Informe de Práctica 1: Clasificación de eventos de interés en entornos portuarios

Correo	Nombre
i.moure@udc.es	Iván Moure Pérez
pablo.hernandez.martinez@udc.es	Pablo Hernández Martínez

Introducción

Esta práctica se enmarca en el contexto de plataformas inteligentes para la gestión de infraestructuras portuarias, en particular en la integración de un sistema de videovigilancia automatizado dentro del concepto *Smartports*. Uno de los retos operativos más relevantes es la identificación precisa y eficiente de eventos de interés en el entorno del muelle, como la presencia o atraco de embarcaciones. A partir del análisis de imágenes provenientes de cámaras CCTV, se busca implementar un sistema de clasificación que sea capaz de detectar estos eventos con alta fiabilidad.

Para ello, disponemos de un conjunto de datos que incluye 294 imágenes tomadas en diferentes condiciones de luz y perspectiva, etiquetadas por la presencia de barcos o el hecho de si están atracados o no. A partir de estas imágenes, se propone el entrenamiento de modelos de clasificación binaria utilizando redes neuronales profundas, explorando diferentes configuraciones de entrenamiento, uso de modelos preentrenados, y técnicas de aumento de datos. Este informe presenta el enfoque seguido, los modelos desarrollados y los resultados obtenidos.

Desarrollo de la Tarea 1: Implementación del Dataset

La primera parte de la práctica consistió en la creación de una clase `Dataset` personalizada, utilizando PyTorch, que permitiera una carga eficiente y flexible de los datos de entrada. Esta clase fue diseñada para funcionar tanto con las etiquetas de presencia de barco (`ship`) como con las de barco atracado (`docked`), permitiendo conmutar fácilmente entre uno y otro objetivo de clasificación.

El diseño de la clase permite cargar las imágenes desde directorios estructurados, leer las etiquetas desde ficheros CSV, y aplicar transformaciones tanto básicas como de aumento de datos. Esta flexibilidad fue fundamental para facilitar las distintas configuraciones de entrenamiento exploradas más adelante. La integración del parámetro `transform` en la clase permitió aplicar estrategias de *data augmentation* de forma controlada, sin tener que duplicar lógica de preprocesado.

Desde una perspectiva crítica, este diseño modular resultó ser una buena decisión, ya que permitió replicar el mismo pipeline para ambas tareas de clasificación, cambiando únicamente el archivo de etiquetas.

A continuación se muestra la distribución de clases de cada dataset.

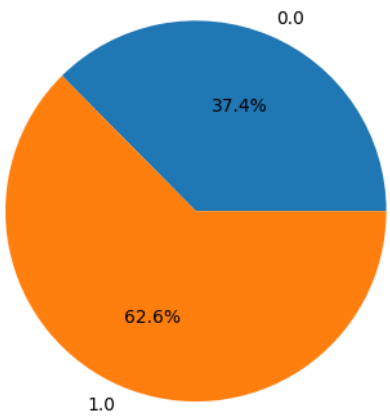

Figura 1: Distribución de clases en el dataset Ship (1: Hay barco, 0: No hay barco)

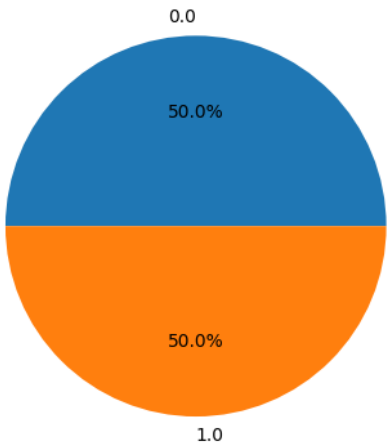

Figura 2: Distribución de clases en el dataset Docked (1: Atracado, 0: No atracado)

Desarrollo de la Tarea 2: Clasificación *Ship / No-ship*

La segunda tarea de la práctica tuvo como objetivo desarrollar un modelo capaz de identificar la presencia o ausencia de barcos en las imágenes. Se utilizó como base la arquitectura SqueezeNet1_0, la cual escogimos por su eficiencia computacional.

Empleamos cuatro configuraciones de entrenamiento diferentes:

1. Entrenamiento desde cero, sin técnicas de *data augmentation*.
2. Entrenamiento desde cero, utilizando *data augmentation*.
3. Refinamiento de un modelo preentrenado (*transfer learning*), sin *data augmentation*.
4. Refinamiento de un modelo preentrenado, con *data augmentation*.

El uso de *data augmentation* se diseñó cuidadosamente considerando las características del entorno portuario. Se aplicaron transformaciones como giros horizontales aleatorios, ajustes de brillo y contraste, y rotaciones leves, con el objetivo de simular las variaciones naturales de punto de vista, iluminación y condiciones atmosféricas que se encuentran en entornos reales.

Los resultados obtenidos muestran de forma clara que el uso de modelos preentrenados proporciona una ventaja significativa, especialmente cuando se combinan con técnicas de *data augmentation*. El modelo preentrenado con *data augmentation* logró no solo una mayor accuracy, sino también una mejor capacidad de generalización en el conjunto de validación, demostrando una menor pérdida y mayor estabilidad durante el entrenamiento.

Resultados de la clasificación *Ship / No-ship*

Configuración	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Desde cero - Sin Augmentation	86.44	84.62	94.29	89.19
Desde cero - Con Augmentation	86.44	82.93	97.14	89.47
Preentrenado - Sin Augmentation	91.53	89.47	97.14	93.15
Preentrenado - Con Augmentation	91.53	87.50	100	93.33

Lo que más nos llamó la atención es que el `recall` es siempre un poco más alto que el `precision`, indicando que los modelos aciertan todos o casi todos los ejemplos positivos pero etiquetan mal algunos negativos. Nuestra hipótesis es que esto se debe al ligero desbalanceo de clases, y que el modelo tiende a predecir positivo porque es la clase mayoritaria. A mayores también mostramos la curva precision-recall para todos los modelos, que nos puede ayudar a encontrar un umbral de clasificación distinto de 0.5 que equilibre más las métricas. A continuación se muestra una curva de ejemplo.

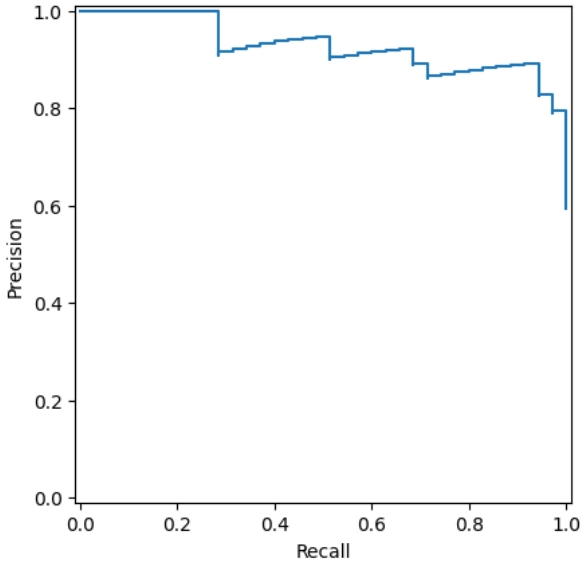

Figura 3: Curva Precision-Recall para el modelo entrenado desde cero sin Data Augmentation en el dataset Ship

En base a las necesidades de la aplicación, podíamos haber escogido otro umbral. En este caso consideramos que los falsos positivos son preferibles a los falsos negativos, así que decidimos dejar el umbral de 0.5.

Desarrollo de la Tarea 4 (opcional): Clasificación *Docked / Undocked*

También hemos realizado la tarea de clasificación para distinguir si el barco está atracado o no. Esta tarea presenta un nivel adicional de complejidad respecto a la detección de barcos, ya que las imágenes de una y otra clase pueden ser muy similares, cambiando solo por unos pocos píxeles.

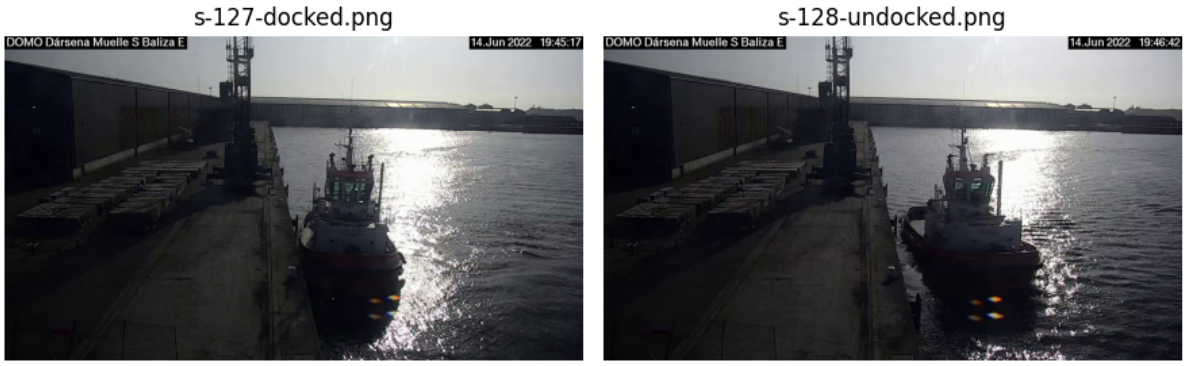

Figura 4: Ejemplo de 2 imágenes muy similares pero de clases opuestas

Reutilizamos el pipeline diseñado para la Tarea 2, adaptándolo al nuevo conjunto de etiquetas (`docked.csv`). De nuevo, exploramos las mismas cuatro configuraciones (entrenamiento desde cero y *transfer learning*, con y sin *data augmentation*).

Los resultados obtenidos siguieron una tendencia similar: los modelos preentrenados con *data augmentation* generalmente alcanzaron mejores métricas, aunque las diferencias entre las distintas configuraciones fueron ligeramente menores que en la Tarea 2. Lo cual puede deberse a la complejidad de la tarea o a la posible ambigüedad en algunas imágenes sobre el estado real de atraco.

Resultados de la clasificación *Docked / Undocked*

Configuración	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Desde cero - Sin Augmentation	72.97	69.57	84.21	76.19
Desde cero - Con Augmentation	72.97	68.00	89.47	77.27
Preentrenado - Sin Augmentation	75.68	70.83	89.47	79.07
Preentrenado - Con Augmentation	89.19	85.71	94.74	90.00

Los resultados obtenidos son visiblemente inferiores a los de la Tarea 2, lo cual no es sorprendente, porque ya sabíamos que este problema es más difícil que el anterior.

Experimentos y mejoras

Dataset

En primer lugar, leemos las imágenes de disco en el método `__init__` y no en el `__getitem__`. El primer método solo se ejecuta cuando se lee el dataset, y el segundo se ejecuta cada vez que accedemos a un índice: es decir, en cada epoch de entrenamiento estaríamos leyendo todo el dataset de disco. Esto sería lo adecuado si los batches de imágenes fuesen muy grandes, pero el dataset es suficientemente pequeño como para cargarlo entero en la memoria de la GPU. Este pequeño cambio nos ahorra sobre 5 segundos por epoch con un batch size de 64.

Al leer el dataset decidimos reescalar las imágenes a 144x256, preservando el ratio de aspecto original. No todas están en 16:9, pero nos decantamos por la opción mayoritaria. Más adelante, al ver que el entrenamiento era muy rápido (0.4 segundos por epoch), nos dimos cuenta de que nos podíamos permitir aumentar la resolución a 288x512. Esto mejoró el `accuracy` de la mayoría de modelos entre un 0% (preeentrenados en `ship`) y un 11% (desde cero sin DA en `ship`, preentrenado con DA en `docked`). Como decíamos anteriormente, la diferencia atracado/no atracado es cuestión de píxeles en algunas imágenes, y creemos que reducir demasiado al resolución de las imágenes estaba destruyendo esa información en algunos casos.

Entrenamiento y ajuste de hiperparámetros

Decidimos añadir parada temprana al bucle de entrenamiento para no tener que encontrar a mano el número óptimo de epochs. Al principio establecimos una paciencia de 3, pero vimos que los modelos en ambos datasets estaban entrenando demasiado poco (en concreto, los modelos desde cero en `docked` predecían siempre la misma clase). Subimos la paciencia a 5, luego a 7, y terminamos en 11. Al principio este valor nos parecía alto, pero había numerosas ocasiones donde el *loss* bajaba repentinamente después de una mala racha. 11 epochs es un margen tan amplio que solo se interrumpe el entrenamiento cuando el overfitting es bastante evidente. Decidimos devolver el modelo con los mejores pesos (respecto al loss de validación) porque normalmente el último modelo ya se ha sobreajustado demasiado. En la siguiente imagen podemos ver que si la paciencia fuese demasiado baja, el modelo habría parado alrededor de la epoch 60; sin embargo, con paciencia de 11 el modelo tarda más en "rendirse" y termina con un resultado mejor.

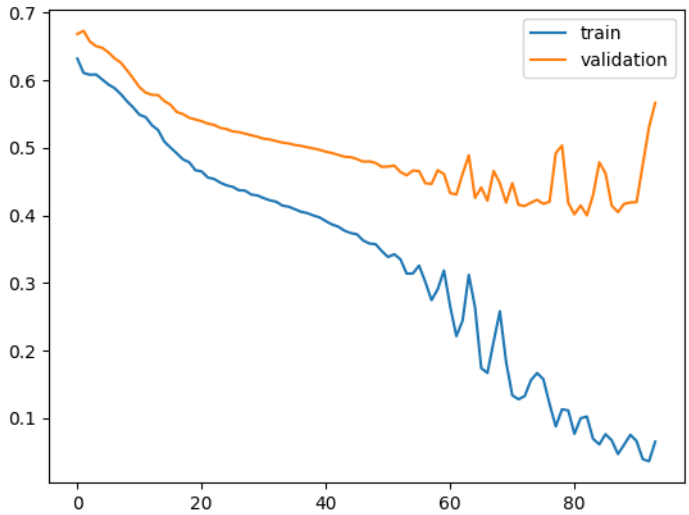

Figura 5: Desarrollo del loss a lo largo del entrenamiento del modelo desde cero con DA en Ship

En cuanto al *learning rate*, empezamos con 1e-3 para los modelos entrenados desde cero y 1e-4 para los preentrenados. El modelo desde cero sin DA en `ship` tiene probablemente el problema más simple, y por ende es el que converge más fácilmente y sigue teniendo 1e-3. Para el resto de modelos experimentamos con distintos valores, y finalmente nos decantamos por 1e-4. Un valor más alto no permitía converger y uno más bajo tardaba demasiado en hallar un mínimo.

Todos los modelos emplearon el mismo *batch size* de 64 por motivos similares a los del *learning rate*: un valor más alto no converge, y uno más bajo es demasiado lento. Si un *batch* de 64 no cupiese en memoria podríamos cambiar a 32 sin demasiado problema, pero no es el caso.

Posibles mejoras futuras

En el problema de `ship` observamos un leve desbalanceo de clases, con aproximadamente 2 fotos con barco por cada foto sin barco. Podríamos haber ponderado las instancias en entrenamiento o haber aplicado over/undersampling, pero es un desbalanceo bastante ligero y no parece que afecte demasiado al rendimiento de los modelos.

En nuestro sistema de parada temprana podríamos haber realizado algún tipo de comprobación para decidir si devolver el último modelo o el mejor. Si el mejor tiene 0.35 de loss y el último 2.1 la decisión es clara, pero si el mejor tiene 0.3521 y el último 0.3547 puede que el segundo haya "aprendido" más a pesar de tener un loss peor.

También pensamos en dividir las imágenes en ejemplos "claros" y "difíciles" mediante algún método de detección de anomalías. Con las imágenes fáciles realizaríamos un primer entrenamiento, y después realizaríamos un segundo entrenamiento con las imágenes más ambiguas. Creemos que esto podría ayudar a los modelos a generalizar mejor.

Otras opción sería emplear arquitecturas más potentes que SqueezeNet 1.0, como VGG16, AlexNet o una ResNET. Esto podría mejorar los resultados (creemos que por un margen muy pequeño) pero resultaría en un entrenamiento más lento y un modelo más grande. Nuestros modelos tienen 1.2m de parámetros y ocupan 4.83MB cada uno, mientras que una instancia de VGG16 tiene 138 millones de parámetros y ocupa 528MB. Consideramos que un modelo más pequeño sería más adecuado para ejecutar en local en la realidad, porque es poco probable que un puerto tenga la infraestructura de IA requerida para ejecutar modelos gigantes en tiempo real (o con una frecuencia aceptable).

Conclusiones

Esta práctica nos ha permitido diseñar e implementar un sistema de clasificación visual robusto para tareas críticas en el entorno portuario. La modularidad del pipeline y la correcta utilización de buenas prácticas en el preprocesado, partición de datos y entrenamiento han resultado claves para la obtención de buenos resultados.

Entre las principales conclusiones de esta práctica, se destaca que el uso de modelos preentrenados ha supuesto una mejora clara con respecto al entrenamiento desde cero. Estos modelos, al haber sido entrenados previamente sobre conjuntos de datos extensos, ofrecen una base sólida que permite obtener buenos resultados incluso en contextos con volúmenes de datos reducidos, como es el caso de este dataset portuario.

De la misma manera, las técnicas de aumento de datos han demostrado ser especialmente efectivas. Al introducir transformaciones que simulan variaciones típicas del entorno portuario (como cambios en la iluminación, ángulos de visión o rotaciones) hemos logrado aumentar la capacidad de generalización de los modelos, reduciendo el riesgo de sobreajuste y mejorando el rendimiento en los conjuntos de validación y prueba.

Por último, cabe mencionar que la tarea de detección de barcos atracados, a pesar de ser más compleja que la detección de presencia de barcos, puede abordarse de forma eficaz mediante una adaptación cuidadosa del mismo enfoque. Aunque los resultados en esta segunda tarea han sido más modestos, la metodología empleada ha demostrado ser válida y nos ofrece una base más que prometedora sobre la que seguir mejorando.

Debido a la dependencia de los datos para el entrenamiento en esta práctica, supondría una gran mejora en los resultados contar con una mayor cantidad de imágenes, así como con mayor variedad de escenarios. Si los recursos nos lo permitieran también nos ayudaría experimentar con arquitecturas más avanzadas que pueden capturar relaciones espaciales más complejas.