

6.3.1 Módulos con Lightning

Como se mencionó en el tercer capítulo 3 la librería más importante que empleamos en este proyecto, aparte de PyTorch, es Lightning. Gracias a ella muchos detalles de bajo nivel del uso de redes de aprendizaje profundo se obvian. Esto supone una gran ventaja a la hora de trabajar con estas redes, ya que la comprensión del funcionamiento del sistema en sí es más sencilla. Esto no es solo por su legibilidad, sino porque el número de líneas de código necesarias para implementar un módulo Lightning se reduce notablemente.

Como se puede apreciar en el extracto de código 6.1, para poder entrenar un modelo mediante un módulo Lightning solo hace falta especificar cómo cargar el modelo, establecer los hiperparámetros y el optimizador, y finalmente describir los pasos de entrenamiento y validación.

Por sencillez a la hora de trabajar con los modelos, los pasos de entrenamiento y validación se han simplificado al máximo, registrando el error producido en el paso de entrenamiento y guardando las detecciones en el paso de validación para promediarlas posteriormente, aparte de registrar su error.

6.3.2 Función de pérdida

Debido a que la implementación del modelo de EfficientDet que usamos generaba su propia pérdida y el resto de modelos prescindían de este mecanismo, diseñamos una función de pérdida simple para poder evaluar todos los modelos igualmente. Esta función está basada en el Índice de Jaccard, centrándonos solo en las predicciones de clase y localización mediante cajas. Esta función es la suma dos cálculos, uno de pérdida de clasificación y otro de regresión.

Para el cálculo del error de regresión, tomamos las *bounding boxes* generadas por el modelo y aplicamos el Índice de Jaccard/IoU para hallar las mejores correspondencias con las *bounding boxes* anotadas y reducir ambos conjuntos. Estos dos conjuntos con las mismas dimensiones sirven de argumentos para la función que computa la pérdida de forma completa. Esta función tiene en cuenta el área de superposición, la normalización de la distancia al punto central y la relación de aspecto, implementando la fórmula propuesta en el artículo “Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression” [7].

Para el cálculo de la clasificación tomamos la reducción de las clases identificadas y computamos su error con la función de Entropía Cruzada [31] por simplicidad, ya que solo contemplamos una clase. El error final para una predicción es la suma de ambos errores de clasificación y regresión para cada detección.

```

1  class RetinaNetLightning(LightningModule):
2      def __init__(self,
3          num_classes=config.NUM_CLASSES,
4          lr=config.LR,
5          threshold=0.2, #Threshold de scores de las bounding boxes
6          iou_thr=config.IOU_THR): #Threshold de solapamiento de cajas
7          super().__init__()
8          self.lr = lr
9          self.model = retinanet_resnet50_fpn(
10              weights = RetinaNet_ResNet50_FPN_Weights.DEFAULT,
11              weights_backbone = ResNet50_Weights.IMAGENET1K_V1)
12          self.threshold = threshold
13          self.iou_thr = iou_thr
14          self.loss_fn = compute_loss
15          self.mean_ap = MeanAveragePrecision()
16          self.val_step_outputs = []
17          self.val_step_targets = []
18
19      def forward(self, images : torch.Tensor, targets=None):
20          outputs = self.model(images, targets)
21          if not self.model.training or targets is None:
22              outputs = threshold_fusion(outputs, images_sizes(images),
23                  iou_thr=self.iou_thr, skip_box_thr=self.threshold)
24          return outputs
25
26      def configure_optimizers(self):
27          return torch.optim.AdamW(self.model.parameters(), lr=self.lr)
28
29      def training_step(self, batch, batch_idx):
30          images, targets, ids = batch
31          loss = self.forward(images, targets)
32          self.log('train_class_loss', loss['classification'].detach())
33          self.log('train_box_loss', loss['bbox_regression'].detach())
34          return loss['classification'] + loss['bbox_regression']
35
36      @torch.no_grad()
37      def validation_step(self, batch, batch_idx):
38          images, targets, ids = batch
39          outputs = self.forward(images, targets)
40          loss = self.loss_fn(outputs, targets)
41          mean_ap = self.mean_ap(outputs, targets)
42          self.log('val_class_loss', loss['class'])
43          self.log('val_box_loss', loss['box'])
44          self.val_step_outputs.extend(outputs)
45          self.val_step_targets.extend(targets)
46          return loss['total']
47
48      @torch.no_grad()
49      def test_step(self, batch, batch_idx):
50          images, targets, ids = batch
51          outputs = self.forward(images, targets)
52          loss = self.loss_fn(outputs, targets)
53          self.log('test_class_loss', loss['class'])
54          self.log('test_box_loss', loss['box'])
55          return loss['total']
56
57      def on_validation_epoch_end(self):
58          val_all_outputs = self.val_step_outputs
59          val_all_targets = self.val_step_targets
60          mean_ap = self.mean_ap(val_all_outputs, val_all_targets)
61          for k in config.KEYS:
62              self.log("val_"+k, mean_ap[k], logger=True)
63          self.val_step_outputs.clear()
64          self.val_step_targets.clear()

```

Listing 6.1: Ejemplo de módulo implementado con Lightning.

6.3.3 Integración de los modelos

Una vez descrito el comportamiento básico de los modelos, el siguiente paso fue adaptar su uso a la librería (PyTorch) *Lightning*. Como se explicó en 3, esto nos evita las partes más tediosas y repetitivas del entrenamiento y evaluación. Esto nos permite simplemente definir el comportamiento del modelo en las etapas de entrenamiento y validación. Con esto, solo faltaba definir los hiperparámetros para el entrenamiento y usar el módulo propio de *Lightning* para entrenar cada modelo, registrando las pérdidas en las etapas de entrenamiento y validación.

6.3.4 Módulo de división del conjunto de datos

Para generar versiones del conjunto de datos con distintas distribuciones para entrenamiento, validación y pruebas se diseñó un módulo que se encargara de esta funcionalidad. Aparte de otras funciones auxiliares este módulo contiene dos funciones principales: una para separar las anotaciones según las divisiones especificadas(80/10/10, 70/15/15, ...) y otra para convertir los ficheros en formato xml en un *DataFrame*. Con la finalidad de poder probar los modelos entrenados con cada división específica de los conjuntos de datos la función de separación tiene en cuenta que las imágenes de prueba sean comunes a todas las divisiones.

6.4 Umbralización

En esta parte se abordan los mecanismos integrados en el sistema para llevar a cabo un proceso de umbralización.

Como base del proceso de umbralización contemplamos dos umbrales: uno en base a la confianza del modelo para una predicción y otro para considerar dos predicciones como posiblemente la misma.

Con esto, la función de fusión ponderada de cajas citada en el capítulo 3 nos permite integrar ambos umbrales. Así, la función recibe las predicciones generadas por el modelo y ambos valores y en base a estos une las cajas muy solapadas, promediando la confianza de las cajas fusionadas.

De esta forma, con el umbral de solapamiento reducimos el número de predicciones finales y con el umbral de confianza provocamos que la salida solo refleje las predicciones con mejor puntuación.

Este proceso de división de los resultados en base a su índice de confianza tiene una alta variabilidad en sus resultados para los casos de modelos entrenados durante un número reducido de épocas o mediante la técnica de aprendizaje por transferencia. Así, un filtrado de los resultados con un umbral único para todas las predicciones se vuelve trivial ya que la red es capaz de discernir con una alta fiabilidad los casos correctos.

A parte del enfoque del umbral único tuvimos en consideración integrar algún mecanismo para que los datos de predicción se filtraran de forma dinámica en base a la entrada. La primera opción barajada fue incluir como parte de la red un parámetro entrenable que se ajustase en cada época de entrenamiento, a modo de umbral que el propio modelo fuese cambiando en base a la experiencia. Esta opción fue descartada ya que los resultados no eran satisfactorios, al menos mejores que los producidos por un umbral único.

Otra opción fue la de intentar cribar los resultados en base a un promedio de los tamaños de las cajas de predicción, pero se desecharó ya que de esta forma se reducía la capacidad individual de cada red propuesta de detectar mejor objetos dependiendo de su tamaño.

La última de las opciones tenidas en cuenta se centraba en adaptar el umbral dependiendo de características de la imagen como el brillo o ciertos canales. La implementación de esta opción se tornó demasiado costosa para la poca fiabilidad de los resultados, por lo que decidimos no incluirla.

Capítulo 7

Pruebas

En este capítulo se exponen los resultados de los experimentos llevados a cabo durante el diseño del sistema, ofreciendo un análisis de los mismos.

7.1 Inferencias sin entrenamiento

Como paso previo, y a fin de comprobar si los modelos elegidos pueden ser útiles en nuestro contexto, se realizaron pruebas simples en las que cada uno de los modelos propuestos realizaba el proceso de inferencia sobre un conjunto de imágenes de prueba (figura 7.1). En esta prueba no se alteraron el conjunto de etiquetas por defecto que identifica cada modelo, por lo que se exponen para cada objeto identificado. Este comportamiento se altera en pruebas posteriores, ya que la clase “tomate” no existe en los conjuntos de datos con los que estos modelos se han entrenado.

Por razones de legibilidad se han filtrado predicciones con un grado bajo de confianza, ya que queremos comprobar cuáles son las detecciones que cada red identifica mejor para comprobar si se podrían incluir en el sistema.

Como era esperable, las redes preentrenadas no son capaces de identificar los tomates acertadamente, ni en aspecto de clasificación ni de regresión, por lo que el siguiente paso sería entrenarlos con nuestro conjunto de datos.

7.2 Reentrenamiento completo

En esta sección se exponen los experimentos llevados a cabo adoptando como método de entrenamiento un aprendizaje de la completa, aportando gráficas en las que se muestran los parámetros de cada modelo durante el entrenamiento y ejemplos de inferencias.

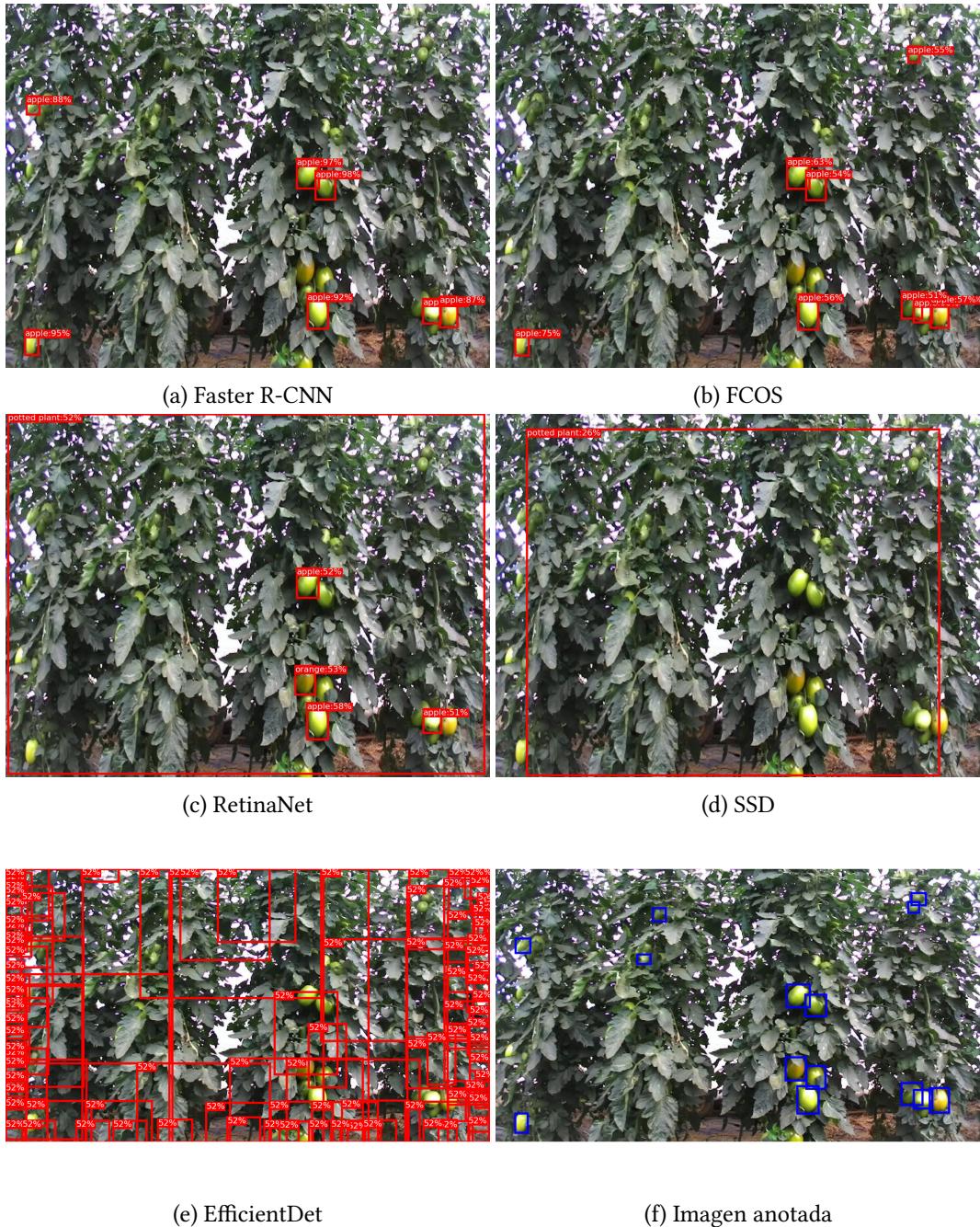


Figura 7.1: Inferencias de los modelos preentrenados

7.2.1 Error durante el entrenamiento

Se ha escogido un número de 50 épocas de entrenamiento para los cinco modelos elegidos como compromiso entre un entrenamiento intensivo y la posibilidad de sobreajuste.

Como se mencionó en el capítulo 4, el proceso de entrenamiento se llevó a cabo en la

máquina dedicada con un procesador superior, por lo que la mayor carga computacional ocurre en la GPU, reduciendo las operaciones en la CPU al cálculo del error de predicción y al computar las métricas de evaluación.

En general el proceso de entrenamiento sucede sin problemas, con un error decreciente al ubicar los tomates (figura 7.2), a excepción del modelo SSD. Como es comprensible la pérdida durante la validación es de un orden de magnitud mayor que la calculada durante el entrenamiento, pues esta se computa mediante una fórmula común y simple para todos los modelos. A diferencia de esta, la pérdida durante el entrenamiento es calculada de forma interna por cada red, incluyendo optimizaciones para ajustarla al funcionamiento específico de cada arquitectura.

7.2.2 Evaluación durante entrenamiento

Como se señaló en el capítulo 4, el criterio de evaluación seguido por el sistema es el *benchmark* de COCO, el cual calcula la precisión y el *recall* promediados. Además, expresa los valores de estas métricas para tres tamaños de los objetos, lo cual nos permite apreciar qué redes identifican mejor qué objetos.

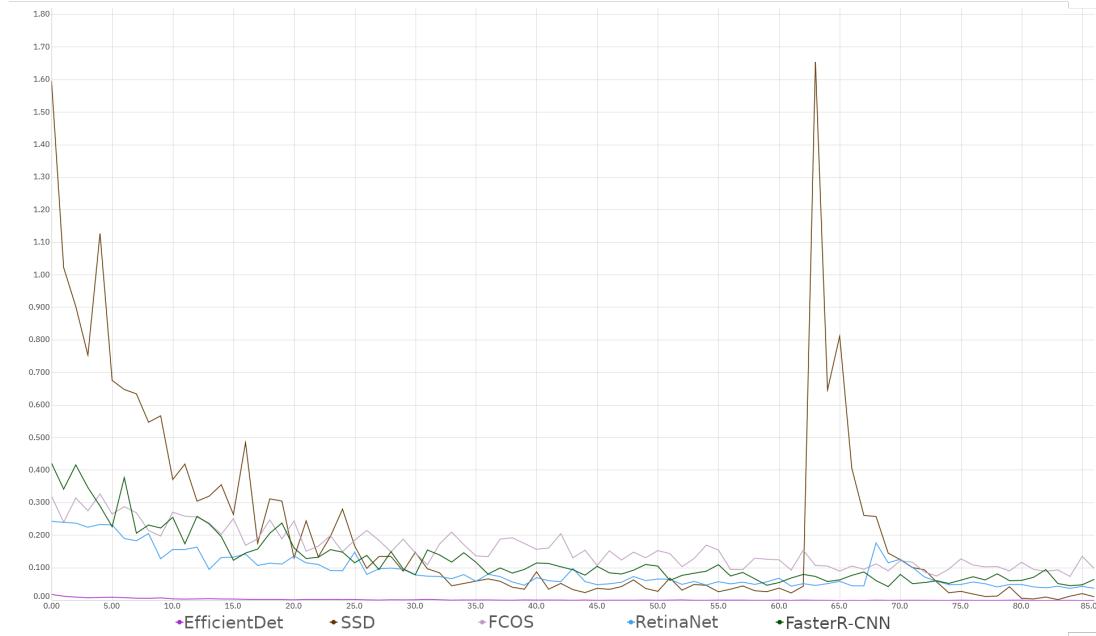
De forma simétrica a la pérdida, se observa que en general a medida que transcurren las épocas de entrenamiento las redes mejoran su precisión y *recall* (figura 7.3).

Evaluación de tiempo y consumo de recursos

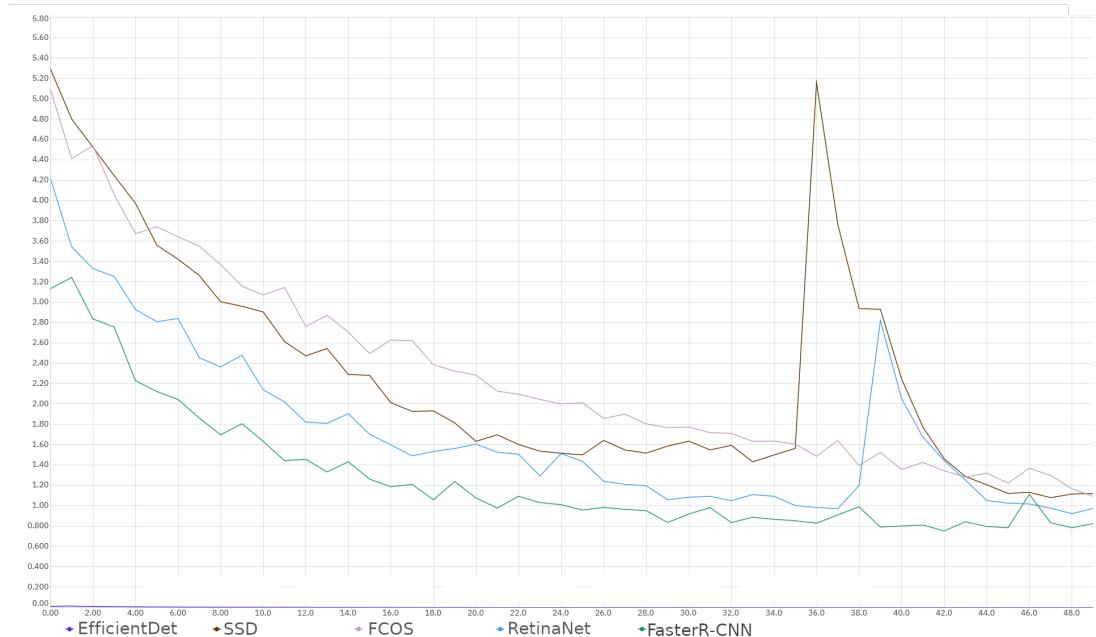
Gracias a la monitorización de las redes llevada a cabo mediante la herramienta Neptune.ai podemos observar la duración de cada entrenamiento, además de su consumo tanto de CPU como de GPU y la memoria total ocupada durante este proceso. Como se puede apreciar en la tabla 7.1 con los valores promediados, la arquitectura de EfficientDet hace honor a su nombre realizando un proceso de entrenamiento en menos tiempo, involucrando menos recursos. El segundo modelo más rápido y con menos gasto de procesador y memoria es SSD, lo cual se debe a que se trata de una arquitectura menos compleja. Los resultados de esta red, consecuentemente son peores, ya que normalmente se toma como columna vertebral de otras redes sobre la que añadir optimizaciones.

7.2.3 Entrenamiento con aprendizaje por transferencia

Como vía alternativa para tratar de optimizar tanto el proceso de entrenamiento del modelo como la capacidad de generalización, llevamos a cabo un entrenamiento de las redes mediante el aprendizaje por transferencia (capítulo 2). De esta forma, se pretendía reducir la carga computacional del entrenamiento, ya que se reducía considerablemente el número

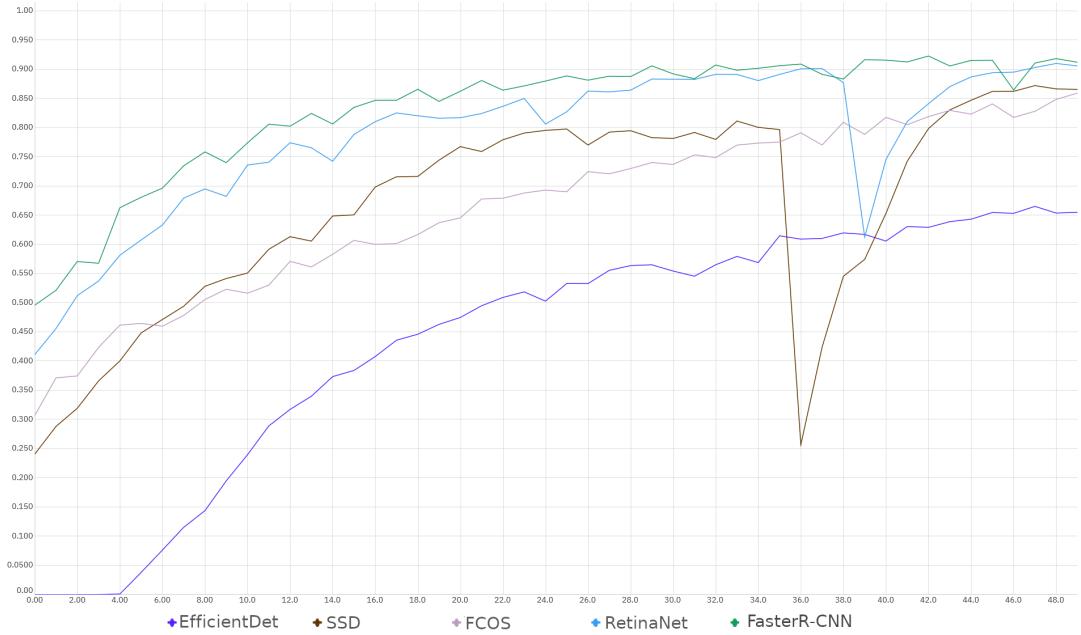


(a) Error de regresión durante el entrenamiento.

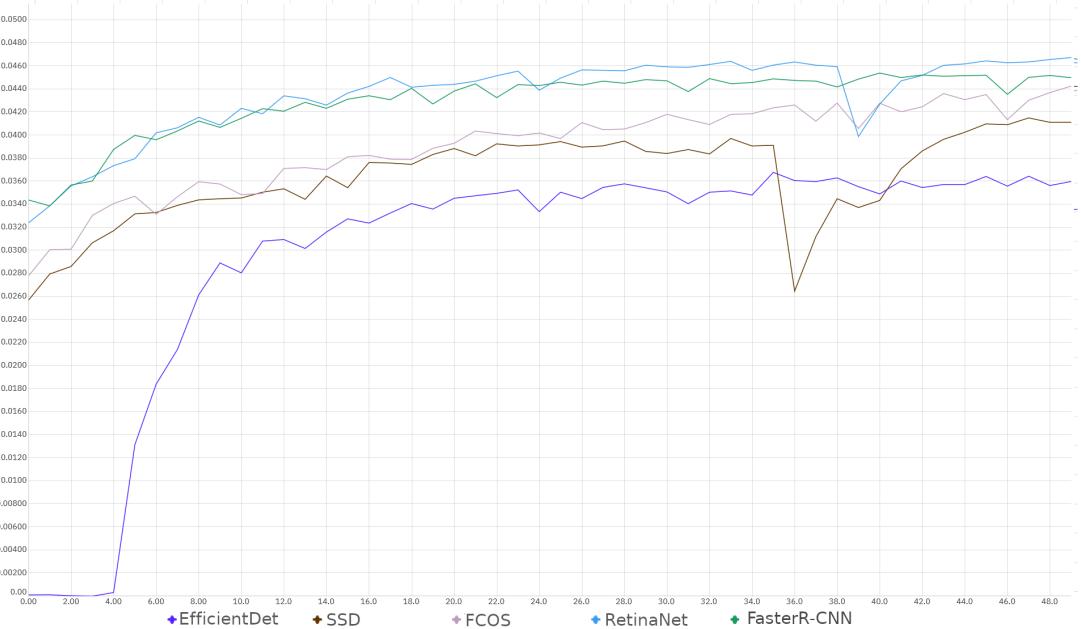


(b) Error de regresión durante la validación.

Figura 7.2: Errores de regresión de los modelos a lo largo del ciclo de entrenamiento.



(a) Precisión media de los modelos.



(b) Recall media de los modelos.

Figura 7.3: Métricas de evaluación durante el entrenamiento.

	Duración	CPU	GPU	Memoria
EfficientDet	33 m	6.9 %	62 %	13.2 GB
FasterR-CNN	58 m	5.5 %	96 %	51.4 GB
RetinaNet	1 h	5.0 %	73 %	25.5 GB
FCOS	41 m	4.1 %	11 %	34.4 GB
SSD	23 m	3.8 %	73 %	18.2 GB

Tabla 7.1: Duración y consumo de recursos (CPU, GPU y memoria) en el entrenamiento.

	Tamaño	Parámetros	Entrenables	No entrenables
EfficientDet	467.619 MB	116.0 M	116.0 M	492 K
FasterR-CNN	167.021 MB	41.8 M	15.0 M	26.8 M
RetinaNet	136.060 MB	34.0 M	6.7 M	27.3 M
FCOS	129.078 MB	32.3 M	4.9 M	27.3 M
SSD	142.567 MB	35.6 M	12.7 M	22.9 M

Tabla 7.2: Tamaño y número de parámetros entrenables y no entrenables de cada modelo.

de parámetros entrenables de cada modelo (tabla 7.2), descartando los propios de la columna vertebral.

Si bien los porcentajes de confianza de los modelos con aprendizaje por transferencia tienen más dispersión que los que han sido reentrenados por completo, los primeros son capaces de inferir acertadamente (figura 7.4), produciendo detecciones similares a los segundos. Como es comprensible la predicción de la red se ve disminuida^{7.5} ya que no adapta su mapa de características al contexto específico del conjunto de entrenamiento, sino que recicla el de su columna vertebral ya entrenada.

7.2.4 Comparación

La técnica de *transfer learning* permite reducir el consumo de recursos durante el proceso de entrenamiento, aunque la diferencia entre la duración de ambas aproximaciones es mínima. En general, las redes replican el aprendizaje sin problema, consiguiendo adaptarse al nuevo contexto, no obstante la precisión se ve notablemente afectada. Esto se debe a que entrenando la forma en la que la red detecta características con la columna vertebral el modelo ajusta las predicciones más exhaustivamente.

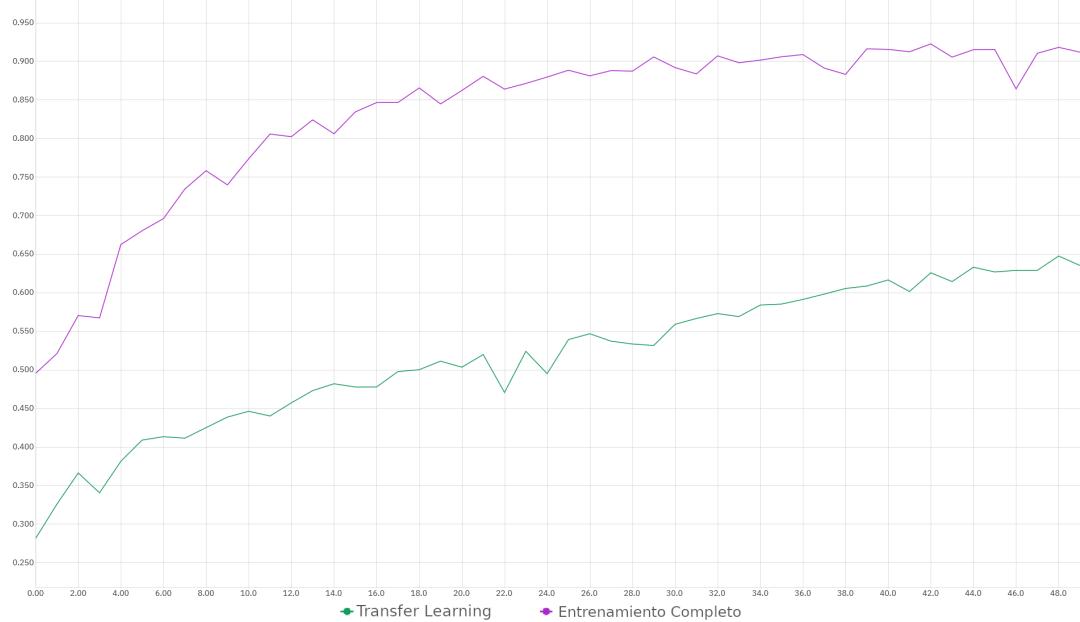


Figura 7.4: Inferencias del modelo Faster R-CNN con entrenamiento completo (azul) y aprendizaje por transferencia (rojo).

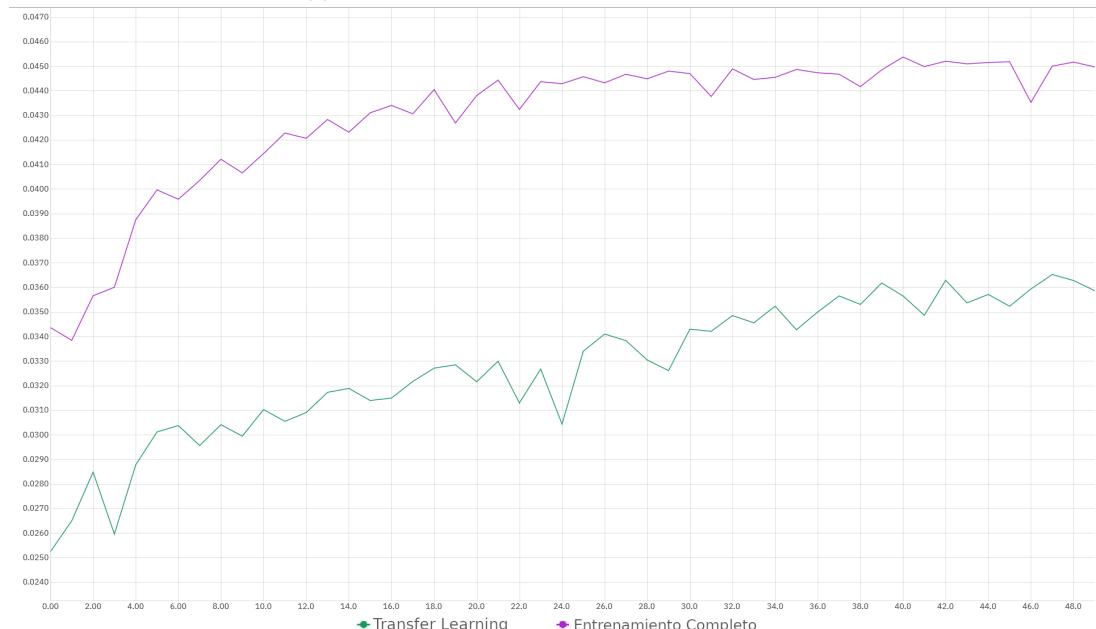
7.3 Umbralización

En esta sección se comentan los experimentos llevados a cabo para ajustar el número de detecciones de las redes, teniendo en cuenta los umbrales ajustables del sistema.

En base al mecanismo de umbralización llevamos a cabo una serie de experimentos con inferencias de los modelos para ver en qué medida cambiaban los resultados del sistema. Como observamos, sin filtrar los resultados, el modelo genera predicciones con porcentajes de confianza muy variables y tamaños dispares (figura 7.6). Si bien las predicciones empleando los dos umbrales de solapamiento y de confianza por separado producen detecciones peores, cuando se combinan ambos estos mejoran considerablemente. Debido a que una vez entrenadas las redes un mayor número de veces, estas producen detecciones con menor variabilidad, estando las más fiables en porcentajes muy altos y las menos en los más bajos. Por esto una variación de ambos umbrales a partir de 0.5 no ofrece mejoras sustanciales.



(a) Precisión media del modelo Faster R-CNN.



(b) Recall media del modelo Faster R-CNN.

Figura 7.5: Comparación de métricas con ambos enfoques de entrenamiento

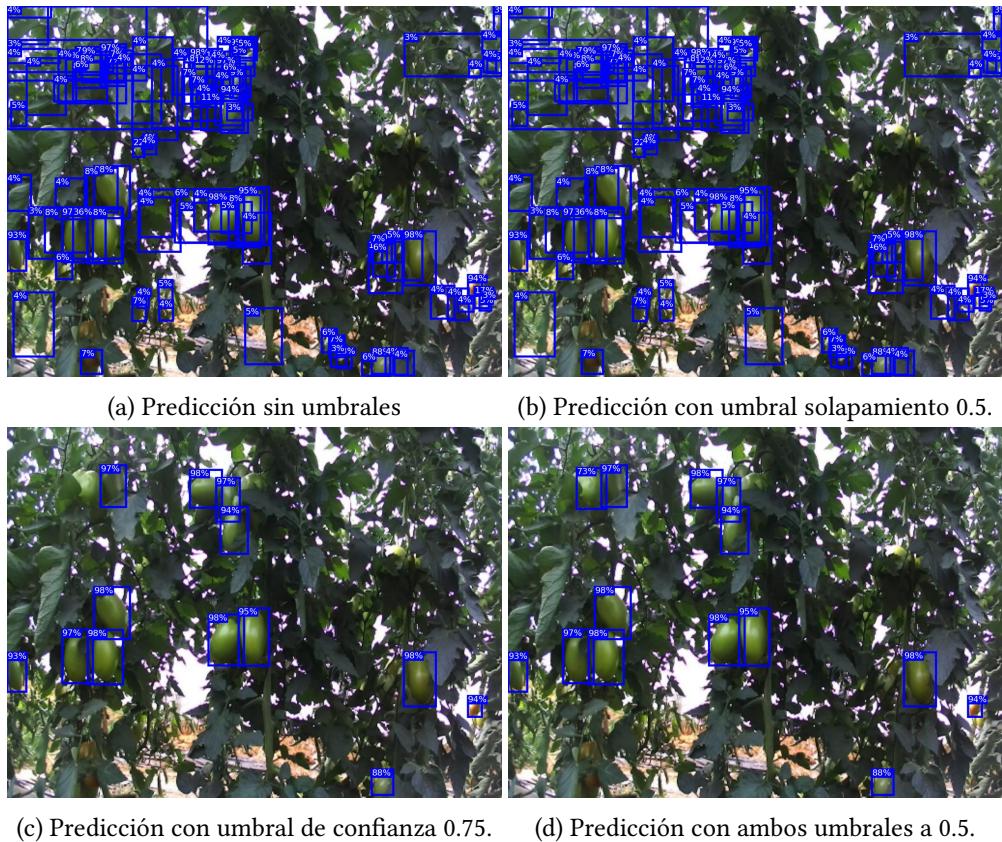


Figura 7.6: Comparación de resultados con umbrales de confianza y solapamiento.

Capítulo 8

Conclusiones

En este capítulo se comentan las conclusiones extraídas de los resultados del sistema, incluidas las críticas al proyecto en si, y las posibles líneas de trabajo futuro.

8.1 Conclusiones

El principal objetivo de este proyecto es estudiar cómo mejorar la aplicación de redes de neuronales convolucionales (CNNs) de última generación en la tarea de detectar tomates en condiciones reales de operación en un invernadero.

En primer lugar, y a diferencia de los *benchmarks* imperantes en el ámbito de la CNNs, nuestra investigación nos ha confirmado nuestras sospechas de que en esta aplicación hay muy pocos datasets públicos disponibles y todos ellos con muy pocas imágenes etiquetadas. Además, sólo uno de ellos refleja las condiciones normales de operación en un entorno real de trabajo, como son la distribución irregular de objetos, la oclusión por las hojas o la gran variabilidad en las condiciones de luz, textura y perspectiva de la cámara. Aunque de nuevo, el número de datos es reducido y se limita a un único invernadero y una única captura de datos.

En segundo lugar, en este trabajo hemos analizado los modelos de acceso público de las nuevas redes convolucionales para determinar cuáles se podrían adaptar mejor a nuestro problema. Después de varias pruebas, nos hemos decantado finalmente por EfficientDet, SSD, FasterR-CNN, FCOS y RetinaNet. Los cuatro últimos se encuentran en la librería Torchvision, siendo el primero una implementación pública tomada como referencia por la comunidad de PyTorch.

De las pruebas realizadas podemos extraer dos conclusiones principales. La primera es sobre la posibilidad de adaptar modelos de aprendizaje profundo de última generación, comparando su rendimiento y costes de entrenamiento. Como hemos visto en el capítulo anterior, modelos novedosos como EfficientDet, Faster R-CNN o RetinaNet son capaces de registrar re-

sultados satisfactorios, pudiendo adaptar el consumo de recursos según el enfoque de entrenamiento. Así, un entrenamiento completo de las redes propicia mejores detecciones, invirtiendo más carga de procesamiento y alojamiento de memoria, estando estos valores más reducidos cuando se aplica un aprendizaje por transferencia, teniendo un coste en la precisión de las redes.

La segunda conclusión está relacionada con el proceso de umbralización integrado en las redes del sistema. En este caso, hemos observado que un filtrado adaptativo no es necesario para la tarea si aplicamos otras técnicas. De esta forma, un entrenamiento exhaustivo de los modelos combinado con una umbralización estática/única produce resultados lo suficientemente satisfactorios.

8.2 Trabajo futuro

Como líneas de trabajo futuro en base a este proyecto hemos contemplado varias opciones:

- Entrenamiento con vídeos completos en lugar de imágenes individuales, por lo que se podría usar información espacio-temporal: los objetos deberían verse en imágenes consecutivas, pero con diferentes perspectivas y condiciones de iluminación, occlusiones, etc.
- Integración de un mecanismo que permita registrar anotaciones en base a predicciones del modelo que sean correctas y no estén anotadas.
- Generalización del sistema para trabajar en modo *Stream Learning*, es decir, intercalando aprendizaje e inferencia, lo que permitiría aplicar técnicas de auto-etiquetado.
- Ampliar el modelo para incorporar información de usuarios expertos a modo de nuevas etiquetas y/o corrección de las inferencias realizadas. Idealmente, durante el *Stream Learning*.

Apéndices

Apéndice A

Material adicional

En este capítulo se detalla la estructura del repositorio del proyecto, abordando los principales archivos y carpetas con una breve descripción.

A.1 Estructura del repositorio

El repositorio empleado para este proyecto^[32] contiene los módulos que implementan las funcionalidades del sistema. No existe interfaz gráfica, por lo que los ficheros se ejecutan directamente desde línea de comandos o desde el intérprete de Python. Los principales archivos son:

- **Modelos.** En esta carpeta residen los archivos que usan el módulo Lightning para modelar las arquitecturas. Dentro hay un fichero por modelo, aparte de uno que usan todos con las funciones comunes de cálculo de error y umbralización.
- **EfficientDet.** En esta carpeta se recogen los ficheros de implementación de la arquitectura y dataset específicos de EfficientDet.
- **Model.py.** Este fichero recoge las funcionalidades básicas de los modelos, tales como la carga de pesos, congelación de capas o inferencia.
- **TomatoDataset.py.** Este fichero contiene el procesamiento de los datos para crear un *DataModule* usado por los modelos. Tiene una división jerárquica de datamodules, datasets y adaptadores.
- **config.py.** Recoge los valores de configuración del sistema que afectan al resto de módulos.
- **train.py.** Módulo diseñado para entrenar un modelo, pudiendo especificar si congelar los pesos y guardarlo. Incluye funciones del módulo de monitorización Neptune.ai.

- **README.md.** Fichero en el que se especifican los comandos necesarios para poder ejecutar cada funcionalidad.

Bibliografía

- [1] Z. Zhang, C. Xie, J. Wang, L. Xie, and A. L. Yuille, “Deepvoting: A robust and explainable deep network for semantic part detection under partial occlusion,” 2018.
- [2] K. Saleh, S. Szenasi, and Z. Vamossy, “Occlusion handling in generic object detection: A review,” in *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE, Jan. 2021. [En línea]. Disponible en: <http://dx.doi.org/10.1109/SAMI50585.2021.9378657>
- [3] L. Alzubaidi, J. Bai, A. Al-Sabaawi, J. Santamaría, A. S. Albahri, B. S. N. Al-dabbagh, M. A. Fadhel, M. Manoufali, J. Zhang, A. H. Al-Timemy, Y. Duan, A. Abdullah, L. Farhan, Y. Lu, A. Gupta, F. Albu, A. Abbosh, and Y. Gu, “A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications,” *Journal of Big Data*, vol. 10, no. 1, p. 46, Apr 2023. [En línea]. Disponible en: <https://doi.org/10.1186/s40537-023-00727-2>
- [4] “Logic and artificial intelligence,” <https://plato.stanford.edu/entries/logic-ai/>, accessed: 2024-02-15.
- [5] Varios, “Wikipedia: Convolutional neural networks,” 2023, consultado el 15 de febrero de 2024. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Convolutional_neural_network
- [6] R. Solovyev, W. Wang, and T. Gabruseva, “Weighted boxes fusion: Ensembling boxes from different object detection models,” *Image and Vision Computing*, vol. 107, p. 104117, Mar. 2021. [En línea]. Disponible en: <http://dx.doi.org/10.1016/j.imavis.2021.104117>
- [7] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, “Distance-iou loss: Faster and better learning for bounding box regression,” 2019.
- [8] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, “Generalized intersection over union: A metric and a loss for bounding box regression,” 2019.

- [9] W. Falcon and T. P. L. team, “Pytorch lightning,” 3 2019. [En línea]. Disponible en: <https://www.pytorchlightning.ai>
- [10] N. S. Detlefsen, J. Borovec, J. Schock, A. Harsh, T. Koker, L. D. Liello, D. Stancl, C. Quan, M. Grechkin, and W. Falcon, “Torchmetrics - measuring reproducibility in pytorch,” 2 2022. [En línea]. Disponible en: <https://www.pytorchlightning.ai>
- [11] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, “Albumentations: Fast and flexible image augmentations,” *Information*, vol. 11, no. 2, 2020. [En línea]. Disponible en: <https://www.mdpi.com/2078-2489/11/2/125>
- [12] L. Perez and J. Wang, “The effectiveness of data augmentation in image classification using deep learning,” 2017.
- [13] R. Solovyev, “Weighted-boxes-fusion,” <https://github.com/ZFTurbo/Weighted-Boxes-Fusion>, 2019.
- [14] N. team, “neptune.ai,” 2 2019. [En línea]. Disponible en: <https://neptune.ai/>
- [15] “What is agile development?” <https://www.opentext.com/what-is/agile-development>, accessed: 2024-02-14.
- [16] “Manifesto for agile software development,” <http://agilemanifesto.org/iso/en manifesto.html>, accessed: 2024-02-14.
- [17] “Principles behind the agile manifesto,” <http://agilemanifesto.org/principles.html>, accessed: 2024-02-14.
- [18] “MLOps Principles,” <https://ml-ops.org/content/mlops-principles>, accessed: 2024-02-14.
- [19] M. Tan, R. Pang, and Q. V. Le, “Efficientdet: Scalable and efficient object detection,” 2020.
- [20] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” 2018.
- [21] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, *SSD: Single Shot MultiBox Detector*. Springer International Publishing, 2016, p. 21–37. [En línea]. Disponible en: http://dx.doi.org/10.1007/978-3-319-46448-0_2
- [22] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” 2016.
- [23] Z. Tian, C. Shen, H. Chen, and T. He, “FCOS: Fully Convolutional One-Stage Object Detection,” 2019.

- [24] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature pyramid networks for object detection,” 2017.
- [25] laboroai, “Laboro tomato: Instance segmentation dataset,” 2020, <https://github.com/laboroai/LaboroTomato> [Last accessed: (01/30/2024)].
- [26] up2metric, “tomatod,” 2020, <https://github.com/up2metric/tomatOD> [Accessed: (01/30/2024)].
- [27] V. Tsironis, S. Bourou, and C. Stentoumis, “Tomatod: Evaluation of object detection algorithms on a new real-world tomato dataset,” *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLIII-B3-2020, pp. 1077–1084, 2020. [En línea]. Disponible en: <https://isprs-archives.copernicus.org/articles/XLIII-B3-2020/1077/2020/>
- [28] G. Moreira, S. A. Magalhães, T. Padilha, F. N. dos Santos, and M. Cunha, “RpiTomato dataset: Greenhouse tomatoes with different ripeness stages,” 2021.
- [29] S. A. Magalhães, L. Castro, G. Moreira, F. N. dos Santos, M. Cunha, J. Dias, and A. P. Moreira, “Evaluating the single-shot multibox detector and yolo deep learning models for the detection of tomatoes in a greenhouse,” *Sensors*, vol. 21, no. 10, p. 3569, May 2021. [En línea]. Disponible en: <http://dx.doi.org/10.3390/s21103569>
- [30] I. P. Castro, “IMCV - External internship: Detection of tomatoes in a greenhouse,” 2021, comunicación personal.
- [31] A. Mao, M. Mohri, and Y. Zhong, “Cross-entropy loss functions: Theoretical analysis and applications,” 2023.
- [32] N. Martínez González, “*tfgtomates*,” 2022.