se deciden incluir algunas de las imágenes anteriores generadas sintéticamente para brindarle robustez al modelo. De esta manera, se logra brindarle al modelo una mayor robustez ante casos reales.

Se incluyeron aumentaciones adicionales:

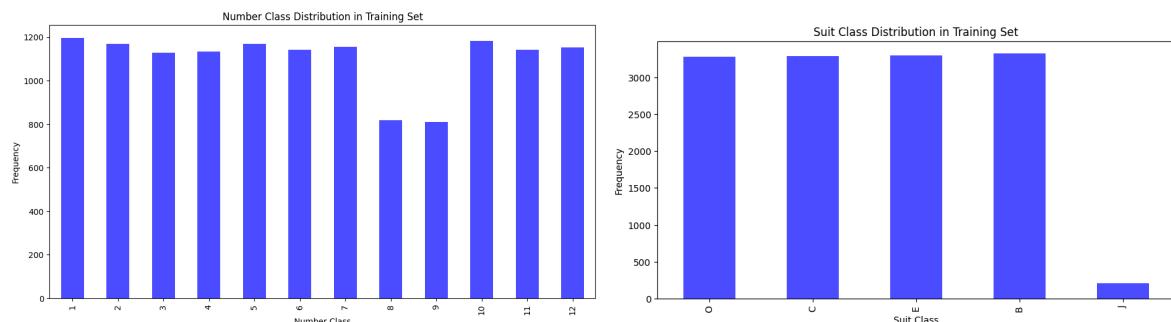
- Rotaciones.
- Se usaron backgrounds de cartas para las rotaciones de los recortes sintéticos.



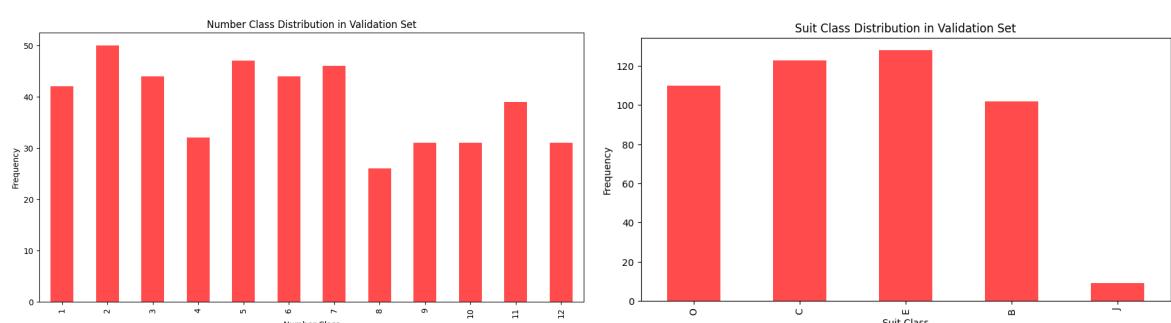
Carta aumentada con background de cartas

Balanceo de clases:

Training Set



Validation Set



Podemos ver un cierto desbalance en los 8 y 9, además del comodín. Esto puede deberse al hecho de que algunos alumnos entendieron que el dataset debía ser solo para el truco. Se podría considerar balancear las clases, sin embargo no se cree necesario.

Los resultados son satisfactorios:



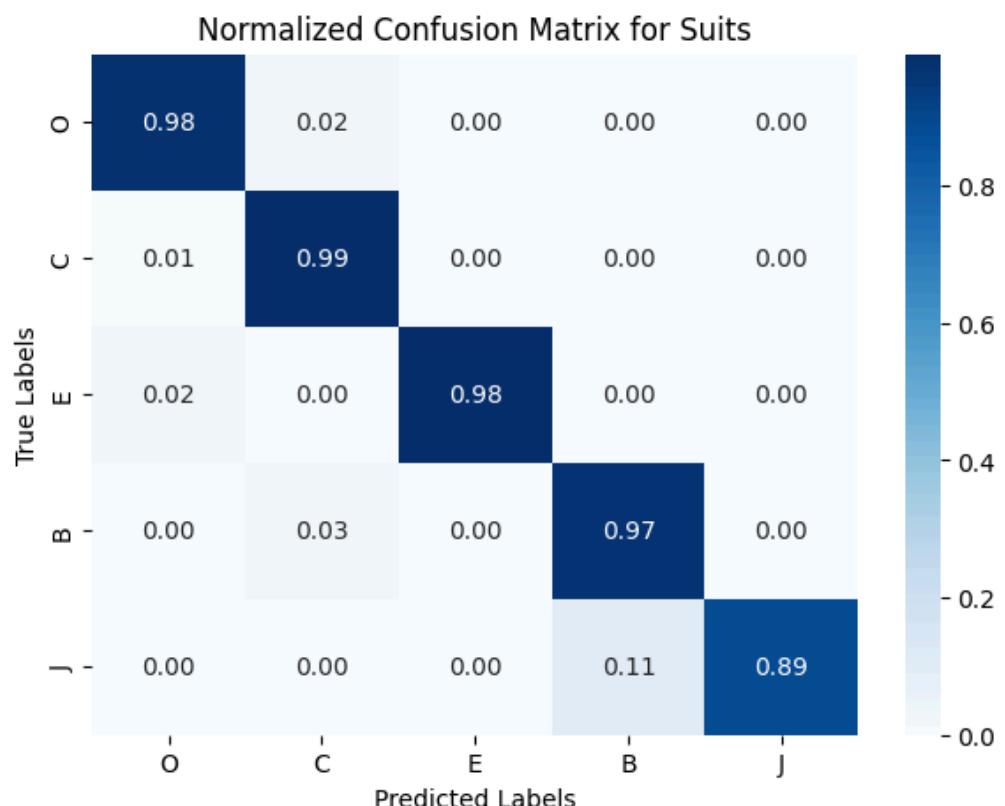
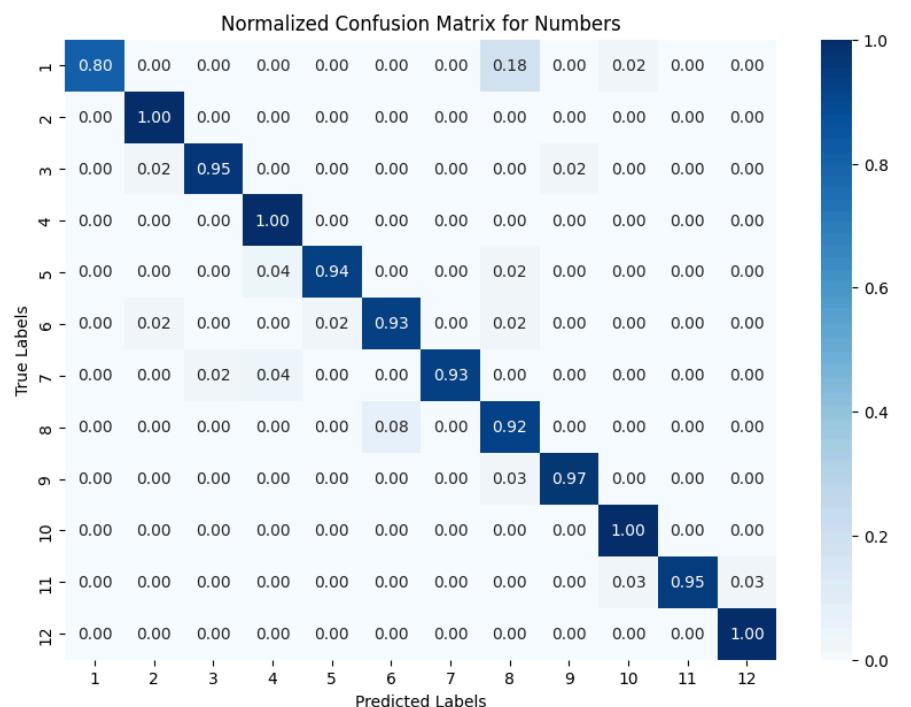
Inferencia sobre el conjunto de validación

Para las métricas del clasificador, se utilizó accuracy y recall tanto para los palos como para los números, estos son los resultados:

```
105/105 ━━━━━━━━ 25s 240ms/step - loss: 0.1802 - num_output_accuracy: 0.9299 - num_output_recall: 0.9332 - suit_output_accuracy: 0.9798 - suit_output_recall: 0.9733 - val_loss: 0.4427 - val_num_output_accuracy: 0.9449 - val_num_output_recall: 0.9611 - val_suit_output_accuracy: 0.9809 - val_suit_output_recall: 0.9809
```

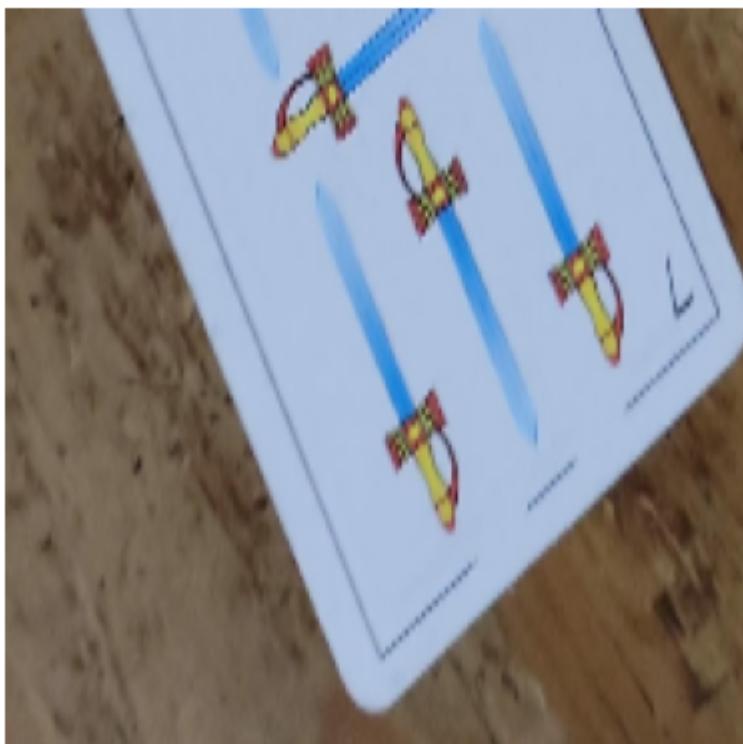
Notar que la diferencia de métricas entre train y val no se debe al underfitting, sino que las imágenes sintéticas se agregaron únicamente al conjunto de train, no al de validación.

Grafica de confusión



Haciendo inferencia sobre la carta que tenía problemas anteriormente:

Predicted: 7 of E



Análisis de métricas

Antes de comenzar a analizar las métricas de cada tarea, repasemos estas en la tarea de detección y clasificación de objetos.

mAP (mean average precision):

Repasemos los términos precision y recall para la detección de objetos (más adelante hablaremos del IoU):

Precision $TP / (TP + FP)$: Cuenta la cantidad de TP entre todas las predicciones realizadas.

Recall $TP / (TP + FN)$: Cuenta la cantidad de TP entre todos los targets reales.

En el caso de que haya una sola predicción realizada correctamente entre 10 boxes reales, la precision sería del 100% mientras que el recall sería del 10%. De la misma manera, en el caso de que haya un solo bbox real y 10 predicciones donde una le acierta, el recall sería del 100%, mientras que la precision del 10%

En nuestro caso, nos interesan ambas métricas, pues nos interesa tanto que no hayan predicciones donde no hayan cartas (precision alta) como también que no hayan cartas sin detectar (recall alto)

El mAP toma el area bajo la curva de la gráfica de precision-recall. El objetivo del trabajo no es explicar esta métrica. Sin embargo, hace falta mencionar que para definir TP (true positives) se usa un umbral de IoU específico. De esta manera el mAP-50 calcula el área tomando como TP a aquellos boxes que tienen un IoU mayor o igual a 0.5 con alguno real.

Teniendo esto en cuenta, y luego del análisis exploratorio del dataset, donde vimos que las anotaciones no han sido consistentes o de mala calidad, valoramos más métricas de mAP con un valor bajo de IoU, como 50%, pues es imposible ajustar boxes perfectos por lo antes mencionado.

Es esperable en nuestro caso que haya una brecha demasiado grande entre el mAP-50 y el mAP-95, como lo hemos visto en el modelo detector de cartas (98% y 63% respectivamente).

Ensamble de modelos

Antes de ensamblar modelos, se tiene que mencionar el formato de estos. Como nuestro approach deja en segundo plano la rapidez y se enfoca en la calidad de las predicciones, tiene sentido usar un formato de modelo que no suponga una pérdida de performance. Por esto, se usa el formato pb (protobuf) para el detector, y keras para el clasificador. De esta manera, el modelo detector pesa 32,5Mb y el clasificador 12,6Mb. Para realizar inferencia sobre estos modelos, se utiliza la API de Ultralytics de YOLOv10 para el detector y Tensorflow para el clasificador.

Este trabajo también contempla la posibilidad de utilizar el modelo en celulares o edge devices, por lo que se exportaron ambos modelos a formato tflite. La comunidad reporta que la cuantización en YOLO provoca una pérdida de información bastante alta, por lo que la performance de estos modelos no es la esperada. El modelo detector termina pesando 9Mb mientras que el clasificador 3,2Mb. Para realizar inferencia sobre estos modelos, se utiliza el intérprete de Tensorflow Lite. La inferencia del ensamble de estos modelos puede verse en el archivo tflite-inference.ipynb

El ensamble consiste en usar el detector YOLO para detectar únicamente cartas, y luego sobre cada bounding box, recortar el box y correr el modelo clasificador. De esta manera, conseguimos la misma detección que si estuviésemos utilizando un modelo de YOLO que también realize la tarea de clasificación.

Métricas

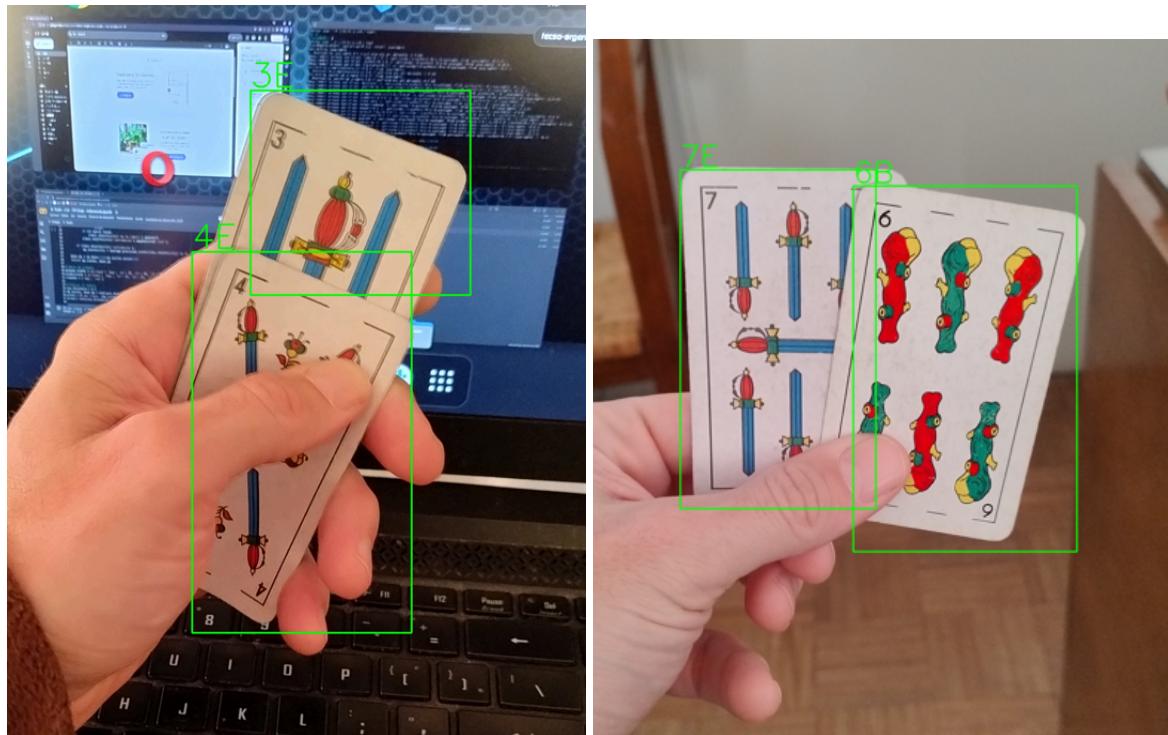
Una vez ensamblado el modelo, y utilizando las imágenes y anotaciones provistos por los profesores, calculamos las métricas solicitadas: mAP-50, precision, recall y F1Score. El cálculo de estas se detalla en el archivo card-model-metrics.ipynb. El cálculo de estas se realiza de la misma manera que se haría con un solo modelo YOLO para detección y clasificación, pues luego del ensamble se obtienen las mismas anotaciones.

IoU/Métrica	Precision	Recall	F1-score	mAP
0.5	0.8333	0.8698	0.8448	0.8333
0.75	0.7240	0.7604	0.7354	0.7240
0.95				0.1250

Como podemos observar, se obtuvieron buenas métricas para un valor de IoU de 0.5, sin embargo, existe una brecha muy grande entre esta y la de 0.95 por lo antes mencionado.

Envío

Los profesores solicitaron el cálculo de los puntos del envido. Esto puede encontrarse en el archivo card-model-inference.ipynb



Resultados del ensamble

Archivos:

Datasets:

- 224-classifier-w-real.zip: Imágenes para el modelo clasificador
- 640-rotated-1-class.zip: Imágenes para el modelo detector

Modelos:

- yolov10-card-96p.pt: Modelo detector
- best-classifier-model.keras: Modelo clasificador

Resultados:

- annotations.zip: Resultados sobre el conjunto de test dados por los profesores.
- annotations_envido.zip: Resultados de los puntos del envido.

Notebooks:

- card-model-inference.ipynb: Aquí se realiza la inferencia sobre el dataset de los profesores
- card-model-metrics.ipynb: Aquí se calculan las métricas del ensamble
- yolov10-train.ipynb: Entrenamiento del detector
- tp-final-classifier-train.ipynb: Entrenamiento del clasificador.