UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Segmentación de Manos en Imágenes en Primera Persona empleando Deep Learning

Autor:

Veganzones Parellada, Miguel

Tutor:

De La Fuente López, Eusebio

Dpto. Ingeniería de Sistemas y Automática

Valladolid, junio 2022.

Resumen

Durante los últimos años, el aprendizaje automático ha permitido avances significativos en gran cantidad de áreas del conocimiento, y especialmente en ingeniería. Siguiendo esta tendencia, este trabajo busca contribuir a un sistema de rehabilitación de manos potenciado por el uso de realidad aumentada haciendo uso de las herramientas de *Deep Learning* más utilizadas en aplicaciones de visión artificial. Se propone a continuación un sistema automático de detección de manos en imágenes, mostrando además su proceso de diseño. Los resultados obtenidos demuestran que es posible utilizar las arquitecturas sencillas utilizadas para realizar segmentación semántica en ciertos problemas de segmentación de instancias con buenos resultados.

Palabras clave

Palabras clave: Reconocimiento de manos; Segmentación semántica; Segmentación de instancias; Aprendizaje automático; Arquitecturas convolucionales

Abstract

In recent years, Machine Learning has enabled a great amount of progress in many disciplines, especially engineering. Following this trend, this project aims to contribute an augmented reality-based hand function rehabilitation system using state-of-the-art Deep Learning solutions for computer vision applications. This proyect proposes an automated hand detection system and shows its design process. Results show that the simple architectures used in semantic segmentation can also be successfully used in some instance segmentation problems.

Key words

Keywords: Hand recognition; Semantic segmentation; Instance segmentation; Machine Learning; Convolutional architectures

Índice general

1. Introducción **1**
1.1. Justificación 2
1.2. Objetivos 3
1.3. Estado del arte 4

2. Metodología **7**
2.1. Alternativas actuales 7
2.1.1. Detectores de dos etapas 7
2.1.2. Detectores de una etapa 9
2.2. Solución elegida 11
2.2.1. U-Net 13
2.2.2. ResNet 15

3. Dataset **17**
3.1. Procesado 19
3.2. Data augmentation 20
3.3. Reparto del dataset 24

4. Arquitecturas consideradas **27**
4.1. Detalles de la arquitectura 28
4.1.1. U-Net 28
4.1.2. ResNet 29
4.2. Primeros resultados 30
4.2.1. Elección de la arquitectura 31
4.2.2. Análisis del efecto del dropout 35

5. Ajuste de hiperparámetros **41**
5.1. Entrenamiento 42
5.2. Pruebas realizadas 45
5.2.1. Función de activación 45
5.2.2. Función de pérdida 47
5.2.3. Ratio de aprendizaje 52

6. Resultados **55**
6.1. Visualización de los resultados 55
6.2. Comparación con otros modelos 56
6.3. Análisis de los resultados 57

7. Conclusiones **65**
7.1. Conclusiones 65
7.2. Líneas futuras 66

A. Support Vector Machines **71**

B. Métrica de la similitud media entre centroides **75**

Índice de figuras

1.1. Ejemplo de interacción con una bola virtual en un sistema de realidad aumentada. [*ITAP Medical Robotics - Research*] 2
1.2. Sistema general. [*ITAP Medical Robotics - Research*] 3
1.3. Máscara de segmentación superpuesta sobre la imagen original. [Bambach *et al.*, 2015] 4

2.1. Esquema de funcionamiento de R-CNN [Girshick *et al.*, 2014] 8
2.2. Esquema de funcionamiento de YOLOv1 [Redmon, Divvala *et al.*, 2016] 10
2.3. Arquitectura de YOLOv1 [Redmon, Divvala *et al.*, 2016] 10
2.4. Arquitectura de la U-Net original 14
2.5. Estrategia seguida por la U-Net para procesar imágenes 14
2.6. Bloque residual 15

3.1. Cuatro imágenes de EgoHands con sus máscaras de segmentación. [Bambach *et al.*, 2015] 18
3.2. Imagen de ejemplo con su máscara de segmentación y mapa de pesos asociado. 21
3.3. Ecuación 3.10 21
3.4. Ejemplo de volteo horizontal como método de data augmentation. . . 22
3.5. Ejemplo de zooms aleatorios como método de data augmentation. . . 23
3.6. Ejemplo de añadir un offset a todos los canales de imágenes como método de data augmentation 23
3.7. Ejemplo de modificar la saturación en imágenes como método de data augmentation. 24

4.1. Arquitectura U-Net utilizada 29
4.2. Arquitectura ResNet utilizada 30
4.3. Evolución del IoU en entrenamientos de la U-Net con problemas de convergencia. 32
4.4. Evolución del IoU en dos entrenamientos de la ResNet y dos entrenamientos de la U-Net con dos dropout rates diferentes. 34
4.5. Segmentaciones de dos imágenes obtenidas con una ResNet y una U-Net entrenadas sin dropout. 35
4.6. IoU de entrenamiento en la misma arquitectura ResNet en función de la cantidad de dropout utilizada durante el entrenamiento. 36
4.7. Evolución del IoU de entrenamiento y validación durante el entrenamiento de la arquitectura ResNet propuesta entrenada con dropout rates de 0.0 y 0.1. 37

4.8. Evolución del IoU de entrenamiento y validación en entrenamientos de la arquitectura ResNet propuesta con cuatro dropout rates diferentes. 39
4.9. Ejemplo de segmentaciones obtenidas con uno de los modelos ResNet entrenados con un dropout rate de 0.3. 40

5.1. Topología del loss en dos arquitecturas que difieren únicamente en el uso de conexiones residuales. 42
5.2. Representación gráfica de las funciones ReLU (izquierda) y PReLU (derecha). [*Papers with Code - PReLU Explained*] 46
5.3. Función logística utilizada para aproximar una binarización de la segmentación inferida. 49
5.4. Valor de Focal Loss con varios valores de γ en función de la confianza de la clase correcta. 50
5.5. Comparación de la evolución del IoU de entrenamiento y validación durante el entrenamiento de una arquitectura ResNet con PReLUs utilizando BCE y DICE Loss. 51
5.6. Evolución del learning rate con varios planificadores. 53
5.7. IoU de entrenamiento y validación con cuatro planificaciones diferentes del learning rate. 54

6.1. Ejemplos de buenas segmentaciones (IoU >0.8) incluso en casos exigentes. En magenta se muestra el *ground truth* y en amarillo las máscaras inferidas. 58
6.2. Segmentaciones con buen IoU pero que no han proporcionado una máscara de segmentación para cada instancia. En magenta se muestra el ground truth y en amarillo las máscaras inferidas. 59
6.3. Manos correctamente segmentadas con una máscara para cada instancia en dos casos difíciles. En magenta se muestra el ground truth y en amarillo las máscaras inferidas. 60
6.4. Ejemplos de malas segmentaciones (IoU <0.6) en casos difíciles por diversos motivos. En magenta se muestra el ground truth y en amarillo las máscaras inferidas. 61
6.5. Salida del primer bloque convolucional del encoder. 62
6.6. Salida de un bloque convolucional intermedio del encoder. 62
6.7. Salida del cuello de botella de la arquitectura, último bloque convolucional del encoder, y entrada del decoder. 63
6.8. Salida de un bloque convolucional intermedio del decoder. 63
6.9. Salida del bloque final del decoder. 64
6.10. Salida de la red y máscara de segmentación obtenida de binarizar la salida. 64

A.1. Función de decisión en un espacio de dos dimensiones 72

B.1. Métrica calculada en cuatro segmentaciones obtenidas con distintos modelos. 78

Índice de tablas

4.1. Rendimiento obtenido con las arquitecturas U-Net y ResNet elegidas en función de la cantidad de *dropout* utilizado. 33
4.2. Resultados obtenidos con la arquitectura ResNet elegida en función de la cantidad de dropout utilizado 38

5.1. Resultados obtenidos utilizando ReLUs y PReLUs como función de activación en el mismo modelo. Resultados promediados de tres entrenamientos idénticos. 46
5.2. Efecto de los parámetros de Focal Loss en el aprendizaje de la arquitectura ResNet con ReLUs. 50
5.3. Resultados obtenidos con varias funciones de loss en la arquitectura ResNet considerada y PReLUs. 51
5.4. Resultados obtenidos con varias planificaciones del ratio de aprendizaje. 53

6.1. Comparación de resultados obtenidos sobre EgoHands utilizando la misma cantidad de imágenes originales para entrenar. 56

Capítulo 1

Introducción

Es evidente que las imágenes contienen gran cantidad de información útil, y por ello pueden ayudar a inferir información concreta de un sistema de forma análoga a cómo se percibe a través del sentido de la vista. Es por esto que, poder obtener esta información de forma automática tiene innumerables aplicaciones, especialmente en la industria y medicina.

La visión artificial es el campo de la ciencia que busca analizar imágenes de forma automática a partir de detectar patrones y secuencias en ellas. Con las herramientas que se han desarrollado en este campo, es fácil implementar algoritmos capaces de resolver problemas simples en entornos controlados que puedan encontrar formas, bordes y otro tipo de características sencillas. Pero estas heurísticas fallan a la hora de extraer información de alto nivel de imágenes, y es necesario por ello recurrir a sistema más sofisticados en muchos casos.

El aprendizaje automático ha permitido gran cantidad de progreso en los problemas más complicados que contempla la visión artificial. Estos son: clasificación de imágenes, detección de objetos, segmentación semántica y segmentación de instancias. Todos estos problemas tienen en común que son difíciles porque no existe una única representación concreta de cualquier concepto u objeto, y por ello la información que se busca es abstracta, difusa y está deslocalizada. La principal ventaja que introduce el aprendizaje automático es la capacidad de extraer de forma automática características representativas de objetos de forma eficiente a partir de ejemplos.

La segmentación de imágenes, que consiste en clasificar todos los píxeles de una imagen, es un problema interesante porque permite inferir información precisa sobre la geometría y la localización de objetos, lo que tiene gran cantidad de aplicaciones. Cabe destacar su uso en sistemas de conducción autónoma para distinguir la calzada de otros elementos, y en la detección de anomalías en imágenes médicas.

Este proyecto presenta el diseño de un sistema automático para la segmentación de manos en imágenes en tiempo real. Las manos están involucradas en gran cantidad de actividades, y por ello se puede extraer información importante a partir de ellas en muchas aplicaciones.

En concreto, el sistema diseñado está pensado para ser utilizado en un siste-

ma de rehabilitación de manos potenciado por el uso de realidad aumentada. Este sistema de rehabilitación es un proyecto del departamento de Robótica Médica del Instituto de las Tecnologías Avanzadas de la Producción (ITAP) de la Universidad de Valladolid, grupo dentro del que se ha desarrollado este trabajo.

1.1. Justificación

Las manos juegan un papel central en gran cantidad de actividades, y conocer información sobre su posición y configuración a lo largo del tiempo puede ser muy útil a la hora de analizar sistemas físicos donde intervienen personas.

Segmentar manos aporta información sobre su posición, y puede ser muy útil para extraer información más concreta una vez se han localizado. Una aplicación práctica podría ser evaluar el desempeño de actividades de forma automática a partir de un seguimiento de las manos.

El resultado de segmentar una imagen es un conjunto de máscaras de segmentación que facilitan aislar y descontextualizar los objetos encontrados. Obtener estas máscaras pueden facilitar en gran medida procesados más complejos, como inferir la configuración de la mano.

El proyecto al que se pretende contribuir con este trabajo es RobHand [*ITAP Medical Robotics*], un sistema de rehabilitación de mano y muñeca basado en terapia espejo y que utiliza un exoesqueleto robótico, realidad aumentada e interfaces de juegos para facilitar el proceso. Este sistema toma de forma automática datos de los usuarios para regular el grado de asistencia que ofrece el exoesqueleto a cada caso particular y en función del progreso del paciente.

La realidad aumentada tiene potencial para aumentar la eficacia de sistemas de rehabilitación de la capacidad motora en extremidades perdida por accidentes cerebrovasculares, y especialmente en las extremidades superiores [Phan *et al.*, 2022; Saposnik *et al.*, 2011]. En este caso se utilizará realidad aumentada para permitir al paciente interactuar con objetos virtuales; véase la figura 1.1.

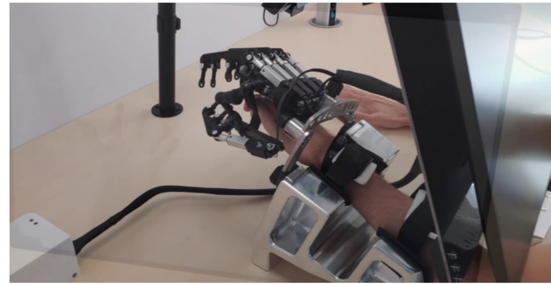
Figura 1.1: Ejemplo de interacción con una bola virtual en un sistema de realidad aumentada. [*ITAP Medical Robotics - Research*]

Segmentar las manos permitirá recortarlas de la imagen para poder colocar objetos por delante o por detrás y determinar sus interacciones. Utilizar sistemas de

realidad aumentada de esta forma permite utilizar una amplia variedad de actividades en la rehabilitación de forma sencilla.

Para poder utilizar realidad aumentada, se interpondrá una pantalla entre el paciente y sus manos, y esto hace necesario encontrar la forma y localización de ambas manos en tiempo real. La pantalla y el sistema completo se muestran en las figuras 1.2a y 1.2b.

(a) Pantalla utilizada en el sistema de rehabilitación considerado.



(b) Sistema completo.

Figura 1.2: Sistema general. [*ITAP Medical Robotics - Research*]

Interponer una pantalla entre el paciente y sus manos permite además ocultar el exoesqueleto durante una terapia espejo, permitiendo una rehabilitación aún más inmersiva.

1.2. Objetivos

El objetivo de este proyecto es obtener un sistema de detección capaz de inferir máscaras de segmentación precisas, respetando las restricciones temporales impuestas por la aplicación. Las máscaras de segmentación aportan información sobre la localización y el contorno de objetos a nivel de píxel. En la figura 1.3 se muestra un ejemplo de una de estas segmentaciones.

Este sistema de detección se implementará haciendo uso de *Deep Learning* y otras herramientas de aprendizaje automático que han permitido grandes avances en el campo de la inteligencia artificial y la visión artificial. En concreto se utilizarán modelos de redes neuronales basados en arquitecturas convolucionales profundas.

Por simplicidad, el sistema de detección diseñado detectará manos de forma genérica, sin tener en cuenta los detalles concretos de la aplicación considerada. Una vez diseñado, es sencillo adaptar el sistema a una aplicación específica más concreta que introduzca sus propias restricciones. Esto facilita el desarrollo y la obtención del modelo deseado a la vez que le aporta generalidad. La única restricción impuesta por la aplicación que se tendrá en cuenta es que las imágenes en las que se debe inferir están tomadas en primera persona. Esto es debido a que la cámara que toma las imágenes se sitúa entre el paciente y sus manos, detrás de la pantalla necesaria

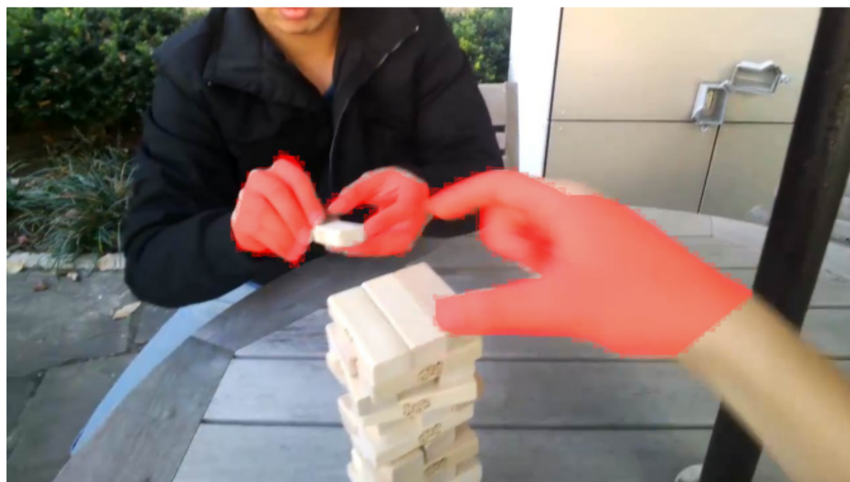


Figura 1.3: Máscara de segmentación superpuesta sobre la imagen original. [Bambach *et al.*, 2015]

por el uso de realidad aumentada.

Los objetivos de este proyecto son por lo tanto:

- Análisis de diferentes modelos utilizados en aplicaciones similares para encontrar la arquitectura más adecuada a esta aplicación.
- Procesado del conjunto de datos utilizado para adaptarlo al problema considerado.
- Análisis del efecto de diferentes hiperparámetros en el entrenamiento de redes convolucionales, buscando una combinación adecuada de los mismos.
- Evaluar objetivamente los resultados obtenidos en base a las métricas más importantes, impuestas por las restricciones de la aplicación concreta considerada y las restricciones del problema genérico.
- Obtener un modelo con la mejor combinación de hiperparámetros encontrada.
- Analizar el modelo diseñado para obtener una intuición sobre los resultados obtenidos y justificarlos.

1.3. Estado del arte

La segmentación de objetos es un problema bien conocido. Gran parte de las herramientas utilizadas en aplicaciones de visión artificial están ampliamente consolidadas y su eficacia está demostrada. En concreto el uso de redes neuronales es la norma por los grandes avances que ha habido en el campo del *Deep Learning*, y por los avances que el *Deep Learning* ha permitido en el análisis de imágenes y el aprendizaje automático en general.

En el campo de la visión artificial se utilizan redes neuronales parcial o totalmente convolucionales (*Convolutional Neural Networks*, CNN). Esto se debe al conjunto

de propiedades que presenta el uso de convoluciones, principalmente porque permiten codificar relaciones espaciales en imágenes de forma implícita, son invariantes ante translaciones y permiten un comportamiento muy sofisticado con pocos parámetros. Por ello se utilizan convoluciones siempre en las primeras capas de la red para extraer información de las imágenes.

El principal progreso reciente en el campo de la visión artificial se debe a la investigación en arquitecturas de redes y los hiperparámetros que determinan la evolución de los modelos durante el entrenamiento. Las arquitecturas más modernas han alcanzado una gran capacidad de predicción incluso en problemas muy complicados, son fáciles de entrenar y pueden inferir a una velocidad elevada.

Parte del progreso se debe a la accesibilidad de las herramientas utilizadas en *Deep Learning* gracias a la gran cantidad de librerías de álgebra lineal y aprendizaje automático disponibles. Las dos librerías más ampliamente utilizadas son TensorFlow [*TensorFlow*] y PyTorch [*PyTorch*], aunque existen muchas otras, como Apache MXNet [*Apache MXNet*].

Estas librerías tienen interfaces en lenguajes como C++ y Python, y tienen soporte para GPUs y otros aceleradores. Python es el lenguaje de programación estándar *de facto* en el campo del aprendizaje automático por ser un lenguaje de código abierto de muy alto nivel que tiene la facilidad en la lectura del código como un factor de diseño. Utilizar una interfaz de alto nivel se consigue sin apenas pérdidas en rendimiento ya que estas librerías están implementadas en C++ y CUDA C/C++ principalmente, y por lo tanto la mayor parte del trabajo computacionalmente intensivo se realiza por código compilado muy optimizado.

En este proyecto se ha utilizado TensorFlow, en concreto su versión 2.0 por contar con una interfaz sencilla de muy alto nivel llamada Keras [*Keras: the Python deep learning API*] y por ser la librería más popular a día de hoy. TensorFlow cuenta además con muy buena documentación, es una librería completa, actualizada y optimizada, y cuenta con una amplia comunidad.

En visión artificial hay cuatro tipos de problemas principales que se buscan resolver con aprendizaje automático. Estos son: clasificación de imágenes, detección de objetos, segmentación semántica y segmentación de instancias. Estos problemas son fundamentalmente diferentes y por lo general cada uno es más complicado que el anterior.

En los últimos años ha habido avances en todos estos problemas, y sobre todo han aparecido nuevas arquitecturas capaces de segmentar instancias con una eficiencia elevada. Cabe destacar *Mask DINO* [Zhang *et al.*, 2022], modelo que ha obtenido los mejores resultados en segmentación de instancias en el *dataset* COCO [*Papers with Code - COCO test-dev Benchmark (Instance Segmentation)*].

Este trabajo busca resolver una versión menos restrictiva de segmentación de instancias. Utilizando las arquitecturas más simples utilizadas en segmentación semántica pero buscando siempre obtener máscaras individuales para cada objeto.

Esto facilita enormemente el problema a costa de obtener resultados menos consistentes.

Capítulo 2

Metodología

En este capítulo se hablará sobre los modelos de aprendizaje automático utilizados en problemas similares al objetivo de este proyecto, buscando obtener el modelo más adecuado. El problema más común que se busca resolver con este tipo de sistema es la detección de objetos, que es una versión simplificada del problema considerado, la segmentación semántica.

2.1. Alternativas actuales

Actualmente hay gran cantidad de alternativas y enfoques diferentes a la hora de detectar instancias de objetos en imágenes. Esto se debe gracias al trabajo de investigadores de todo el mundo y a los avances en computación de los últimos años. LeCun, B. Boser *et al.* [LeCun, B. Boser *et al.*, 1989] demostraron el potencial de las redes neuronales artificiales entrenadas mediante aprendizaje supervisado, utilizando el algoritmo *backpropagation*, a la hora de procesar gran cantidad de datos de bajo nivel. En este trabajo se utilizaba una red con tres capas ocultas, dos convolucionales y una densa, para clasificar imágenes de números manuscritos.

Los principales modelos utilizados para la detección de objetos se pueden clasificar en dos grupos: detectores de una etapa y detectores de dos etapas. Los detectores de dos etapas son conocidos por priorizar precisión frente a velocidad de inferencia, mientras que los de una etapa buscan velocidad, a costa de precisión, en general. Esta diferencia fundamental hace que no siempre sean intercambiables en función de las restricciones del problema que deban resolver.

2.1.1. Detectores de dos etapas

Un enfoque que ha probado gran eficacia es el de *Regions with CNN features* (R-CNN) [Girshick *et al.*, 2014], que realiza una búsqueda selectiva en las imágenes antes de realizar predicciones. En este sistema se escoge un conjunto numeroso de regiones para luego ser evaluadas por un clasificador. En el caso de R-CNN, este clasificador está formado por una red convolucional que extrae un conjunto de características de tamaño fijo de cada región y un clasificador basado en un conjunto de *Support Vector Machines* (SVM) [B. E. Boser *et al.*, s.f., véase el apéndice A] que analiza los rasgos extraídos de cada región; véase la Figura 2.1. Este modelo utiliza tres etapas, y analiza regiones de la imagen múltiples veces, por lo que el tiempo de

inferencia es elevado. Este modelo permite una gran precisión, con un uso reducido de memoria, incluso para un número elevado de clases, y una velocidad procesado de 13s/imagen en GPU. La inferencia es lenta porque se realiza una propagación hacia delante por la CNN para cada región propuesta. El entrenamiento también es lento y complicado porque sus tres etapas se deben entrenar por separado.

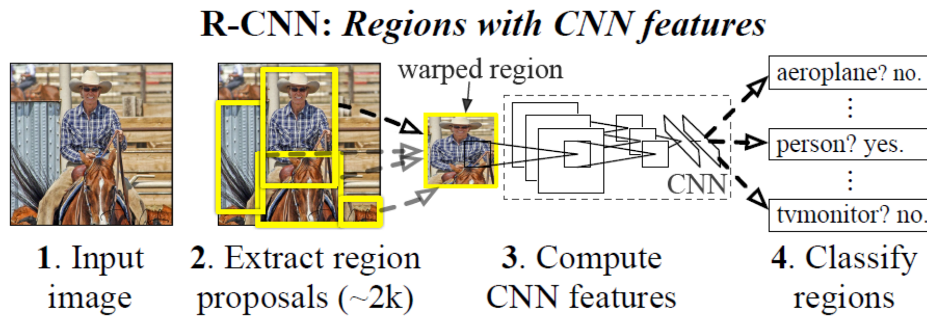


Figura 2.1: Esquema de funcionamiento de R-CNN [Girshick *et al.*, 2014]

Sobre este concepto, se construye Fast R-CNN [Girshick, 2015] utilizando VGG16 [Simonyan y Zisserman, 2015] como *backbone*[1] , una red convolucional profunda. Este modelo consigue mejor precisión que R-CNN en detección de objetos, se entrena un orden de magnitud más rápido, e infiere dos ordenes de magnitud más rápido. Esta mejora se debe a que, tras obtener el conjunto de regiones propuestas, el modelo analiza con la red convolucional la imagen completa, y luego extrae de cada región de interés un conjunto de características. En este caso, y a diferencia de R-CNN, dos regiones de interés que se solapen compartirán computación de la red convolucional.

El cuello de botella de Fast R-CNN es la etapa de proposición de regiones de interés, y por ello parte de la investigación en este tipo de arquitecturas se ha centrado en minimizar este coste. Faster R-CNN [Ren *et al.*, 2017] introduce una *Region Proposal Network* (RPN), que utiliza las mismas características extraídas por la red convolucional para predecir tanto los objetos como su posición. Faster R-CNN une la RPN con la Fast R-CNN utilizando mecanismos de atención [véase Vaswani *et al.*, 2017]. Este modelo consigue ejecutarse a una velocidad de 5fps, y además mejora la precisión sobre Fast R-CNN.

En 2017 se presentó un meta algoritmo basado en Faster R-CNN llamado Mask R-CNN para el problema de la segmentación de instancias. Este modelo comparte la primera etapa con Faster R-CNN, donde una RPN propone regiones de interés. En la segunda etapa añade, en paralelo a la predicción de clases y *bounding boxes* de Faster R-CNN, una red neuronal totalmente convolucional (FCN) para predecir máscaras de segmentación. Mask R-CNN consigue así segmentar instancias, proporcionando la localización de objetos a nivel de píxel. Este modelo añade una ligera carga computacional adicional respecto a Faster R-CNN, aunque puede ejecutarse también a una velocidad de 5fps[2] .

[1] El *backbone* es la parte de la red utilizada para extraer características, de las imágenes de entrada en este caso.

[2] En una GPU K40

Entre 2015 y 2017, los detectores basados en R-CNN consiguieron los mejores resultados en la clasificación del COCO test dataset, uno de los principales conjuntos de datos utilizados para comparar el rendimiento de los modelos. La variante actual con mejores resultados es Cascade R-CNN [Cai y Vasconcelos, 2017]. Este modelo busca obtener un detector que produzca pocos falsos positivos, entrenando secuencialmente detectores cada vez más selectivos, aprovechando la salida del anterior para entrenar el siguiente. Esto se consigue aumentando en cada detector el umbral de confianza requerido para considerar una predicción como positiva. Es necesario aumentar progresivamente este umbral debido a que incrementar este umbral fomenta el *overfitting* y porque dificulta significativamente el problema.

Estos modelos están compuestos de varios detectores más simples conectados en serie. Esto es lo que hace que sean detectores de varias etapas, y lo que limita su velocidad de inferencia, ya que cada región propuesta por el primer detector debe ser analizada individualmente por el segundo. Y, a pesar de que un *hardware* moderno puede paralelizar fácilmente esta operación, la cantidad de estas regiones ronda los pocos miles, en función del modelo.

2.1.2. Detectores de una etapa

Los detectores de una etapa detectan y estiman la posición de objetos en imágenes tras una única evaluación, lo que, en general, permite una velocidad de inferencia mucho mayor que la de los modelos de dos etapas. Este tipo de modelo se suele utilizar para el reconocimiento de objetos en tiempo real y en sistemas embebidos por tener un coste computacional reducido. Tener solo una etapa permite además entrenar el modelo de extremo a extremo para el objetivo final exclusivamente. En otros modelos de dos etapas, es necesario entrenar por separado cada una de estas, o partir de un modelo preentrenado y hacer un ajuste fino para la aplicación en concreto.

La arquitectura más conocida dentro de esta categoría es la de *You Only Look Once* (YOLO) [Redmon, Divvala *et al.*, 2016], cuya aportación fue tratar la detección de objetos como un único problema, no como la unión entre detectar y clasificar. Conseguir un detector de una etapa deriva de abordar el problema de la detección de objetos de esta manera. Como ventaja tiene una mayor velocidad y generalidad que los métodos basado en R-CNN, a costa de menor precisión. No clasificar subregiones de la imagen permite además que la red no solo aprenda a detectar rasgos característicos de los objetos, sino que también codifica información sobre su entorno, lo que se traduce en menos falsos positivos [ibíd.].

Este modelo divide la imagen en $S \times S$ celdas, y para cada una se predicen B *bounding boxes*, y las C probabilidades inferidas que corresponden a cada una de las clases posibles. Un esquema de esto se puede ver en la figura 2.2. Un *bounding box* está definido por cinco valores, cuatro coordenadas de posición y un parámetro de confianza en la predicción. Este parámetro se entrena para que prediga la *Intersection over Union* (IoU) entre el *box* inferido y cualquier *box* real. Las coordenadas de los *bounding boxes* están referidas a la celda asociada con esa predicción, pero

sus dimensiones están referidas a la imagen completa ya que se predice sobre la imagen completa, no solo sobre la propia celda. En el modelo concreto presentado por Redmon, Divvala *et al.* se predicen dos *boxes* y una categoría por celda. Este enfoque permite buenos resultados y velocidad de inferencia, pero limita la cantidad de objetos cercanos que se pueden detectar.

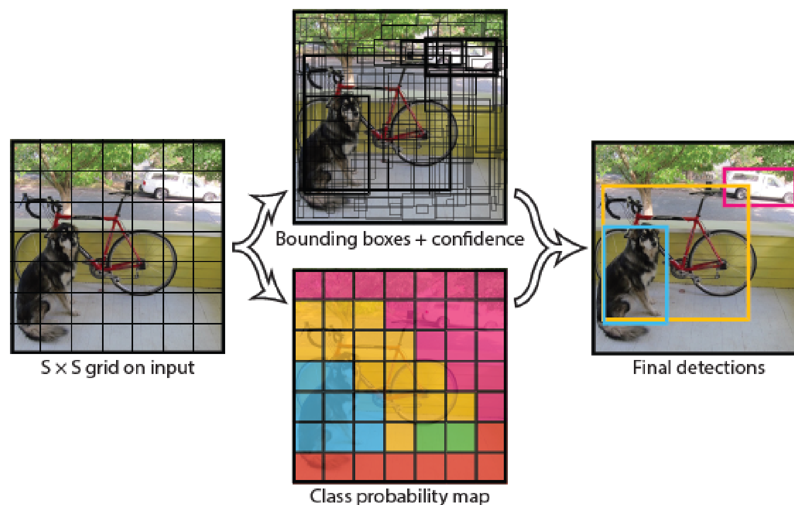


Figura 2.2: Esquema de funcionamiento de YOLOv1 [Redmon, Divvala *et al.*, 2016]

En cuanto a la arquitectura, mostrada en la figura 2.3, YOLO tiene 24 capas convolucionales y una cabeza formada por dos densas. Los *kernels* de convolución utilizados son de tamaño 3×3, salvo en la entrada que son 7×7. Utilizar *kernels* pequeños implica una elevada velocidad de ejecución, a pesar de utilizar una red convolucional profunda. Las capas convolucionales forman un encoder que extrae información de la imagen, y a partir de esta información la cabeza densa realiza predicciones. Un problema de utilizar un encoder profundo como este, es que se degradan los detalles de la imagen, y las predicciones se realizan en base a rasgos generales. Esto limita la precisión en la localización de los objetos, que es la principal fuente de errores de este modelo.

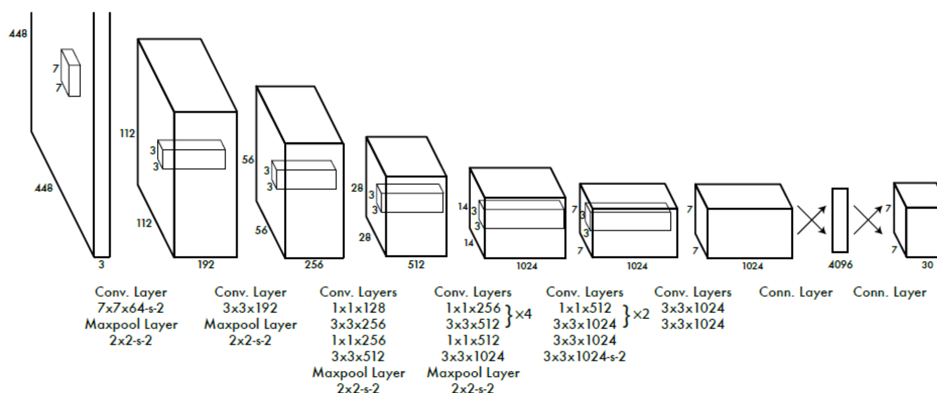


Figura 2.3: Arquitectura de YOLOv1 [Redmon, Divvala *et al.*, 2016]

El modelo YOLO base presentado puede analizar imágenes a una velocidad de

45fps[3] , nueve veces la velocidad de Faster R-CNN y Mask R-CNN. Además, se presenta junto con Fast YOLO, una versión ligera capaz de inferir a 155fps, con peor precisión por supuesto.

Redmon y Farhadi presentaron ese mismo año la segunda iteración de YOLO, YOLOv2 [Redmon y Farhadi, 2016]. Esta red utiliza Darknet-19 como *backbone*, una red con 19 capas convolucionales y 5 capas de *maxpooling* para reducir la dimensionalidad de los resultados intermedios. Este modelo presenta ligeras mejoras sobre la versión original, como el uso de *batch normalization* como método de regularización [Ioffe y Szegedy, 2015] para mejorar la convergencia de la red y reducir el sobreajuste. Además es capaz de predecir gran cantidad de clases, debido a haber sido entrenada no solo con *datasets* de detección, sino también con *datasets* de clasificación, que contienen etiquetas más precisas. Para esto fue necesario unir los dos tipos de conjuntos de datos, que no tienen etiquetas excluyentes, y entrenar partes de la red con cada tipo de muestra; véase Redmon y Farhadi [2016] para una explicación detallada del proceso.

En 2018 se presentó YOLOv3, buscando una mayor precisión que las dos versiones anteriores, aunque manteniendo la capacidad de procesar imágenes en tiempo real [Redmon y Farhadi, 2018]. Para aumentar la precisión YOLOv3 sigue la tendencia de los últimos años de utilizar redes cada vez más profundas, que ha permitido grandes avances en reconocimiento de imágenes, y utiliza Darknet-59 como *backbone*, una red con 59 capas convolucionales. Pero, como demostraron He *et al.* en 2015, una red más profunda no implica mejores resultados, y por ello YOLOv3 incluye además bloques residuales [He *et al.*, 2015a] para aprovechar el incremento en profundidad.

Este tipo de arquitectura es ampliamente utilizada para detección de objetos, pero no proporciona más información sobre la geometría de estos que sus dimensiones aproximadas. Por otro lado, el entrenamiento de estas redes requiere de conjuntos de datos muy opacos, compuestos por pares formados por una imagen y un tensor de grandes dimensiones.

2.2. Solución elegida

Las arquitecturas mostradas hasta ahora infieren la posición de objetos en imágenes, pero salvo por Mask R-CNN, no proporcionan máscaras de segmentación de estos objetos. Para el objetivo de este trabajo conviene obtener tanto la posición de los objetos como su geometría. Las arquitecturas utilizadas para la detección de objetos se podrían utilizar para proponer regiones para ser analizadas por otro tipo de red, de forma análoga a un detector de dos etapas modificado para segmentación. Esto tiene el potencial de producir resultados muy precisos, aunque a costa de complicar enormemente el entrenamiento y requerir de dos conjuntos de datos diferentes. Además es igualmente necesario introducir otro tipo de red para inferir la geometría de los objetos.

[3] En una GPU Titan X

Una forma intuitiva de detectar objetos y además obtener información sobre su geometría es segmentar la imagen, que consiste en clasificar cada uno de sus píxeles. Segmentar una imagen aporta información sobre la posición y geometría de los objetos detectados a nivel de píxel. La principal arquitectura diseñada para segmentación es la red totalmente convolucional (FCN, por sus siglas en inglés). Esta arquitectura utiliza una red convolucional como *encoder* para extraer información de alto nivel de las imágenes, y otra red neuronal convolucional como *decoder* para construir las máscaras de segmentación a partir de los rasgos extraídos. Una FCN utiliza por lo tanto el mismo tipo de codificador que YOLO para extraer información de las imágenes, pero la procesa con otra red convolucional en vez de una densa.

Una de las ventajas de las redes convolucionales a la hora de procesar imágenes es que codifican implícitamente información posicional por estar formadas por capas localmente conectadas. Información que es claramente muy importante en una imagen. Las convoluciones son además invariantes ante translación, lo que permite diseñar sistemas de detección resilientes. La red convolucional más común es el *encoder* convolucional que reduce progresivamente la resolución de las imágenes a la salida de cada capa, a la vez que aumenta los canales de profundidad. Contraer las imágenes de esta forma permite extraer más información contextual para cada píxel, además de reducir el uso de memoria del modelo.

Por otro lado, las capas densas no pueden captar información posicional fácilmente, ni son invariantes ante translación, y por ello se utilizan principalmente para procesar datos de más alto nivel obtenidos de algún procesado previo. Además de que el tamaño de las imágenes es, en general, demasiado grande como para poder ser procesado por una red densa. Las redes convolucionales tienen en comparación muchos menos parámetros, ya que en cada capa se entrena un conjunto de *kernels* de convolución independiente de la resolución de la imagen de entrada y los resultados intermedios.

Una arquitectura FCN puede entrenarse de extremo a extremo a partir de imágenes y sus respectivas máscaras de segmentación. Una máscara de segmentación es, en general, una imagen binaria con un solo canal de profundidad y una resolución igual o proporcional a la de las imágenes de entrada. El problema principal que se pretende resolver con redes totalmente convolucionales es la segmentación semántica, donde se entrena un clasificador para predecir para cada píxel la probabilidad de pertenecer a un conjunto de categorías.

Para segmentación semántica, la salida de la red es un conjunto de máscaras de segmentación, una por cada clase a detectar, más una para el fondo. Esto se corresponde con una imagen con tantos canales como clases puede pertenecer un píxel, incluyendo el fondo. La red se entrena por lo tanto con el objetivo de predecir para cada píxel la probabilidad de pertenecer a cada una de las categorías posibles. Si solo se busca segmentar una clase, se puede utilizar una salida con solo un canal de profundidad porque es suficiente para representar toda la información necesaria y hay funciones de activación que lo permiten.

Un problema similar pero intrínsecamente diferente es la segmentación de ins-

tancias. En este caso se busca clasificar cada píxel dentro de una clase y como parte de una instancia. Esto implica que hace falta codificar información sobre el tipo de objeto detectado en cada máscara, lo que no se puede conseguir directamente con una red totalmente convolucional. En una arquitectura FCN para segmentación, se asocia implícitamente cada categoría a un canal concreto de la salida durante el entrenamiento. Una solución similar se puede adoptar para una versión restringida de segmentación de instancias, en que se pretende detectar un conjunto conocido y acotado de objetos, pero no para el problema genérico.

Para segmentar instancias se utilizar arquitecturas de dos etapas parcialmente convolucionales. Hay dos variedades principales, las basadas en proposición de regiones como Mask R-CNN, y las basadas en arquitecturas FCN que incluyen un postprocesado para pasar de segmentación semántica a segmentación de instancias. Un ejemplo de una red basada en FCN es *InstanceFCN* [Dai *et al.*, 2016], que introduce el concepto de posiciones relativas para segmentar por separado diferentes regiones de cada objeto, lo que puede utilizarse para reconstruir los objetos detectados. La red convolucional se utiliza para obtener resultados parciales del estilo: *lado izquierdo de una persona* o *lado derecho de una persona*, que luego se utilizan para construir máscaras de instancias individuales sabiendo que dos lados derechos y dos lados izquierdos pertenecen, por lo general, a dos instancias diferentes.

Las dos implementaciones más populares de la arquitectura FCN son la U-Net [Ronneberger *et al.*, 2015] y la ResNet [He *et al.*, 2015a]. Estas son las dos arquitecturas que se han considerado para hacer pruebas sobre la aplicación concreta de este trabajo, que se mostrarán en los siguientes capítulos. Se intentará además, con un preprocesado del conjunto de datos, obtener una aproximación razonable a la segmentación de instancias utilizando estas arquitecturas simples.

2.2.1. U-Net

La U-Net está compuesta por un codificador y un decodificador convolucionales, e introduce conexiones a modo de atajo entre ambos; véase la figura 2.4. El codificador es una red convolucional que contrae las dimensiones de la imagen de entrada a la vez que aumenta los canales de profundidad, con el objetivo de conseguir información contextual de alto nivel. El decodificador, también convolucional, es simétrico al codificador, y construye las máscaras de segmentación a partir de la información extraída por el codificador. Las conexiones entre el codificador y el decodificador sirven para hacer llegar información de bajo nivel poco procesada hacia las últimas capas de la red. Esto permite que las capas convolucionales del decodificador tengan acceso a información posicional muy precisa, que de otra forma se podría perder fácilmente al reducir la resolución de los resultados intermedios en el codificador.

La U-Net original utiliza convoluciones sin *padding* puesto que, por limitaciones de memoria de las GPUs, está pensada para procesar imágenes de un tamaño arbitrario por secciones. Esto permite trabajar con imágenes de alta resolución sin imponer grandes restricciones sobre el *hardware*. Al procesar las imágenes por trozos tiene sentido utilizar convoluciones sin padding, ya que de otra forma se deteriora notablemente la información en los bordes. Para procesar los bordes de la imagen

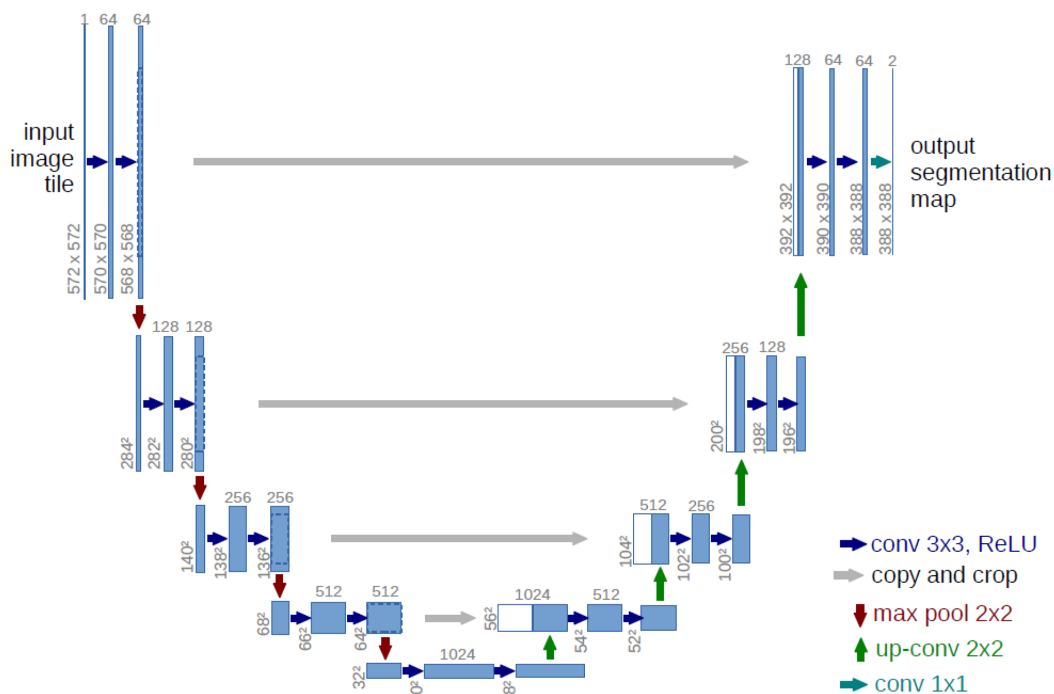


Figura 2.4: Arquitectura de la U-Net original [Ronneberger *et al.*, 2015]

original se añade como *padding* la propia imagen reflejada; véase la figura 2.5. Esta forma de procesar imágenes permite resultados muy precisos a costa de un mayor tiempo de inferencia. Lo más común en otras aplicaciones donde no se necesita una elevada precisión es reducir el tamaño de las imágenes y utilizar padding para que puedan ser procesadas con una sola pasada por la red.

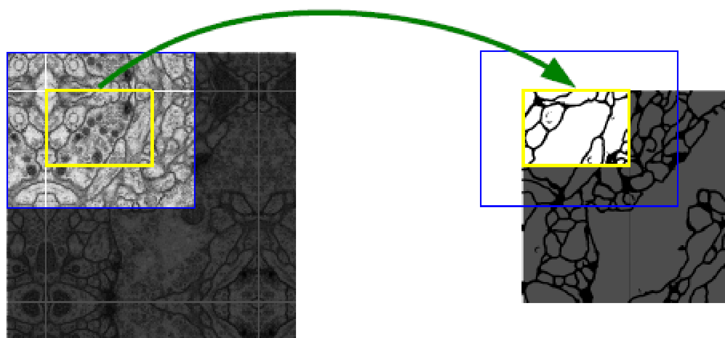


Figura 2.5: Estrategia seguida por la U-Net para procesar imágenes de gran tamaño [Ronneberger *et al.*, 2015]

Los autores consiguieron con esta arquitectura mejorar los resultados conseguidos hasta la fecha de forma significativa en los *datasets* PhC-U373[4] y DIC-HeLa[5] . Ambos *datasets* contienen una cantidad muy reducida de imágenes de células segmentadas (35 y 20 respectivamente), por lo que se utilizaron extensivamente técnicas de *data augmentation* para conseguir un detector robusto. En el caso de segmentación de células, hay invarianzas ante rotación, deformaciones y algunos cambios

[4] Department of Bioengineering University of California at Berkeley. Berkeley CA (USA)
[5] Erasmus Medical Center. Rotterdam. The Netherlands

de color, que se pueden plasmar en los datos de entrenamiento para que la red lo aprenda. [Ronneberger *et al.*, 2015]

2.2.2. ResNet

Res-Net [He *et al.*, 2015a] es una arquitectura presentada por He *et al.* en 2015 que introdujo un nuevo tipo de bloque convolucional con el objetivo de facilitar el entrenamiento de redes muy profundas. Por una parte las redes muy profundas son difíciles de entrenar porque tener una gran cantidad de parámetros facilita la capacidad de aprender funciones excesivamente complejas y a medida de los datos, facilitando el *overfitting*. Por otro lado, al tener muchas capas, es fácil que algunas partes produzcan activaciones de gran magnitud, mientras que otras apenas contribuyen a la salida, lo que afecta sobre todo a arquitecturas con varios caminos entre la entrada y la salida como la U-Net. En Ronneberger *et al.*, 2015 se prevenía este problema inicializando los *kernels* de las capas más profundas con valores más pequeños, que se conseguía extrayendo estos valores de una distribución normal con una varianza inversamente proporcional al número de canales en la capa anterior.

La arquitectura Res-Net se basa en los bloques residuales, que introducen una conexión entre la entrada y la salida de cada bloque convolucional; véase la figura 2.6. Esta conexión permite parametrizar cada bloque alrededor de la función $f(x) = x$, en vez de parametrizar alrededor de $f(x) = 0$, que es lo más común. La función identidad se implementa ahora con *kernels* de convolución con todo ceros, en vez de necesitar aprender la función identidad desde cero. En redes muy profundas esto facilita que la red aprenda a no hacer nada cuando no hacer nada es lo mejor que se puede hacer [He *et al.*, 2015a]. Las redes con bloques residuales han demostrado ser más fáciles de optimizar, menos sensibles a las condiciones iniciales y otros hiperparámetros, y consiguen mejores resultados.

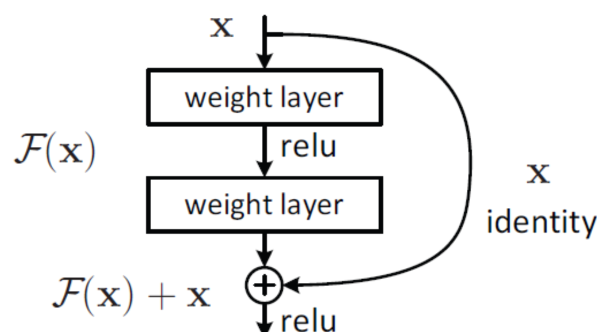


Figura 2.6: Bloque residual [He *et al.*, 2015a]

Capítulo 3

Dataset

Los resultados que se pueden conseguir con un clasificador entrenado mediante aprendizaje supervisado están limitados por la cantidad y calidad de los datos de entrenamiento, y por lo tanto estos son sumamente importantes. Por supuesto, no es lo único que determina la calidad del sistema, ya que este debe ser capaz de extraer suficiente información relevante de los datos.

Una de las principales limitaciones para desarrollar sistemas de visión artificial es que los datos de entrenamiento pueden ser muy difíciles de obtener. Etiquetar muestras de forma automática implica que la calidad del conjunto de datos es la del sistema de etiquetado. Si no se utilizan técnicas de *data augmentation* ni se tratan las muestras, un clasificador entrenado con estos datos aspira a ser tan eficaz como el sistema de etiquetado. Esto, en general, implica que el etiquetado se debe hacer de forma manual, lo que es un proceso laborioso y que puede tener un coste elevado. Dicho esto, ciertos sistemas han conseguido mejorar de forma consistente los resultados obtenidos por humanos, como Ciresan *et al.*, 2012 en el reconocimiento de señales de tráfico [Stallkamp *et al.*, 2011]. Este ejemplo muestra el caso de un problema muy simplificado, donde se busca reconocer señales de tráfico recortadas. Por lo general, en problemas más complejos se está lejos de conseguir resultados[1] comparables a los humanos con métodos artificiales. Por ejemplo, Ronneberger *et al.*, 2015 consiguieron una precisión a nivel de píxel del 93.89 %, mientras que la precisión conseguida por humanos es del 99.9 %.

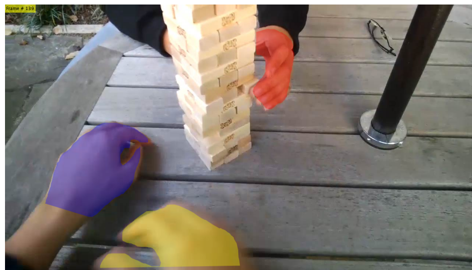
Por suerte, junto con la popularización masiva del aprendizaje automático en los últimos años, varias instituciones y sus investigadores han creado enormes conjuntos de datos accesibles de gran calidad. Estos conjuntos de datos fomentan la investigación en el campo del *Machine Learning* y permiten medir los avances que ha habido durante los años. Dos ejemplos destacables son ImageNet [Deng *et al.*, 2009], creado por investigadores de Princeton y Stanford, y MNIST [LeCun y Cortes, 2010], creado por investigadores de Google, Microsoft y la Universidad de Nueva York a partir del*dataset* NIST, creado por el *National Institute of Standards and Technology* de Estados Unidos.

El conjunto de datos que se ha utilizado en este trabajo es el EgoHands dataset

[1] Hablando de *top-1 performance*.

(*Egocentric Hands*) [Bambach *et al.*, 2015], creado por la Universidad de Indiana. Contiene 4800 imágenes etiquetadas de 48 vídeos de interacciones complejas entre dos personas grabados en primera persona. Las etiquetas son cuatro máscaras de segmentación precisas, clasificadas en cuatro clases posibles, distinguiendo mano derecha e izquierda del observador y de la otra persona. Permite por lo tanto realizar segmentación de instancias, segmentación semántica o detección de objetos, con un simple procesado de las máscaras. Tanto las máscaras de segmentación como las imágenes tienen una resolución de 1280x640 píxeles, por lo que permite entrenar clasificadores muy precisos.

En los 48 vídeos aparecen dos de cuatro personas, en uno de tres entornos, y realizando una de cuatro actividades diferentes; véase la figura 3.1. Las diferentes combinaciones introducen cierta variabilidad en cuanto a los entornos y las condiciones de iluminación en las que el clasificador debe aprender. Esta variabilidad implica que hay ciertos invariantes relevantes en la detección de manos representados en los datos, y por lo tanto una red entrenada con este conjunto de datos podrá aprender a ignorar esta información contextual no relevante.



(a) Partida de jenga en el patio

(b) Haciendo un puzle en el salón

(c) Jugando al ajedrez en el patio

(d) Jugando a las cartas en una oficina

Figura 3.1: Cuatro imágenes de EgoHands con sus máscaras de segmentación. [Bambach *et al.*, 2015]

En este caso, como se busca segmentar semánticamente las imágenes, es necesario combinar todas las máscaras asociadas a cada mano en una única máscara de segmentación; véase 3.2c. A pesar de no hacer segmentación de instancias, se busca obtener una máscara individual para cada mano, representado como un *blob* en la segmentación. Si no se procesa ligeramente el *dataset*, en casos donde aparezcan dos o más manos suficientemente cerca, la red tenderá a detectar ambas con un único *blob*. Esto se debe a que si se compensa de alguna manera el desequilibrio entre

clases (hay más píxeles de fondo que de manos), por lo general se penaliza menos un falso positivo que un falso negativo, y por lo tanto tiene sentido para la red predecir como positivo un píxel en la vecindad de varios otros positivos.

3.1. Procesado

Es necesario procesar ligeramente las máscaras antes de combinarlas para asegurar que hay al menos un píxel de fondo entre las máscaras de cada mano para poder codificar la separación entre manos en los datos.

Aparecen dos problemas al combinar las máscaras directamente, aunque en esencia es el mismo problema. Por una parte hay máscaras que se solapan, es decir que hay casos extremos en los que un único píxel pertenece a dos clases en el *ground truth* original. Por otro lado, hay máscaras que son contiguas, sin llegar a solaparse.

Las máscaras del conjunto de datos utilizado contienen valores entre 0 y 255, por lo que el primer paso será binarizarlas. Esto facilitará trabajar con ellas, y así representarán algo más tangible, si un píxel pertenece a una mano o no. Por lo general el interior de cada *blob* tiene un valor de 255, el fondo tiene un valor de 0, y los bordes tienen valores intermedios. El umbral elegido es 128, ya que que los bordes de las manos son bastante regulares para este umbral. Una vez binarizadas las máscaras, se utilizarán dilataciones ($\oplus$) y operaciones AND ($\wedge$), OR ($\vee$) y NOT ($\neg$) para detectar y eliminar conflictos.

Para poder asegurar una separación de al menos un píxel entre cada *blob*, primero se detectarán las regiones conflictivas hallando la intersección de las máscaras tras haber hecho una dilatación. Para poder asegurar que esta separación se mantenga incluso tras reducir la resolución de las imágenes[2] , se ha utilizado en las dilataciones un *kernel* 5x5, redondeado para suavizar las intersecciones. Este *kernel* se denotará a continuación como k

$$k = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

A continuación se obtienen las versiones dilatadas de las máscaras de segmentación utilizado el *kernel* k. Se denomina como A, B, C y D las cuatro máscaras de segmentación asociadas a una imagen, y A^d, B^d, C^d y D^d sus versiones dilatadas

$$A^d = A \oplus k \tag{3.1}$$

$$B^d = B \oplus k \tag{3.2}$$

$$C^d = C \oplus k \tag{3.3}$$

$$D^d = D \oplus k \tag{3.4}$$

[2] Lo que se hará para reducir los requisitos de memoria del modelo como se explica en el capítulo §4

Se obtendrá a continuación la intersección de cualquier pareja de estas máscaras dilatadas R como:

$$R_0 = A^d \wedge (B^d \vee C^d \vee D^d) \tag{3.5}$$
$$R_1 = B^d \wedge (C^d \vee D^d) \tag{3.6}$$
$$R_2 = C^d \wedge (D^d) \tag{3.7}$$
$$R = R_0 \vee R_1 \vee R_2 \tag{3.8}$$

Y por último, la máscara procesada M será la unión lógica de A, B, C y D, como representan los datos originales, quitando además las zonas conflictivas R

$$M = (A \vee B \vee C \vee D) \wedge \neg R \tag{3.9}$$

Las zonas conflictivas serán por lo tanto parte del fondo, y se utilizarán para das más importancia a esos píxeles durante el entrenamiento. Igual que se hizo en Ronneberger *et al.*, 2015, se utilizará una función de pérdida por píxel, ponderada con mapas de pesos para dar más importancia los píxeles que sirven de separación entre dos *blobs*, con el objetivo de forzar a la red a aprender esta separación; véase la figura 3.2d. Añadir más peso a ciertas regiones del fondo amplifica el desequilibrio entre clases, para minimizar esto se ha reducido además el peso base de los píxeles del fondo a 0.95, respecto al peso de 1.0 de los píxeles de las manos. Este valor de 0.95 se ha obtenido experimentalmente como un valor con el que conseguía un porcentaje de falsos positivos y falsos negativos equilibrado. El peso de los píxeles del fondo es por lo tanto:

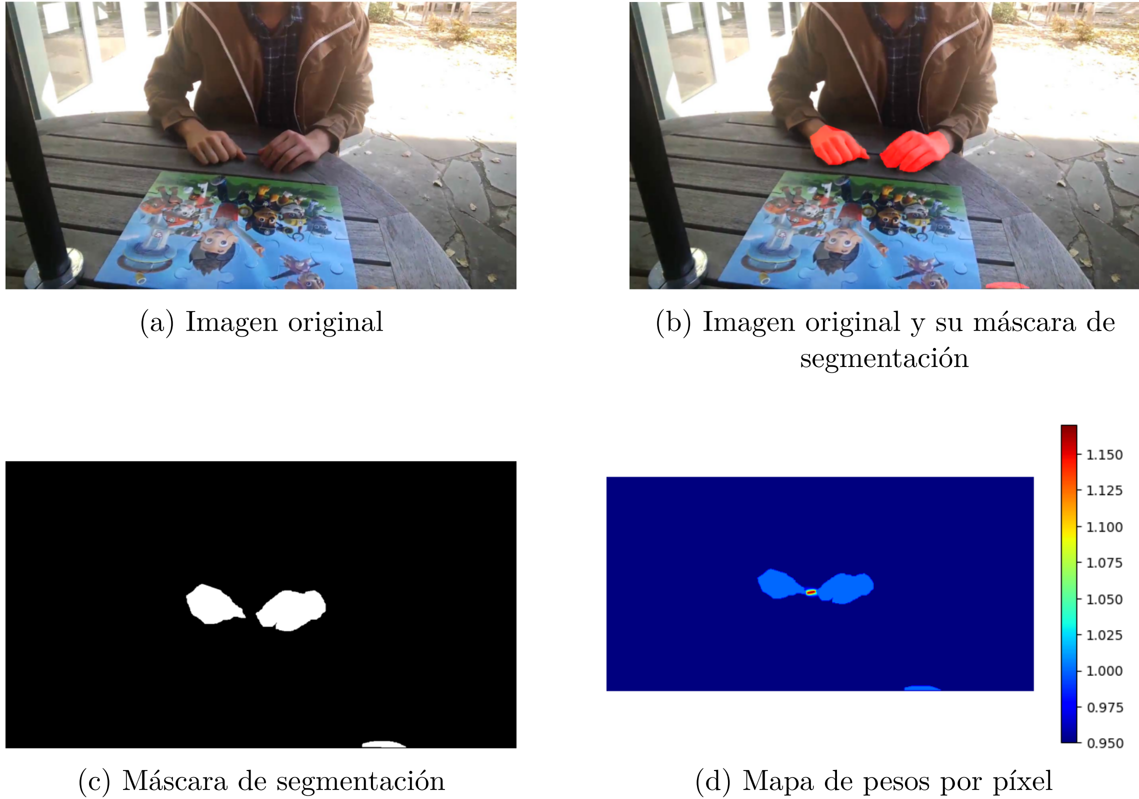
$$w(x) = w_c(x) + w_0 \cdot \exp\left(- \frac{(d_1(x) + d_2(x))^2}{2\sigma^2} \right) \tag{3.10}$$

Siendo w_c el peso base de los píxeles del fondo, w_0 un multiplicador, σ una distancia de referencia en píxeles, y d_1 y d_2 las distancias del píxel en cuestión a las dos manos más cercanas. Los valores utilizados son $w_c = 0.95$, $w_0 = 12$, y $\sigma = 5$. Que la función sea una exponencial decreciente, hace que para que el valor añadido sobre el peso base sea significativo, ambas d_1 y d_2 deben ser suficientemente pequeñas; véase 3.3.

3.2. Data augmentation

Los conjuntos de datos utilizados utilizados en visión artificial contienen, por lo general, una cantidad reducida de imágenes debido a que el proceso por el que se obtienen es poco automatizable. Esto aplica sobre todo al caso de segmentación, donde se requiere de etiquetas muy precisas. Es muy fácil ampliar un *dataset* de segmentación aplicando transformaciones sencillas a las imágenes y sus máscaras, lo que permite explotar más eficientemente los datos.

El tipo de transformaciones que se pueden realizar depende de la aplicación. Voltear horizontalmente las imágenes es la transformación más genérica, ya que en la naturaleza tiene tendencia a la simetría, y porque no suele ser relevante la distribución horizontal de objetos independientes en una imagen. Hay aplicaciones en



(a) Imagen original

(b) Imagen original y su máscara de segmentación

(c) Máscara de segmentación

(d) Mapa de pesos por píxel

Figura 3.2: Imagen de ejemplo con su máscara de segmentación y mapa de pesos asociado.

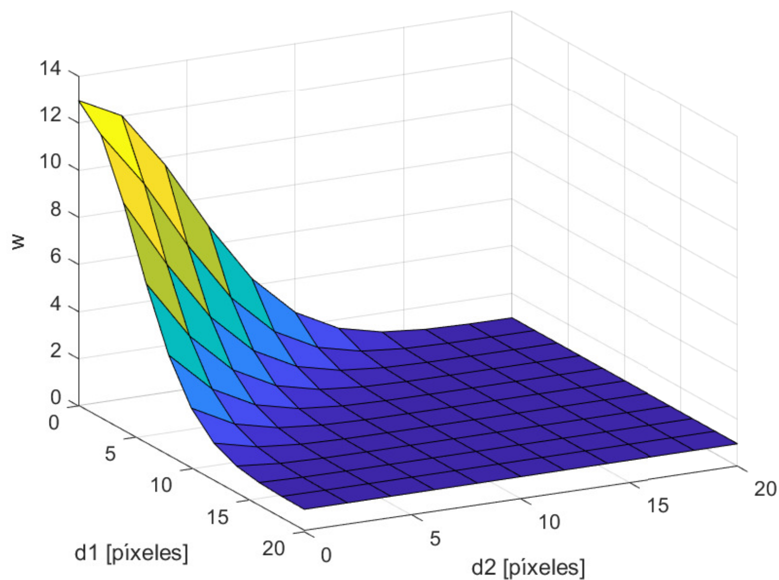


Figura 3.3: Ecuación 3.10 con los valores $w_c = 0.95$, $w_0 = 12$, y $\sigma = 5$. Se puede ver que solo toma valores significativos para valores simultáneamente pequeños de d_1 y d_2.

las que un volteo horizontal no preserva las características de la imagen, como es el caso de texto o dígitos. Este tipo de transformación añade variabilidad a un *dataset*,

y al codificar un invariante en los datos, hará un detector entrenado con estos datos más robusto. Véase la figura 3.4.

(a) Imagen original (b) Imagen modificada

Figura 3.4: Ejemplo de volteo horizontal como método de data augmentation.

Un volteo vertical de las imágenes aporta lo mismo que un volteo horizontal, pero no es aplicable a los mismos casos. Es discutible si un objeto sigue siendo el mismo si está boca abajo. Para la aplicación que considera este trabajo, una mano puede aparecer perfectamente dada la vuelta, pero por el mismo motivo esto ya se representa en los datos. En este caso, esta transformación no preserva la información de la imagen ya que distorsiona excesivamente el contexto del objeto a detectar, y por lo tanto no se considerará. Hay otras aplicaciones, como segmentación de células, donde esta transformación representa un invariante más en los datos.

Random cropping y *zooms* son dos variaciones del mismo concepto, que se basa en que, en ciertas aplicaciones, las imágenes no contienen exclusivamente los objetos a detectar, sino que estos aparecen en el contexto de la imagen. Por ello, una porción de la imagen que contenga objetos de interés y conserve algo de contexto contiene información útil para entrenar un modelo. *Random cropping* consiste en seleccionar zonas de pequeño tamaño en imágenes y aumentar su resolución a la de la imagen original. Una versión menos extrema es hacer un pequeño *zoom* a la imagen, que preserva más contexto y es más adecuado cuanto menos distinguible sea el objeto descontextualizado. En este caso se han recortado trozos de un ancho aleatorio (entre un 0 y un 12 % de la dimensión correspondiente) de todos los lados de la imagen. Véase la figura 3.5. Se puede apreciar en esta figura que debido a que se reescala la imagen a la resolución original aparecen pequeñas distorsiones geométricas y variaciones de escala que pueden mejorar el aprendizaje.

Añadir variaciones en la iluminación es muy importante ya que los detectores pueden ser muy sensibles a cambios de este tipo. Esto hará un detector más robusto a variaciones en la iluminación del entorno, además de introducir un nuevo invariante en los datos, que permitirá al modelo centrarse en los rasgos más importantes. En este caso se han introducido pequeños *offsets*, y variaciones en la saturación. Véanse las figuras 3.6 y 3.7 respectivamente.

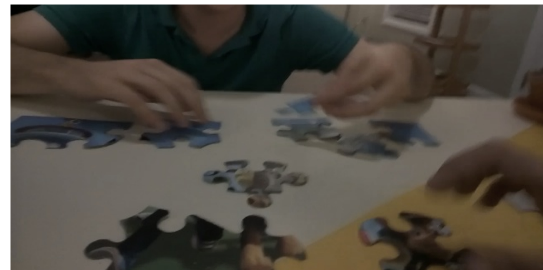
Utilizar pequeñas rotaciones para crear imágenes nuevas es otra técnica ampliamente utilizada ya que no distorsiona la geometría original de las imagen e introduce un nuevo invariante. Como inconveniente, es necesario reducir la resolución de la

(a) Imagen original

(b) Imagen modificada

Figura 3.5: Ejemplo de zooms aleatorios como método de data augmentation.

(a) Imagen original



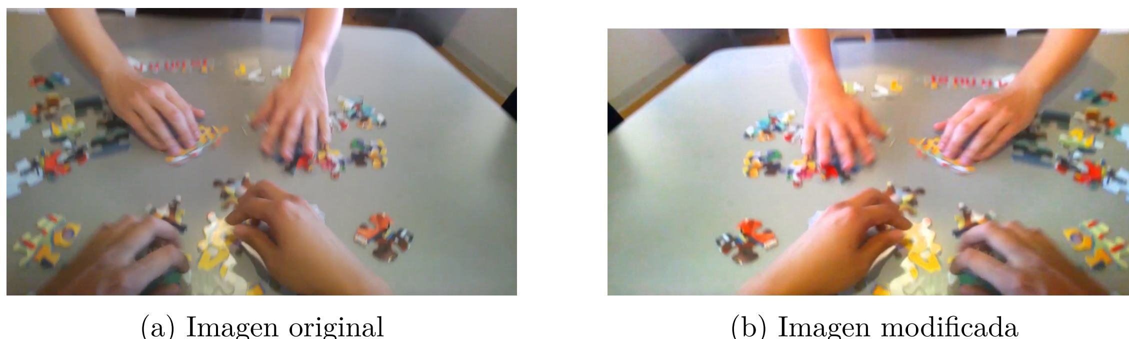
(b) Imagen modificada

Figura 3.6: Ejemplo de añadir un offset a todos los canales de imágenes como método de data augmentation. En este caso la imagen ha sido volteada para introducir más variabilidad.

imagen y reescalar por el efecto que tiene en los bordes. Esto hace que introducir rotaciones significativas puede degradar en gran medida las proporciones de la imagen. En este caso se ha decidido no utilizar transformaciones de este tipo porque, como las imágenes están tomadas en primera persona, ya contienen pequeñas rotaciones. Introducir rotaciones significativas que no estén ya representadas en el conjunto de datos implicaría introducir imágenes relativamente degradadas.

Las imágenes obtenidas por métodos de *data augmentation* codifican información nueva en los *datasets* y añaden más ejemplos sobre los que una red puede aprender. Por otra parte, la información introducida no es nueva, una imagen obtenida de esta forma no es igual que una nueva muestra. Esto hace que abusar del *data augmentation* puede acelerar el sobreajuste, sobre todo si las variaciones introducidas son pequeñas. En este caso se han obtenido 1222 imágenes nuevas a partir de las 3600 imágenes utilizadas para entrenar. De estas, un 65 % están volteadas horizontalmente, un 55 % tienen un zoom aplicado, y un 40 % tienen alguna variación de color.

Introducir modificaciones de este tipo en un *dataset* de segmentación semántica es muy sencillo debido a que es suficiente con aplicar la misma transformación a la imagen y su máscara de segmentación, y en este caso también al mapa de pesos. Algunas modificaciones, como las variaciones en el color son aun más sencillas ya que solo se deben aplicar a la propia imagen si se busca que la red aprenda a ignorar pequeñas variaciones en el color.



(a) Imagen original (b) Imagen modificada

Figura 3.7: Ejemplo de modificar la saturación en imágenes como método de data augmentation. En este caso la imagen también ha sido volteada para introducir más variabilidad.

3.3. Reparto del dataset

El reparto de los datos para el entrenamiento se hará reservando conjuntos de vídeos completos para entrenamiento, validación y *test*. No tener fotos de todos los vídeos en cada conjunto permite que los resultados de la evaluación posterior sean más representativos ya que de otra forma la red habría entrenado con imágenes muy similares. Por otro lado, de esta forma los datos de entrenamiento contienen menos variabilidad, y se puede esperar que el modelo obtenido sea marginalmente peor.

Tener un conjunto de datos para validar y otro para *test* permite también mitigar el sobreajuste del modelo producido por el ajuste fino de hiperparámetros al *dataset* utilizado. Dicho esto, no siempre tiene sentido fragmentar tanto el conjunto de datos, sobre todo en este caso en el que el *dataset* es relativamente pequeño. Las medidas realizadas sobre pocos vídeos tendrán mucho ruido ya que los resultados obtenidos en cada vídeo tienen gran efecto sobre el total. El conjunto de datos de validación tiene un sesgo tras el entrenamiento en cuanto al tamaño de la red, los valores de algunos hiperparámetros, y numero de épocas entrenadas, sobre todo si se utiliza *early stopping*, y por ello es necesario tener un conjunto de datos de *test*.

En el reparto inicial del conjunto de datos se destinaron 36 vídeos a entrenamiento, 7 a validación y 5 a *test*, lo que representa un 75 %, 15 % y 10 % del *dataset* original respectivamente. Pero, en base a los resultados obtenidos, se ha decidido utilizar los 7 vídeos destinados originalmente para validar tanto para validación como para *test*, con objetivo de obtener evaluaciones más significativas de los modelos. Por lo que se utilizarán 7 vídeos para validar, y 12 vídeos para *test*, los 5 destinados originalmente a *test* y los 7 de validación. Esto no es ideal, pero de esta forma se han obtenido resultados mucho más consistentes, y aunque la mayoría de vídeos de *test* presentan cierto sesgo, la cantidad de vídeos sin sesgo es significativa.

Los vídeos utilizados para el entrenamiento, validación y *test* son siempre los mismos. Esto implica que los datos de *test* sí presentan un cierto sesgo, debido a que el rendimiento de las redes se comparan en base a sus resultados sobre estos datos. Esto se intentará mitigar teniendo en cuenta solo las diferencias en el rendimiento que sean significativas o consistentes, y evaluando los modelos con un criterio rigu-

roso. Cuando los resultados obtenidos por distintos modelos sean muy similares, se recurrirá a *k-fold validation* para asegurar que el sesgo introducido es mínimo.

Capítulo 4

Arquitecturas consideradas

A partir de las conclusiones obtenidas en la sección §2.2, se ha decidido explorar dos tipos de redes totalmente convolucionales, la U-Net y la ResNet. En este capítulo se mostrarán los detalles de los modelos considerados y los resultados obtenidos con estos. A partir de los resultados se escogerá una de las dos arquitecturas para en el siguiente capítulo realizar un ajuste fino del modelo evaluando el efecto de varios hiperparámetros.

Como suele pasar en problemas de visión artificial, se dispone de un conjunto de datos limitado, y por lo tanto el riesgo de sufrir problemas de sobreajuste es elevado. Para poder entrenar durante varias épocas y obtener un modelo que generalice bien, se ha decidido utilizar *dropout* en todos los modelos. Se ha tenido en cuenta la cantidad de *dropout* como parte de la arquitectura, y no como un hiperparámetro más[1] , porque en base a las pruebas realizadas se ha visto que influye tanto sobre la calidad del entrenamiento como sobre la capacidad de aprendizaje del modelo. Por otro lado, no hay consenso sobre si se debe utilizar *dropout* en redes convolucionales o no, y por ello se va a prestar especial atención a este hiperparámetro. Comparar las dos arquitecturas variando un hiperparámetro permite además tener una muestra mucho más significativa en base a la que elegir un modelo u otro.

El efecto positivo del *dropout* en capas densas durante el entrenamiento es ampliamente reconocido y está constatado por múltiples estudios. Cabe destacar Srivastava *et al.*, 2014, p. 1948, donde se puede apreciar una clara mejora en la calidad de los rasgos extraídos de las imágenes por las capas convoluciones, por utilizar *dropout* exclusivamente en la cabeza densa de la red. El efecto del *dropout* en capas convolucionales no tiene fundamento matemático[2] , su uso está poco extendido, y no hay consenso sobre si tiene efectos positivos o negativos en el entrenamiento. Analizar el efecto del *dropout* y elegir un *dropout rate* adecuado será uno de los aspectos fundamentales de este capítulo.

[1] Nótese que el *dropout* no es parte de la arquitectura, sino un hiperparámetro que solo influye en el entrenamiento del modelo. Si el *dropout* tiene un efecto positivo sería que reduce el sobreajuste durante el entrenamiento, aumentando su capacidad de generalizar, no que aumenta la capacidad de aprender del modelo.

[2] Como sí tiene el uso de *dropout* en capas densas.

4.1. Detalles de la arquitectura

A continuación se comentarán los detalles de los modelos utilizados y las consideraciones que han influido en su diseño.

Uno de los factores principales que determinan el tamaño de un modelo es la cantidad de memoria disponible en GPU. Si no se recurre a métodos sofisticados, debe haber espacio suficiente para almacenar simultáneamente la red, un *batch*, y los resultados intermedios que sean necesarios. Estos modelos se han entrenado en una GPU con a penas 3GBs de VRAM, lo que impone una fuerte restricción sobre el tamaño de los modelos. Una U-Net debe almacenar gran cantidad de resultados intermedios del codificador porque se necesitan en el decodificador de la arquitectura. Una ResNet requiere de menos memoria porque necesita almacenar solamente la salida del bloque anterior, además del actual. Esto implica que utilizando la misma memoria, una ResNet puede tener más parámetros y realizar más operaciones que una U-Net. Se puede prever por lo tanto mejor rendimiento de la arquitectura ResNet en este caso.

4.1.1. U-Net

Se ha utilizado una U-Net estándar, con tres bloques convolucionales en el codificador y el decodificador, unidos por un cuello de botella y las *skip connections* características de la U-Net. La arquitectura utilizada se muestra en el figura 4.1, y está basada en un modelo encontrado en Kaggle [*U-net, dropout, augmentation, stratification*].

Cada bloque del codificador cuenta con dos capas convolucionales con activación ReLU y una operación de *maxpooling* para reducir la resolución para el bloque siguiente. En el decoder, primero se utiliza una convolución traspuesta con un paso mayor que la unidad como método de *upsampling* para aumentar la resolución de los resultados intermedios, que se concatenan con la salida de las capas convolucionales del bloque del codificador correspondiente. A continuación se utilizan dos capas convolucionales para reducir los canales de profundidad y procesar la información. El codificador y el decodificador se unen por un cuello de botella, donde debería obtenerse la información más compleja, de más alto nivel y menos posicionalmente precisa.

Todas las convoluciones utilizan *kernels* de tamaño 3x3 como es costumbre en redes totalmente convolucionales. Se ha probado con otros tamaños, pero los mejores resultados se han obtenido con estos *kernels* pequeños. La red tiene un total de 2.1 millones de parámetros y una velocidad de inferencia de 51ms/imagen[3] en imágenes con una resolución de 640x320 píxeles.

No se utiliza *batch normalization* como método de regularización, solo *dropout* para reducir el sobreajuste. Utilizar *batch normalization* ayuda a reducir el sobreajuste que puede ocurrir si un subconjunto de la red produce activaciones mucho mayores

[3] En una GPU gtx 1050

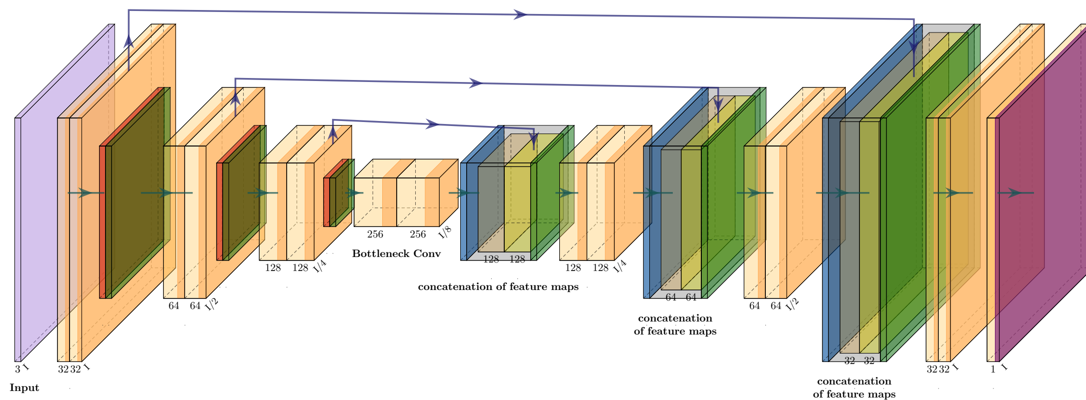


Figura 4.1: Arquitectura U-Net utilizada.
Leyenda: Lila - Input; Naranja claro - Convoluciones; Naranja oscuro - Activación ReLU; Rojo - MaxPooling; Verde - Dropout; Azul oscuro - UpSampling; Gris - Concatenación; Amarillo - Bloque concatenado; Morado - Activación Sigmoide

que el resto, y por lo tanto tiene mucho mayor peso en las predicciones. Utilizar *batch normalization* controla la magnitud de los parámetros de la red, reduciendo el *overfitting* y ayudando a la convergencia del modelo. Se ha decidido utilizar *batch normalization* en la ResNet y no en la U-Net para no comparar modelos demasiado similares.

4.1.2. ResNet

También es completamente estándar la ResNet utilizada, solamente se puede resaltar el uso de *dropout*, que no es convencional en este tipo de arquitectura. La arquitectura utilizada se basa en un modelo encontrado en la documentación de Keras [*Keras documentation*], y se muestra en la figura 4.2.

En este caso la primera capa es una *strided convolution*[4] , una convolución con un paso mayor que 1, que permite disminuir la resolución de la imagen de entrada a la vez que añade canales de profundidad y comienza el procesado. El *encoder* contiene tres bloques de capas convolucionales, *maxpooling* y *dropout* al igual que antes. Además se añaden operaciones de suma entre la entrada y la salida de cada bloque residual; véase la figura 2.6. Las conexiones residuales del encoder se realizan con una capa convolucional con *strided convolutions*, que ajusta el tamaño de uno de los tensores a sumar. El *decoder* tiene tres bloques residuales simétricos al *encoder*, con convoluciones transpuestas con paso 2 como método de *upsampling* igual que en la arquitectura anterior. En este caso las conexiones residuales se hacen con una capa de *upsampling* y una capa convolucional. El cuello de botella es un bloque residual convolucional más, análogo al encontrado en la U-Net.

Se han utilizado activaciones ReLU, y convoluciones con *kernels* 3x3 principalmente, salvo en las conexiones residuales del encoder que se utilizan *kernels* 1x1,

[4] Convolución con un paso mayor que 1.

utilizados convencionalmente para ajustar la resolución[5] y profundidad de los tensores de rasgos inferidos. La red tiene un total de 3.1 millones de parámetros y una velocidad de inferencia de 48ms/imagen[6] en imágenes con una resolución de 640x320 píxeles.

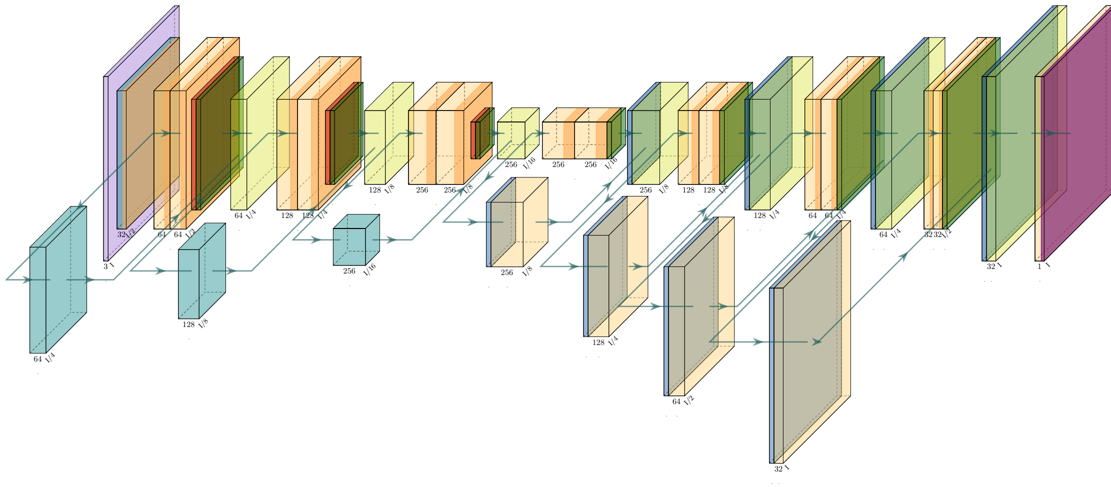


Figura 4.2: Arquitectura ResNet utilizada.
Leyenda: Lila - Input; Azul claro - Strided Convolution; Naranja claro - Convoluciones; Naranja oscuro - BatchNorm y Activación ReLU; Rojo - MaxPooling; Verde - Dropout; Amarillo - Suma; Azul oscuro - UpSampling; Morado - Activación Sigmoide.

En esta arquitectura sí se utiliza *batch normalization* como método de regularización. Se ha podido observar que esto facilita enormemente la convergencia de la red y permite resultados altamente consistentes, incluso con una cantidad elevada de *dropout*. Utilizar *batch normalization* ayuda a reducir el *overfitting* al igual que el *dropout*, aunque de forma diferente. Puesto que es discutible si el *dropout* en redes convolucionales permite a la red aprender mejor, es común utilizar exclusivamente *batch normalization* y no *dropout*. En este capítulo se harán pruebas con diferentes cantidades de *dropout* para observar su efecto en esta aplicación en concreto.

4.2. Primeros resultados

Se han utilizado las 4800 imágenes de los 48 vídeos del *dataset* original, de los cuales se han dedicado 36 a entrenamiento, 7 a validación y test, y 5 a *test* exclusivamente, como se ha explicado el reparto original[7] en el apartado §3.3.

[5] Con paso mayor o igual que 2. A pesar de que esto implica perder información, permite reducir la dimensionalidad de los datos antes de realizar operaciones computacionalmente costosas sobre ellos. [Szegedy *et al.*, 2016]

[6] En una GPU gtx 1050

[7] En posteriores entrenamientos se hará un reparto ligeramente diferente para reducir el ruido en las medidas de la evaluación del modelo.

Como función de *loss* se ha utilizado entropía binaria cruzada (BCE, por sus siglas en inglés) ponderada con un peso por píxel en función de los mapas generados a partir del *dataset*; véase la figura 3.2d. Con esta combinación se han obtenido segmentaciones con buena definición y, salvo en los casos más exigentes, máscaras de segmentación individuales para cada mano. Como optimizador se ha utilizado Adam [Kingma y Ba, 2017], porque al ser un optimizador adaptativo y estimar momentos, es muy robusto y funciona bien en la mayoría de los casos sin ningún ajuste. Se ha entrenado durante 30 épocas en todos los casos porque se ha visto como una cantidad adecuada en las pruebas realizadas. Se ha visto que 30 épocas era suficiente para que convergiese el IoU de validación en todos los modelos entrenados, sin ser demasiado como para que después divergiese por problemas graves de *overfitting*.

Como métrica principal para comparar modelos se va a utilizar *Intersection over Union* (IoU), una de las más representativas a la hora de evaluar segmentaciones. Pero no es la única métrica a tener en cuenta, es importante también que aparezca un *blob* de segmentación por cada mano en la imagen. Que los bordes de las máscaras estén bien definidos también es algo muy a tener en cuenta ya que da una idea de la confianza con la que la red realiza sus predicciones. En concreto se utilizarán como métricas *mean IoU* y *mean pixel accuracy* (*mIoU* y *mPixel accuracy* respectivamente).

Por otro lado, la precisión a nivel de píxel es una métrica muy poco representativa de la calidad de una segmentación. En un dataset como este, donde el *foreground* representa del 8 % de los píxeles, los píxeles del *background* son fáciles de clasificar y poco informativos, y por lo tanto no son una buena referencia para juzgar una segmentación. Utilizar una métrica más exigente como IoU permite ignorar estos píxeles sencillos. A pesar de esto, se ha decidido incluir esta métrica en esta comparativa porque es la más intuitiva.

4.2.1. Elección de la arquitectura

Se han entrenado ambos modelos con los mismos datos e hiperparámetros para poder comparar los resultados de forma justa, a pesar de que esto introduzca un *bias* en los datos de validación y *test*. En estos entrenamientos no se han incluido las imágenes obtenidas con data augmentation porque no se buscan resultados precisos todavía, y utilizar estas imágenes solo ralentizaría los entrenamientos.

Se ha decidido entrenar ambos modelos con *dropout rates* entre 0.0 y 0.7, a intervalos de 0.1. Los resultados se muestran en la tabla 4.1. Como ya se ha mencionado, se ha entrenado durante 30 épocas, con BCE ponderada por píxeles y Adam como optimizador.

La U-Net ha demostrado problemas de convergencia con *dropout rates* a partir de 0.3 inclusive, y por ello no se muestran los resultados a partir de 0.4; véase la figura 4.3.

En la figura 4.4 se muestra la evolución del IoU de entrenamiento y validación de cuatro de los modelos considerados en la tabla 4.1. Es fácil apreciar que la arqui-

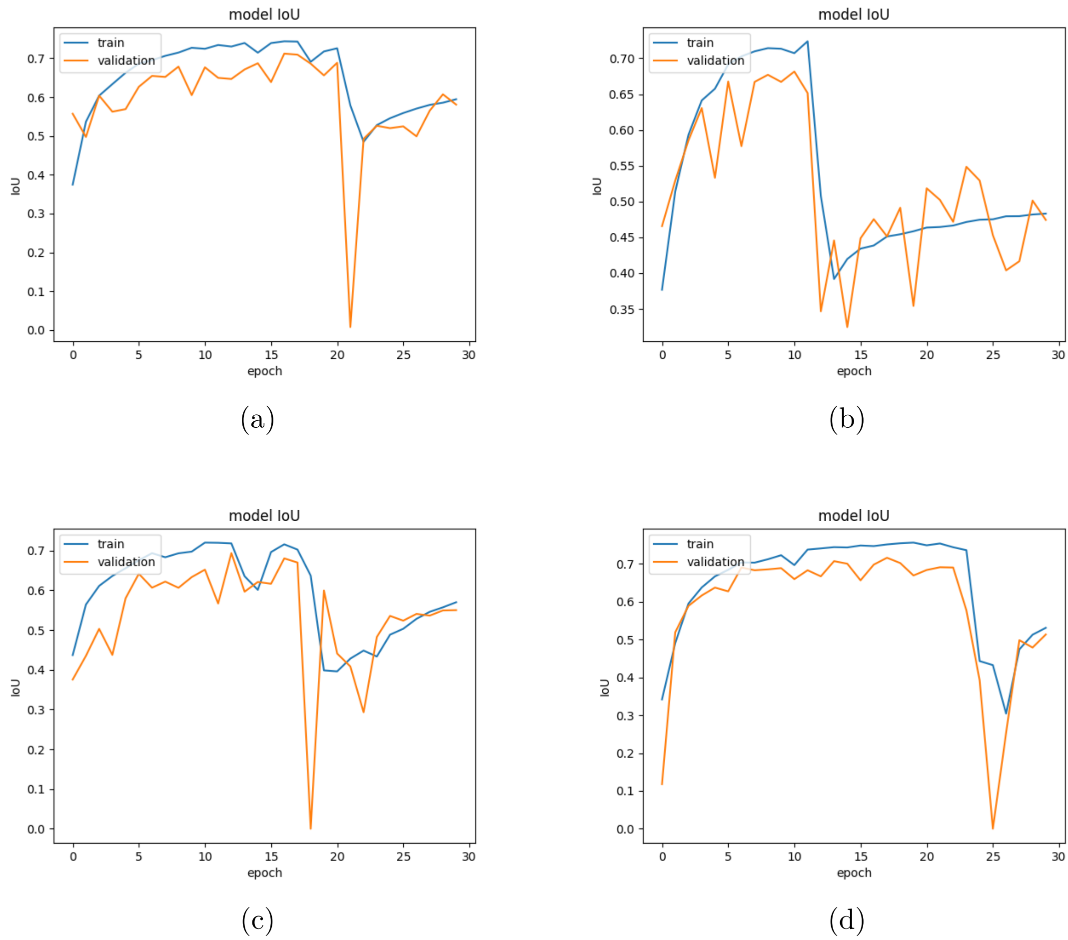


Figura 4.3: Evolución del IoU en entrenamientos de la U-Net con problemas de convergencia. En todos casos se utilizaba un dropout rate de 0.4. En cada uno de los modelos se utilizó un método diferente para inicializar los kernels de convolución buscando la estabilidad del modelo, incluido el método sugerido en Ronneberger *et al.*, 2015, p. 5.

Tabla 4.1: Rendimiento obtenido con las arquitecturas U-Net y ResNet elegidas en función de la cantidad de *dropout* utilizado.

Dropout rate	ResNet		U-Net	
	mIOU	mPixel accuracy	mIOU	mPixel accuracy
0.0	0.791	0.982	0.738	0.976
0.1	0.760	0.979	0.768	0.979
0.2	**0.804**	**0.983**	0.706	0.973
0.3	0.799	0.982	0.742	0.977
0.4	0.755	0.978	—	—
0.5	0.769	0.980	—	—
0.6	0.756	0.979	—	—
0.7	0.721	0.976	—	—

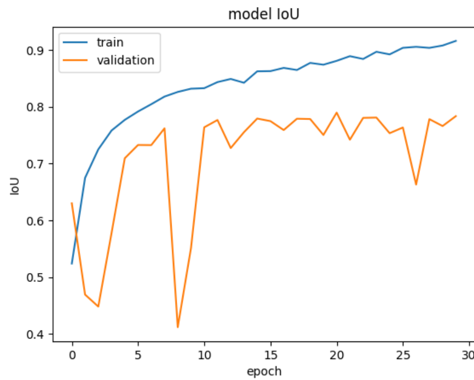
tectura ResNet propuesta tiene capacidad de obtener mejores resultados, y por lo tanto mayor potencial que la U-Net propuesta. Esto se puede explicar por el hecho de que la ResNet tiene aproximadamente un 50 % más de parámetros que la U-Net, a pesar de requerir la misma memoria en GPU y ser un modelo más rápido[8] . Un modelo con más parámetros será capaz de implementar funciones más complejas que pueden ser necesarias para representar la transformación codificada en el *dataset*, a costa de ser más propenso al *overfitting*.

Se puede apreciar una cantidad muy significativa de ruido en la evolución del IoU en casi todos los modelos. Esto significa que una parte significativa de los píxeles de cada imagen se clasifica de forma aleatoria. Esto es debido a que la red está aprendiendo ruido de los datos en vez de las relaciones más profundas representadas en ellos. Esto es uno de los puntos que mejora el uso de *data augmentation* como se verá más adelante. Tener tanto ruido hace difícil además asesorar el modelo en base a las métricas en las muestras de validación e impide utilizar métodos como *early stopping*.

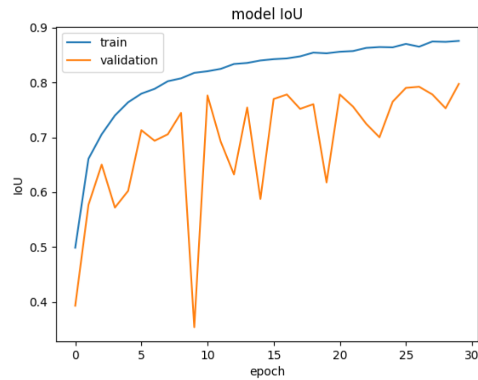
La figura 4.4 puede sugerir que el uso de *dropout* reduce la consistencia de los resultados obtenidos durante la validación. Y es cierto que el uso de *dropout* dificulta la convergencia del modelo por ser una fuente de ruido en cada paso de *gradient descent*. Aun así esto no se ha podido apreciar de forma consistente y no es una de las conclusiones obtenidas en en análisis realizado. Se ha utilizado un optimizador muy resistente al ruido, y es difícil estimar cuanto del ruido presente en la curvas de validación se puede adjudicar al uso de *dropout*.

Para proporcionar una idea sobre la calidad de las segmentaciones obtenidas con los modelos, se presenta en la figura 4.5 la segmentación obtenida con ellos para las mismas dos imágenes de entrada. Para esta comparación se han utilizado los modelos sin *dropout* porque obtuvieron resultados comparables dentro de su categoría para la misma cantidad de *dropout*.

[8] Nótese que esto podría ser muy diferente si se dispusiese de más memoria en la GPU utilizada, ya que es un factor clave en el rendimiento.

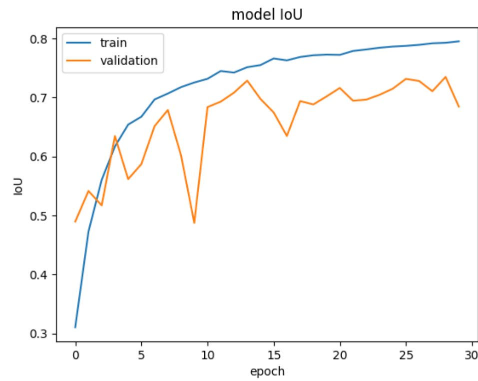


(a) ResNet entrenada con un dropout rate de 0.0



(b) ResNet entrenada con un dropout rate de 0.2

(c) U-Net entrenada con un dropout rate de 0.0



(d) U-Net entrenada con un dropout rate de 0.2

Figura 4.4: Evolución del IoU en dos entrenamientos de la ResNet y dos entrenamientos de la U-Net con dos dropout rates diferentes.

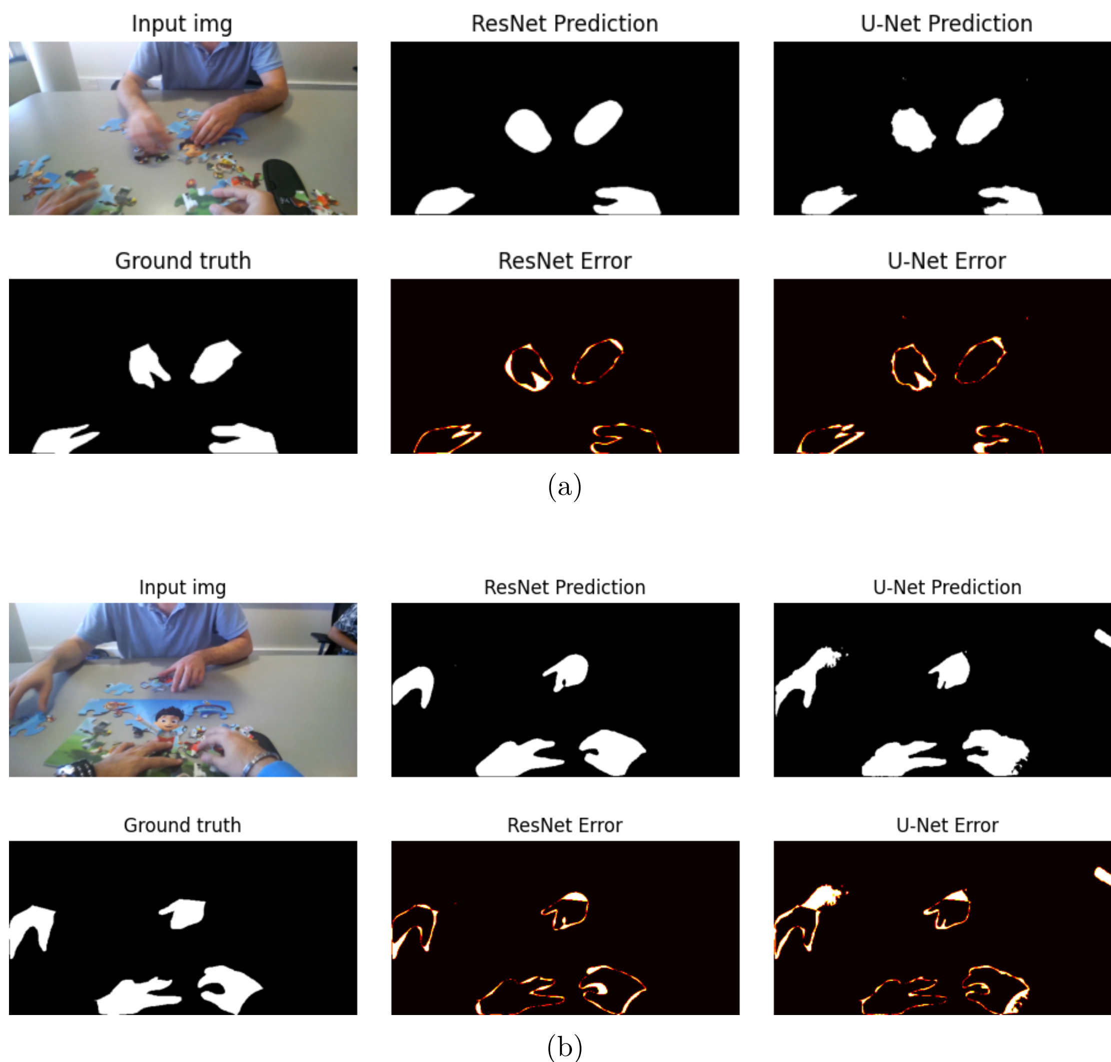


Figura 4.5: Segmentaciones de dos imágenes obtenidas con una ResNet y una U-Net entrenadas sin dropout.

4.2.2. Análisis del efecto del dropout

En este apartado se comentarán los resultados obtenidos a partir de entrenar la ResNet con diferentes cantidades de *dropout* y se elegirá un modelo concreto para hacer un ajuste de otros hiperparámetros en el capítulo §5.

Se han obtenido resultados claramente degradados con *dropout rates* a partir de 0.5, y resultados poco consistentes a partir de 0.4. Se ha podido apreciar por lo tanto un efecto perjudicial claro por utilizar cantidades elevadas de *dropout* durante el entrenamiento, que es apreciable en los resultados de la tabla 4.1.

En la figura 4.6 se muestra la evolución del IoU de entrenamiento durante el entrenamiento de los ocho modelos ResNet comparados en la tabla 4.1. Se puede ver claramente que existe una relación inversamente proporcional entre la cantidad de *dropout* utilizada y los resultados obtenidos en los datos de entrenamiento. Esto indica que el *dropout* reduce en gran medida la capacidad de sobreajuste del mo-

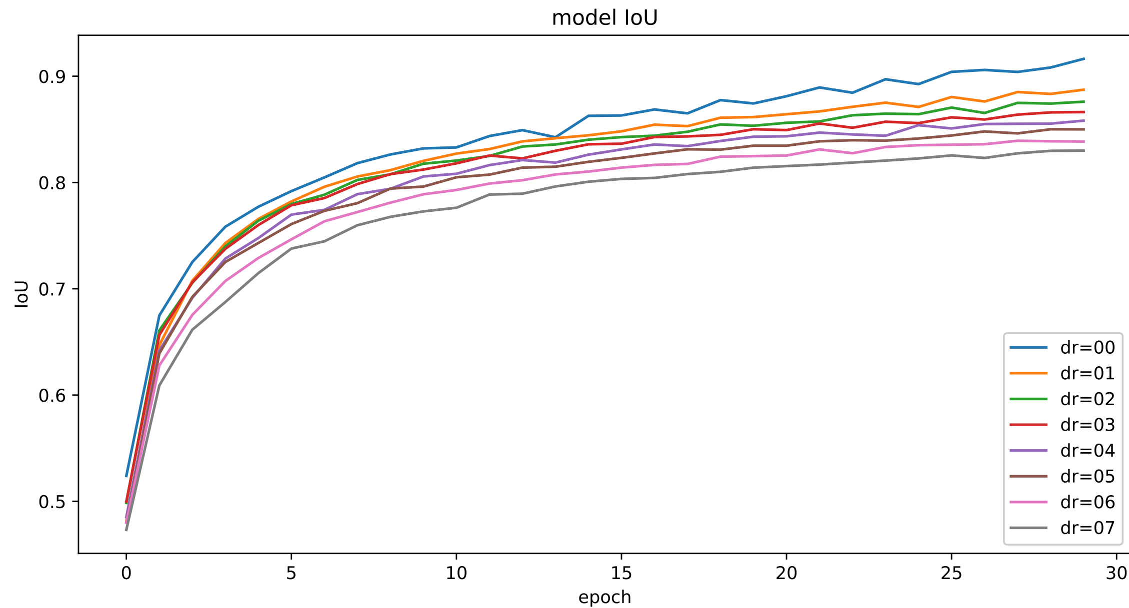


Figura 4.6: IoU de entrenamiento en la misma arquitectura ResNet en función de la cantidad de dropout utilizada durante el entrenamiento.

delo, aunque como se puede apreciar en la tabla 4.1, esto en parte sucede porque también se reduce la capacidad de aprender del modelo. El objetivo es encontrar una cantidad adecuada de *dropout* que limite el *overfitting* y a la vez permita a la red aprender lo suficiente como para poder producir buenos resultados.

En la figura 4.7 se muestra la evolución del IoU de entrenamiento y validación durante el entrenamiento de los modelos ResNet con *dropout rates* de 0.0 y 0.1 considerados en tabla 4.1. No se puede apreciar diferencia entre el IoU de validación de ambos modelos, pero si se puede apreciar una reducción notable en el IoU de entrenamiento. Que la diferencia entre la curva de entrenamiento y la de validación sea menor implica que el modelo con *dropout* ha sufrido menos sobreajuste, y por lo tanto, es un modelo mejor si produce los mismos resultados en los datos de *test*. Esto se debe a que un modelo tendrá más varianza a la hora de inferir en datos nuevos cuanto más se haya sobreajustado a los datos de entrenamiento, y por ello será un modelo menos fiable. Por lo tanto, si los resultados mostrados en 4.7 son consistentes, y no se puede apreciar una diferencia clara en el rendimiento de ambos modelos, se preferirá elegir el modelo entrenado con mayor cantidad de *dropout*.

Se ha reentrenado la arquitectura ResNet considerada con *dropout rates* de entre 0.0 y 0.4, y se han añadido más métricas para poder comparar los modelos con más información. En estos entrenamientos se han utilizado además las imágenes obtenidas mediante *data augmentation* comentadas en la sección §3.2.

Dos de las métricas añadidas son la *Precisión* (*Precision*) y el *Recall*. La precisión indica el porcentaje de píxeles de *foreground* predichos por la red que están bien clasificados[9] . Esto permite distinguir dos modelos con con el mismo IoU en función

[9] En otras palabras, la precisión es el ratio entre los positivos verdaderos y todos los positivos.

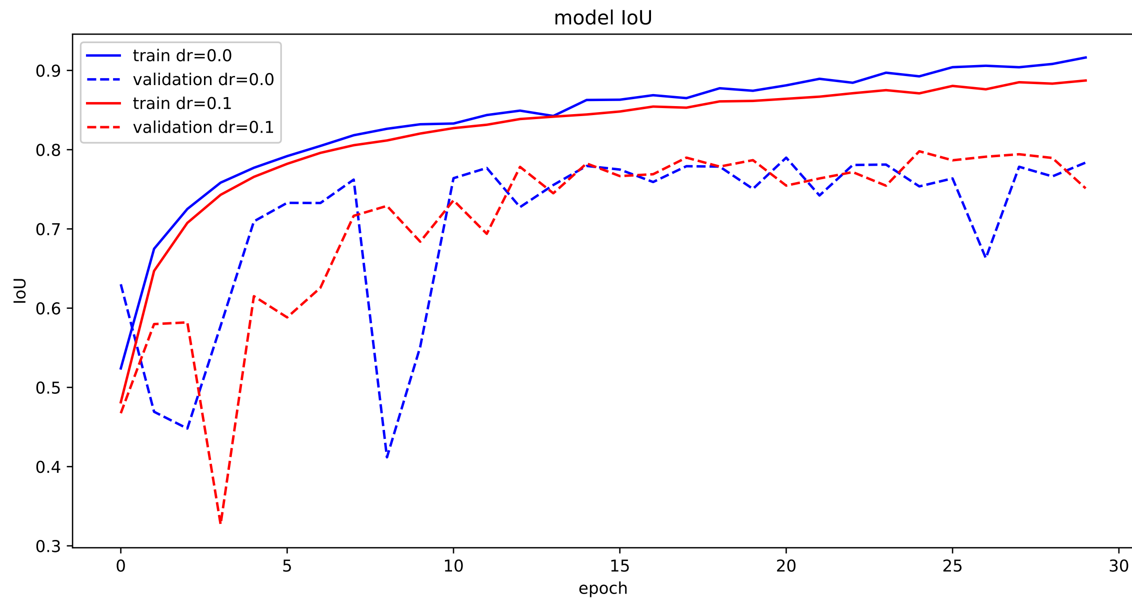


Figura 4.7: Evolución del IoU de entrenamiento y validación durante el entrenamiento de la arquitectura ResNet propuesta entrenada con dropout rates de 0.0 y 0.1.

de cual es más conservador en sus predicciones. El *Recall* indica el porcentaje de píxeles de *foreground* del *ground truth* bien predichos por la red, y da una idea de la capacidad del modelo para identificar correctamente la información más relevante[10]. Aunque ambas métricas se buscan maximizar, se ha decidido que la *Precisión* es más importante que el *Recall* en esta aplicación porque es costoso tratar con *blobs* de falsos positivos y los falsos negativos no presentan ningún riesgo. Tener ambas métricas aporta información sobre las estrategias tomadas por los diferentes modelos para segmentar y permitirá encontrar un equilibrio adecuado entre ellas. Nótese que es muy difícil optimizar ambas métricas simultáneamente, ya que esto implicaría mejoras notables sobre el IoU, que es una métrica muy exigente.

La otra métrica introducida se ha diseñado para poder estimar de forma imparcial una de las cualidades que se busca en el clasificador diseñado, la consistencia con la que se obtiene una máscara individual para cada instancia representada en el *ground truth*. Este requisito deriva de que se pretende hacer segmentación de instancias utilizando segmentación semántica. El valor óptimo de esta métrica es 1.0, los valores obtenidos más comunes varían entre 0.95 y 0.3, y pueden llegar a ser negativos en casos de segmentaciones especialmente fragmentadas. Se hará referencia a esta métrica como la *Métrica de la similitud media entre centroides (SMC)*; véase el apéndice B para más información.

Se ha entrenado el modelo tres veces con cada *dropout rate*, y promediado los resultados obtenidos para reducir la cantidad de ruido presente en las medidas y fiabilizar los resultados. Esto es necesario porque las diferencias en los resultados prometen ser pequeñas porque el *dropout* es solo uno de muchos hiperparámetros. Los resultados se muestran en la tabla 4.2.

[10] El *Recall* es especialmente importante en aplicaciones médicas, donde un falso negativo puede tener consecuencias graves.

Tabla 4.2: Resultados obtenidos con la arquitectura ResNet elegida en función de la cantidad de dropout utilizado. Se muestra la media de los resultados obtenidos tras tres entrenamientos independientes para cada modelo.

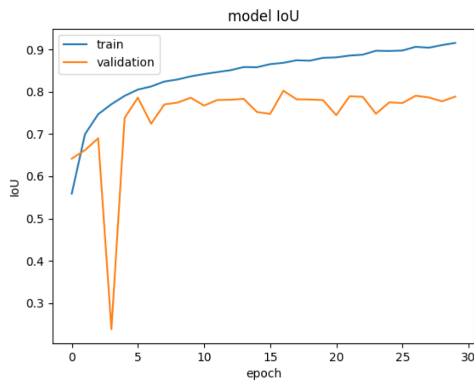
Dropout rate	mIOU	mPrecision	mRecall	mSMC
0.0	0.7928	0.9091	0.8631	0.8163
0.1	0.7936	**0.9174**	0.8565	**0.8226**
0.2	0.7937	0.9054	**0.8671**	0.8172
0.3	**0.7968**	0.9100	0.8662	0.8207
0.4	0.7836	0.9095	0.8535	0.8198

Los resultados obtenidos muestran poca diferencia entre los modelos, aunque se han obtenido resultados mejores de IoU utilizando un *dropout rate* de 0.3 de forma consistente. Puesto que este modelo ha obtenido buenos resultados en todas las métricas se considerará a partir de ahora 0.3 como la cantidad adecuada de *dropout* para esta aplicación, y se partirá de este modelo para hacer un ajuste fino en el capítulo §5.

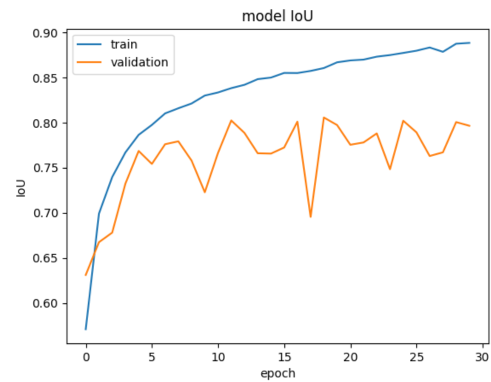
Los valores de *Precision* y *Recall* mostrados en la tabla 4.2 demuestran que el modelo obtenido tiende a hacer predicciones conservadoras. Esto es deseable como se ha explicado antes, pero reducir la diferencia será uno de los puntos de interés a la hora de realizar un ajuste fino del modelo.

La figura 4.8 muestra las curvas de la evolución del IoU de cuatro de los modelos considerados en la tabla 4.2. Se puede apreciar que presentan menos ruido que sus homólogas en la figura 4.4, lo que se puede deber al uso de *data augmentation*. A pesar de esto, no se puede apreciar una mejora significativa sobre el IoU obtenido con estos modelos.

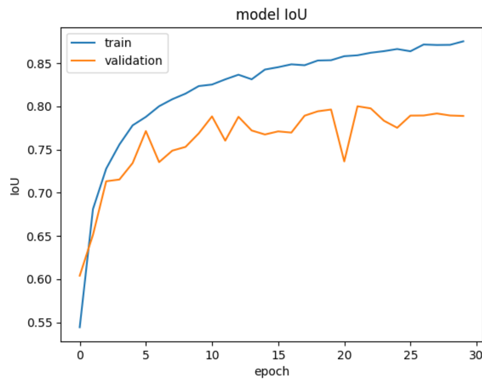
En la figura 4.9 se muestran, a modo de ejemplo, dos segmentaciones obtenidas con uno de los tres modelos ResNet entrenados con un *dropout rate* de 0.3 con los que se han obtenido buenos resultados.



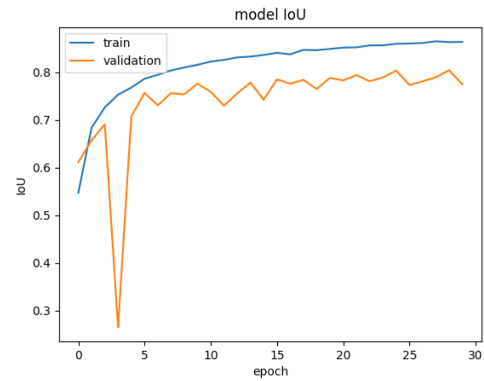
(a) ResNet entrenada con un dropout rate de 0.0



(b) ResNet entrenada con un dropout rate de 0.1

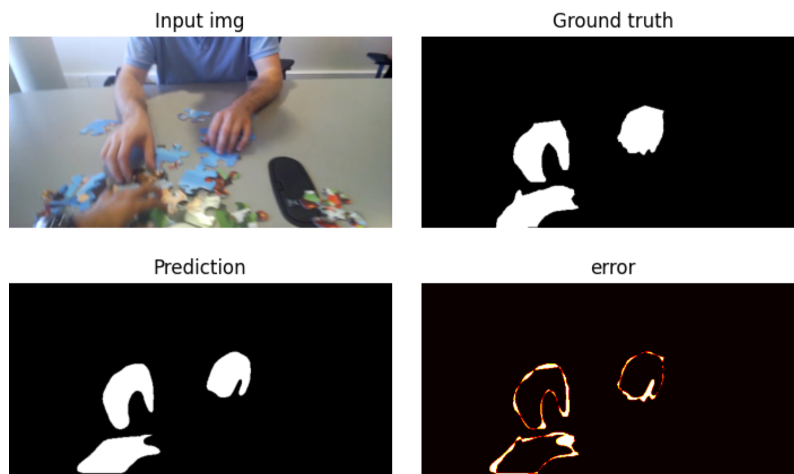


(c) ResNet entrenada con un dropout rate de 0.2

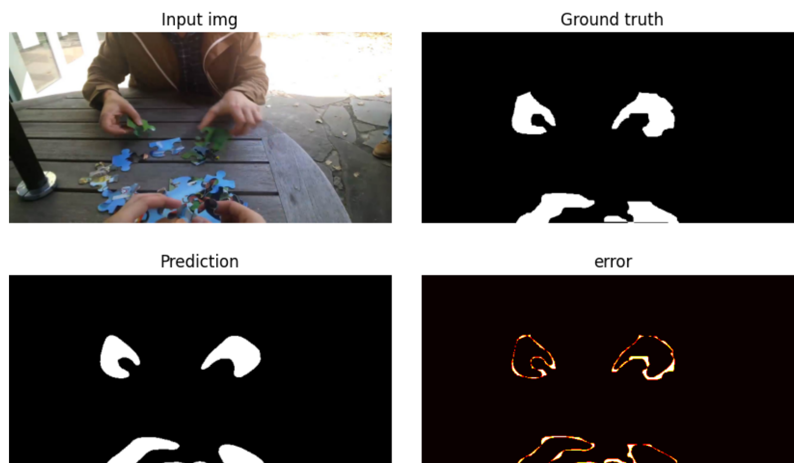


(d) ResNet entrenada con un dropout rate de 0.3

Figura 4.8: Evolución del IoU de entrenamiento y validación en entrenamientos de la arquitectura ResNet propuesta con cuatro dropout rates diferentes.



(a)



(b)

Figura 4.9: Ejemplo de segmentaciones obtenidas con uno de los modelos ResNet entrenados con un dropout rate de 0.3.

Capítulo 5

Ajuste de hiperparámetros

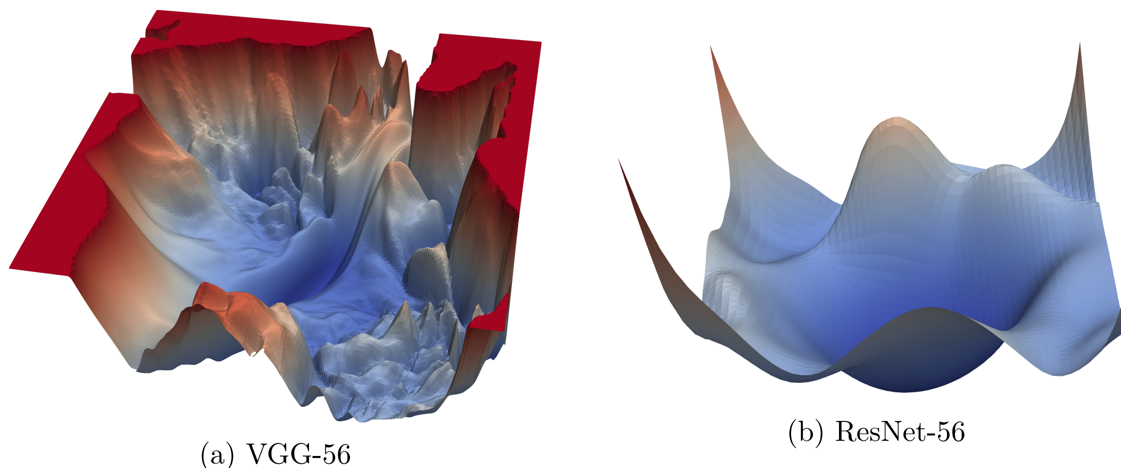
Una de las principales limitaciones del modelo obtenido en el capítulo §4 es la limitada cantidad de parámetros que contiene debido a las restricciones que impone la cantidad de memoria disponible en GPU. Un modelo más grande tendría capacidad de aprender relaciones más complejas, que pueden ser necesarias para representar de forma precisa las transformaciones que hay codificadas en el conjunto de datos utilizado. Es por ello esencial obtener el máximo rendimiento de los parámetros disponibles. Esta optimización del modelo se realizará ajustando ciertos hiperparámetros que controlan la evolución del modelo durante el entrenamiento.

El rendimiento esperado de un modelo está determinado principalmente por el espacio de búsqueda que tiene asociado. Este espacio de búsqueda está definido por la arquitectura, que define el tamaño y la topografía, y la función de pérdida, que define su relieve más detallado. Estos dos son los aspectos principales a tener en cuenta a la hora de diseñar un modelo. El resto de hiperparámetros determinan cómo de eficientemente se busca en este espacio de soluciones.

Li *et al.* [Li *et al.*, 2018] han obtenido representaciones de muy alta resolución de las superficies que definen el espacio de búsqueda de varias arquitecturas en su estudio sobre la topografía del *loss*. Explican que las conexiones residuales producen una convexificación del espacio de búsqueda, lo que facilita su entrenamiento de forma considerable. En la figura 5.1 se muestra la superficie que define el *loss*, y por lo tanto el espacio de búsqueda, de dos modelos con la misma arquitectura básica pero uno de ellos utiliza conexiones residuales y el otro no. Esto aporta una nueva explicación para la diferencia entre los resultados obtenidos con las arquitecturas U-Net y ResNet en el capítulo §4, y en concreto para los problemas de convergencia de la U-Net.

Se van a hacer pruebas principalmente con diferentes funciones de pérdida, en concreto se busca alinear la función de pérdida con la función objetivo para poder acceder a modelos mejores. Alinear la función de pérdida con la función objetivo es el motivo por el que se preprocesó el *dataset* en la sección §3.1.

También se harán pruebas con una función de activación parametrizada, que tiene el potencial de permitir un comportamiento más complejo al modelo introduciendo apenas nuevos parámetros.



(a) VGG-56

(b) ResNet-56

Figura 5.1: Topología del loss en dos arquitecturas que difieren únicamente en el uso de conexiones residuales. [Li *et al.*, 2018].

Por último se harán pruebas con diferentes ratios de aprendizaje (*learning rates*) y su evolución a lo largo del entrenamiento, lo que permitirá optimizar la búsqueda por el espacio de soluciones del modelo.

5.1. Entrenamiento

Los entrenamientos realizados en este capítulo se rigen por los mismos principios en los que se basan los entrenamientos realizados en la sección §2.2. Se utilizarán los mismos 36 vídeos para entrenar, los mismos 7 vídeos para validación y *test*, y los mismos 5 vídeos para *test* exclusivamente. Se utilizarán en todos los entrenamientos además las 1222 imágenes obtenidas con diferentes técnicas de *data augmentation* a partir de los 36 vídeos del conjunto de datos de entrenamiento.

Utilizar siempre los mismos vídeos introduce un sesgo significativo en el *dataset*, ya que se buscará optimizar los parámetros para un subconjunto reducido de los datos. Esto significa que los hiperparámetros obtenidos habrán sufrido de cierto *overfitting*. Por otro lado, esto asegura que el problema que deben resolver todos los modelos es el mismo. El subconjunto de vídeos reservados para validación y *test* se ha elegido de forma aleatoria, y se ha verificado que presenta una diversidad sustancial. Se puede suponer por ello, que la información codificada en el conjunto de datos de *test* es suficiente y suficientemente variada como para que el *overfitting* del modelo debido al ajuste de hiperparámetros sea mínimo.

En un principio se consideró reservar un conjunto de vídeos muy similares para validación y *test*, por ejemplo todos los vídeos asociados al mismo entorno. Esto permitiría estimar el rendimiento del modelo en condiciones realistas que el modelo nunca haya visto antes, y es potencialmente una forma más adecuada de repartir el *dataset*. Se decidió no hacer esto y repartir aleatoriamente los vídeos porque el *overfitting* que puede sufrir el modelo es potencialmente mucho mayor, ya que estaría presente el mismo ruido en gran parte del *dataset*. Este enfoque podría resultar en resultados menos consistentes y por lo tanto difíciles de evaluar, aunque no es algo que se haya comprobado empíricamente.

Respecto a los hiperparámetros que determinan el entrenamiento, se ha decidido seguir utilizando 30 épocas porque se ha visto que es una cantidad adecuada. De todas formas, se almacena una copia de cada modelo cada cinco épocas, lo cual permite escoger entre diferentes estados del modelo en caso de que el entrenamiento diverja. Si es necesario, cualquiera de estos modelos puede reentrenarse para alcanzar el número óptimo de épocas. Por lo general esto no ha sido necesario, y se ha podido ver que no hay un óptimo evidente de épocas para cada modelo, sino que hay un intervalo amplio en la que se estabiliza el rendimiento del modelo y no aporta nada continuar entrenando. Dicho esto, se ha podido observar que con valores más altos de *learning rate* el modelo converge más rápido, y por lo tanto idealmente debería entrenar durante menos tiempo.

Se ha utilizado un *batch size* de 2 por restricciones de memoria. Esto hace que la red se actualice muchas veces durante el entrenamiento, con cambios que dependen del resultado obtenido en solamente dos imágenes y por ello estarán notablemente influenciados por el ruido presente en el *batch*. Un *batch* de dos imágenes no puede representar fielmente el *dataset*, y contendrá ruido que la red tenderá a aprender. A lo largo de varias épocas y *batches*, este ruido se irá cancelando y la red aprenderá la tendencia principal de los datos, pero este ruido será visible en los resultados del entrenamiento. Un *batch size* grande permite promediar los resultados obtenidos sobre más imágenes, y por ello la red se actualizará en base a más información y será menos sensible al ruido. Esto también reduce las probabilidades de que el modelo diverja en las primeras etapas del entrenamiento, lo que ha sido un problema sobre todo a la hora de entrenar la U-Net en el capítulo §4 y al utilizar learning rates inicialmente elevados. Como se muestra en Shen, 2018, un *batch size* pequeño como el que se está utilizando hace que las actualizaciones del gradiente del modelo tras cada *batch* sean de una magnitud muy similar, lo que no siempre interesa. Es preferible dar pasos más grandes al inicio del entrenamiento para facilitar explorar el espacio de soluciones, y pasos más pequeños al final del entrenamiento que permitan al modelo converger a un mínimo local. Este comportamiento se intentará forzar introduciendo utilizando *learning rates* variables.

Como optimizador se ha utilizado Adam [Kingma y Ba, 2017], uno de los algoritmos más robustos al ruido y que ha demostrado una buena capacidad de convergencia. Adam calcula el gradiente de primer orden, y estima los momentos de primer y segundo orden, la media del gradiente y el cuadrado de la media del gradiente respectivamente. Utilizar estos momentos hace que este algoritmo sea muy efectivo a la hora de filtrar el ruido presente en los batches[1] o producido por otras fuentes como el uso de *dropout*. La carga computacional que introduce el cálculo de los momentos es mínima, ya que se reduce a almacenar un valor por cada momento y actualizarlo en cada paso en función del gradiente. Adam es computacionalmente eficiente, requiere de poca memoria y funciona bien con gradientes ruidosos, por lo que es exactamente lo que se busca. Existen variantes interesantes de Adam como

[1] Entrenar con *mini batches* es una forma de *Data subsampling*. Se espera que cada uno de estos *batches* conserve la tendencia del *dataset* completo pero esto no siempre es el caso, lo que introduce ruido en el entrenamiento.

AdamW [Loshchilov y Hutter, 2019] que permiten una mayor parametrización del entrenamiento. Se ha decidido no hacer pruebas con diferentes optimizadores porque no se ha considerado necesario, aunque si se ha hecho algún ensayo con SGD y RMSprop con los que se han conseguido resultados similares; véase Ruder, 2017 para más información sobre estos algoritmos.

Por ahora se ha utilizado un *learning rate* constante, y seguirá así hasta que se exploren diferentes métodos para variar el *learning rate* durante el entrenamiento más adelante en este capítulo. Lo mismo se puede decir de la función de activación, hasta ahora se han utilizado ReLUs convencionales pero se harán pruebas con una variante de estas en este capítulo.

Como función de pérdida se ha utilizado hasta ahora Entropía Binaria Cruzada (BCE). Esta función de pérdida mide la precisión a nivel de píxel y tiene en cuenta la confianza en las predicciones, por lo que es una función relativamente exigente. Utilizar BCE ha permitido obtener segmentaciones precisas, con los bordes muy definidos[2] . Aunque funcione muy bien, no se busca exactamente la precisión a nivel de píxel en esta aplicación, y haber obtenido buenas segmentaciones es solo una consecuencia secundaria de clasificar muchos píxeles correctamente. Ajustar la función de perdida será uno de los objetivos principales de este capítulo.

El *dropout* ya se ha ajustado en el capítulo §4 porque se ha querido tener en cuenta como parte de la arquitectura. Esto es debido a que no hay consenso sobre si se debería utilizar en redes convolucionales, y por ello tiene el potencial de tener un efecto impredecible en el entrenamiento y ser difícil de ajustar. Puesto que el *dropout* es uno de los métodos clave[3] que se han utilizado para combatir el *overfitting*, usar o no *dropout* habría influido en gran medida sobre los entrenamientos y la cantidad de imágenes que se deberían añadir al *dataset* con *data augmentation*. Utilizar *dropout* ha reducido tanto el *overfitting* que se han añadido imágenes con el propósito de codificar invariantes en el *dataset*, no para dificultar el *overfitting*.

Para inicializar los *kernels* se han probado varios métodos, especialmente en el capítulo §4 buscando facilitar la convergencia de la U-Net. Por lo que se ha podido observar, si la red converge adecuadamente el método de inicialización elegido no tiene un efecto apreciable en los resultados obtenidos. De todas formas, se ha utilizado el método de inicialización conocido como *He-Normal* [He *et al.*, 2015a]. Inicializar los pesos de forma aleatoria es extremadamente importante si no se utiliza algún método que introduzca aleatoriedad como el *dropout*. Si al entrenar una red, dos neuronas tienen los mismos pesos a la entrada, los mismos pesos a la salida, el mismo *bias*, y los mismos pesos, entonces tendrán el mismo gradiente y evolucionarán de forma idéntica [*Lecture 6a Overview of mini-batch gradient descent*]. Esto haría que ambas neuronas codifiquen la misma característica, reduciendo la cantidad efectiva de parámetros del modelo. Es evidente que inicializar toda la red con el mismo valor, si no se utiliza *dropout*, derivaría en este problema. Inicializar

[2] Esto se debe a que al utilizar BCE no se buscan solo predicciones acertadas, sino también predicciones con alta confianza. Los bordes de las máscaras son las regiones con menos confianza por lo general, pero utilizar BCE fuerza a que la red decida un borde concreto y no difuso.

[3] Junto con el uso de *Batch Normalization* [Ioffe y Szegedy, 2015]

los *kernels* con valores aleatorios pequeños lo previene.

5.2. Pruebas realizadas

Los hiperparámetros con los que se han hecho pruebas buscan mejorar ciertas características del entrenamiento o del propio modelo que se consideren subóptimas.

En base a las pruebas realizadas y los resultados obtenidos, el límite del rendimiento del modelo actual está en torno a 0.8 de *mean IoU* (mIoU), y para superar este umbral de forma consistente[4] puede ser necesario aumentar la complejidad del modelo. Esto se puede conseguir fácilmente aumentando la cantidad de capas en el modelo o aumentando los canales de características en cada capa, pero las restricciones que impone la memoria en GPU disponible obligan a buscar una forma más eficiente.

Por el uso de un *batch size* tan reducido y por el uso de *dropout* aparecen grandes fluctuaciones en los resultados obtenidos en la validación durante el entrenamiento. Esto se debe a que el propio modelo sufre grandes fluctuaciones. Esto es un problema especialmente al final del entrenamiento porque estas fluctuaciones afectarán al modelo final. Una forma sencilla de forzar al modelo a converger a un mínimo local es reducir considerablemente el *learning rate* al final del entrenamiento.

Lo que potencialmente puede mejorar más el modelo es utilizar una función de *loss* que represente mejor los objetivos considerados. En concreto se buscará codificar que los píxeles del *foreground* son más importantes que los del *background* porque estos son los más difíciles y los que más información codifican.

5.2.1. Función de activación

En este apartado se explorará una variante parametrizada de la función de activación ReLU conocida como *Parametric Rectified Linear Unit* [He *et al.*, 2015b].

Utilizar ReLUs es un estándar *de facto* en aplicaciones de visión artificial porque presenta unas propiedades muy interesantes. El uso de ReLUs facilita en gran medida computar los gradientes[5] , y permite gradientes sencillos que facilitan el entrenamiento del modelo. El uso de ReLUs frente a otro tipo de funciones de activación como la sigmoide es que reducen el problema del *vanishing gradient*, que dificulta y ralentiza la evolución de los pesos de las primeras capas de la red. Un problema que tiene utilizar ReLUs es que si alguna neurona produce siempre una salida negativa para cualquier entrada a la red considerada en los datos de entrenamiento, dejará de evolucionar y efectivamente se perderá. La función ReLU se muestra en la ecuación 5.1 y en la figura 5.2.

[4] Se ha conseguido sobrepasar ligeramente 0.8 de mIoU con el modelo escogido en el capítulo §4, aunque solamente en uno de tres entrenamientos idénticos.

[5] La derivada de la función implementada por una ReLU es 0 o 1 en función del valor de entrada. Otras funciones de activación como la *Sigmoide* modifican mucho más la forma de los gradientes e implican computaciones adicionales.

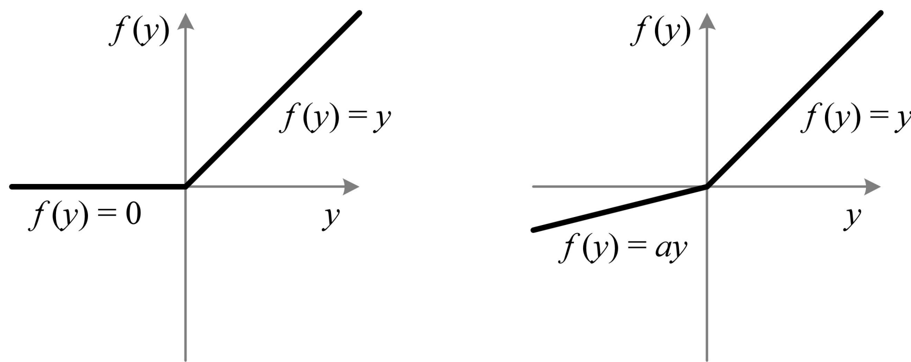


Figura 5.2: Representación gráfica de las funciones ReLU (izquierda) y PReLU (derecha). [*Papers with Code - PReLU Explained*]

$$f(\hat{y}) = \begin{cases} \hat{y} & \text{if } \hat{y} \geq 0 \\ 0 & \text{if } \hat{y} < 0 \end{cases} \tag{5.1}$$

La función PReLU introduce un parámetro adicional α que permite no hacer nulas las activaciones negativas, sino que tomen un valor negativo pequeño; véase la ecuación 5.2 o la figura 5.2. Si α es fijo y pequeño, entonces no es un parámetro y se conoce esta función como *Leaky ReLU*. Si α evoluciona, entonces es un parámetro más que evoluciona durante el entrenamiento. Este parámetro puede ser individual para cada peso o estar compartido entre varios. Lo más normal, y lo que se ha hecho en este caso, es que α sea común a cada canal de características de cada capa. Compartir el parámetro de esta manera permite utilizar PReLUs y solo introducir poco más de 3000 parámetros al modelo. Utilizar esta función de activación añadirá al modelo complejidad de una forma muy eficiente, que le permitirá representar mejor las transformaciones codificadas en el *dataset*.

$$f(\hat{y}) = \begin{cases} \hat{y} & \text{if } \hat{y} \geq 0 \\ \alpha\hat{y} & \text{if } \hat{y} < 0 \end{cases} \tag{5.2}$$

Se ha entrenado la arquitectura ResNet escogida en el capítulo §4. Una ResNet con un *dropout rate* de 0.3. El modelo planteado en este apartado simplemente sustituye las funciones de activación ReLU por PReLU, con un parámetro por cada canal de características como ya se ha mencionado. Los resultados promediados de tres entrenamientos se muestran en la tabla 5.1

Tabla 5.1: Resultados obtenidos utilizando ReLUs y PReLUs como función de activación en el mismo modelo. Resultados promediados de tres entrenamientos idénticos.

Activation function	IOU	Precision	Recall	SMC
ReLU	0.7968	0.9100	**0.8662**	0.8207
PReLU	**0.8011**	**0.9169**	0.8644	**0.8352**

Se han obtenido mejores resultados con el uso de PReLUs como era de esperar.

Por contra la velocidad de inferencia del modelo ha pasado a 68ms/imagen[6] , lo que representa un incremento del 40 %. El tiempo de entrenamiento también ha aumentado, pasando de 5.5h a 6.7h, lo que representa un incremento del 20 %. Es posible que el incremento en precisión conseguido no sea suficiente para justificar la menor velocidad de inferencia en algunas aplicaciones. En este caso se ha decidido seguir optimizando este nuevo modelo porque los resultados obtenidos han sido significativamente mejores.

5.2.2. Función de pérdida

Como ya se ha mencionado, la función de loss que se está utilizando ha permitido buenos resultados, pero no representa exactamente el objetivo considerado en este proyecto. La principal discrepancia es que al utilizar *Entropía Binaria Cruzada* (BCE), los píxeles de *background* y de *foreground* son igual de importantes, y por lo tanto todos los píxeles tienen el mismo efecto en la evolución del modelo. En esta aplicación, y por lo general, los píxeles de *foreground*, los píxeles de manos en este caso, son más importantes porque son más difíciles de predecir y porque codifican más información útil.

En este caso concreto hay un problema añadido derivado de que la mayor parte de los píxeles pertenecen al fondo[7] , que suele ser lo más normal en problemas de segmentación. Que haya más píxeles de fondo implica que en conjunto tienen una influencia mayor sobre el modelo, dándoles así más importancia que no deberían tener. Aunque esto no es algo muy importante al utilizar BCE, porque como los píxeles de fondo son fáciles de predecir, su confianza suele ser elevada, y BCE les quita importancia. Aun así es algo a tener en cuenta.

Se ha decidido probar dos funciones de pérdida diferentes. *DICE Loss* y *Focal Loss*.

DICE Loss

La métrica más utilizada y que mejor representa la calidad de una segmentación en general es su IoU ya que está relacionado directamente con el *recall* y la *precisión*. Es por ello razonable utilizar esta métrica como función de pérdida, lo que alinearía muy bien la función de pérdida con la función objetivo.

En la práctica no se utiliza IoU como función de *loss* sino el coeficiente DICE [Diserud y Ødegaard, 2007], que es esencialmente equivalente pero produce gradientes más sencillos. IoU y el coeficiente DICE son equivalentes como función de pérdida y métrica de la calidad de una segmentación en el sentido de que comparan cualquier dos segmentaciones de la misma forma, existe una correlación positiva entre ambas. Se definen IoU y el coeficiente DICE en función de TP (*True Positives*), FP (*False Positives*) y FN (*False Negatives*) como muestran las ecuaciones 5.3 y 5.4 respectivamente.

[6] En una GPU gtx 1050 e imágenes con una resolución de 640x320 píxeles.
[7] En el conjunto de datos utilizado los píxeles de manos representan alrededor del 8 % de los píxeles.

$$DICE = \frac{2TP}{2TP + FP + FN} \tag{5.3}$$

$$IoU = \frac{TP}{TP + FP + FN} \tag{5.4}$$

La función de pérdida *DICE Loss* se define en función del coeficiente DICE como $1 - DICE$.

Calcular este coeficiente de tal forma que sea derivable implica comparar segmentaciones binarias con segmentaciones no binarias. Esto ha producido mucho ruido durante el entrenamiento y resultados muy pobres. Es de esperar porque esta función de loss representa un problema muy difícil, y por ello muy complicado aprender.

Se ha intentado facilitar el problema polarizando las predicciones realizadas por el modelo utilizando una función sigmoide que se asemeja a una binarización pero es continua y derivable; véase la figura 5.3. Esta transformación de las predicciones esencialmente simula un aumento de la confianza del modelo en sus predicciones. Se ha utilizado una función sigmoide centrada en 0.5, lo que se ha marcado como umbral para las binarizaciones en todos los casos como es costumbre[8] . Los resultados obtenidos utilizando esta transformación fueron marginalmente mejores que sin ella, probablemente porque penaliza en gran medida predicciones con una confianza entorno a 0.5, introduciendo ruido al sistema.

Esta función de pérdida también se ha ponderado con los mapas de pesos obtenidos en la sección §3.1 para poder obtener máscaras de segmentación individuales para cada objeto. En este caso utilizar estos mapas de pesos tiene el efecto de que ciertos píxeles cuenten varias veces como falsos positivos, en función del valor asociado en el mapa. En los casos más extremos, un único píxel puede llegar a contar como doce[9] falsos positivos.

Es posible que ponderar el peso de cada píxel de esta manera y además pseudo-binarizar las segmentaciones inferidas sea el motivo por el que que aparece mucho ruido en la evolución del modelo durante el entrenamiento. En general, los píxeles con altos pesos en el mapa de pesos son difíciles, y tenderán a ser predichos con poca confianza. La pseudo-binarización polarizará estas predicciones, aumentando su confianza para bien o para mal. La influencia de estos píxeles sobre el valor final tendrá mucha varianza ante pequeñas variaciones en la confianza inferida. Dicho esto, utilizar una pseudo-binarización reduce el ruido producido por la mayoría de píxeles, y ponderar los píxeles con mapas de pesos es necesario para la aplicación considerada.

[8] Utilizar umbrales mayores a 0.5 dificulta en gran medida el problema, pero permite entrenar clasificadores más exigentes; véase [Cai y Vasconcelos, 2017].

[9] Por los valores elegidos para computar los mapas de pesos; véase la figura 3.3

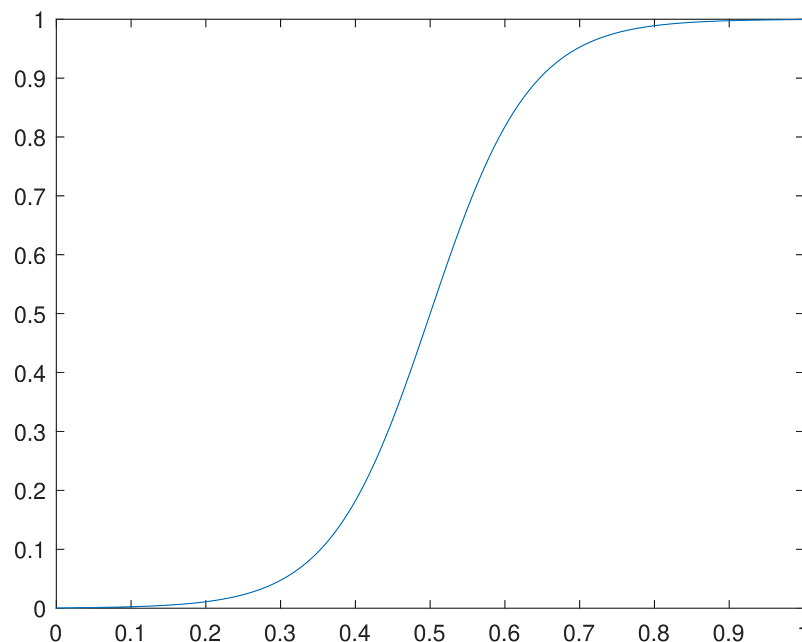


Figura 5.3: Función logística utilizada para aproximar una binarización de la segmentación inferida.

Focal Loss

Focal Loss (FL) [T.-Y. Lin *et al.*, 2018] es una generalización de BCE que introduce dos parámetros adicionales α y γ, utilizados para compensar desequilibrios de la representación de clases en el *dataset* y para dar más importancia a los píxeles más difíciles respectivamente. Para definir FL conviene definir p_t como la confianza en la clase correcta; véanse las ecuaciones 5.5 y 5.6. En la figura 5.4 se muestran las curvas que representan FL con tres valores diferentes de γ, incluido el equivalente a BCE (FL con $\alpha = 0.5$ y $\gamma = 0$).

$$p_t(\hat{y}) = \begin{cases} 1 - \hat{y} & \text{if } y = 0 \\ \hat{y} & \text{if } y = 1 \end{cases} \tag{5.5}$$

$$FL(p_t) = -\alpha(1 - p_t)^{\gamma} \log p_t \tag{5.6}$$

Puesto que FL es una generalización de BCE, es inmediato utilizar los mapas de pesos obtenidos en la sección §3.1 para dar más importancia a ciertos píxeles de la misma forma en la que se han utilizado hasta ahora. El peso definido para cada píxel por FL con unos valores de α y γ concretos se escala de acuerdo con la importancia de ese píxel definida en un mapa de pesos particular de cada segmentación.

Para poder analizar con más detalle el efecto de α y γ, y buscar una buena combinación de los mismos, se ha decidido entrenar el modelo escogido en el capítulo §4, sin PReLUs, con varias combinaciones de α y γ. Los resultados se muestran en la tabla 5.2.

Inicialmente se estableció α en 0.92 para dar la misma importancia global a los

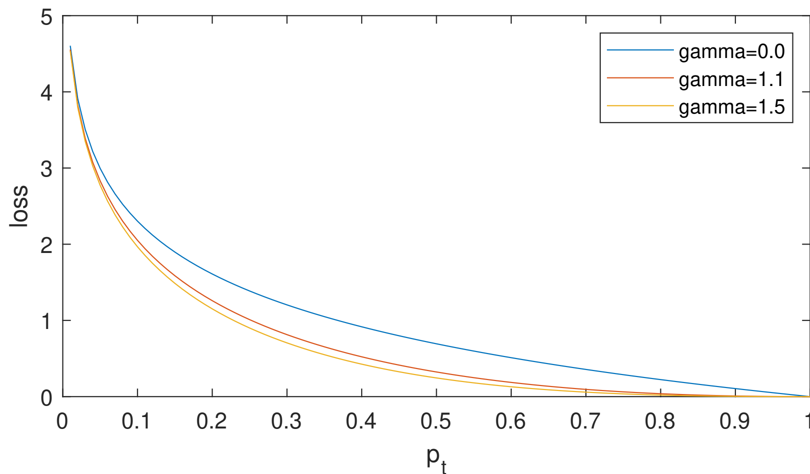


Figura 5.4: Valor de Focal Loss con varios valores de γ en función de la confianza de la clase correcta.

Tabla 5.2: Efecto de los parámetros de Focal Loss en el aprendizaje de la arquitectura ResNet con ReLUs.

Loss function	mIOU	mPrecision	mRecall	mSMC
BCE (FL $\alpha = 0.50$; $\gamma = 0$)	0.7968	0.9100	0.8662	0.8207
FL $\alpha = 0.92$; $\gamma = 1.5$	0.7261	0.7368	**0.9817**	0.7418
FL $\alpha = 0.92$; $\gamma = 1.0$	0.7311	0.7579	0.9580	0.7391
FL $\alpha = 0.50$; $\gamma = 1.0$	**0.7998**	0.9083	0.8716	0.8241
FL $\alpha = 0.50$; $\gamma = 1.2$	0.7936	**0.9139**	0.8585	**0.8257**

píxeles de *background* y *foreground* (sabiendo que el 8 % de los píxeles corresponden a manos), y γ en 1.5 para dar mucha importancia a los píxeles más difíciles. Estos píxeles son en general los píxeles entre manos y los píxeles en bordes. Esta combinación de α y γ resultó en un *recall* demasiado alto, por lo que se ajustaron los parámetros acorde en el siguiente entrenamiento y se repitió el proceso hasta encontrar una combinación adecuada.

Aparentemente $\alpha = 0.5$ es un valor adecuado porque los píxeles de las manos son los más difíciles, y hay suficientes como para que la red se centre en ellos. γ se ha establecido en un valor cercano a uno porque ha dado buenos resultados. Es posible que γ menor que 1.0 produzca mejores resultados, pero en las pruebas realizadas ha dado problemas de convergencia, posiblemente por cómo modifica la forma de los gradientes.

Como se puede ver en los resultados de la tabla 5.2, α y γ permiten un control significativo de los objetivos del modelo. Ha sido muy fácil por ejemplo dar una importancia muy superior a los píxeles del *foreground* con $\alpha = 0.92$, a costa de perder precisión se ha aumentado en gran medida el *recall*.

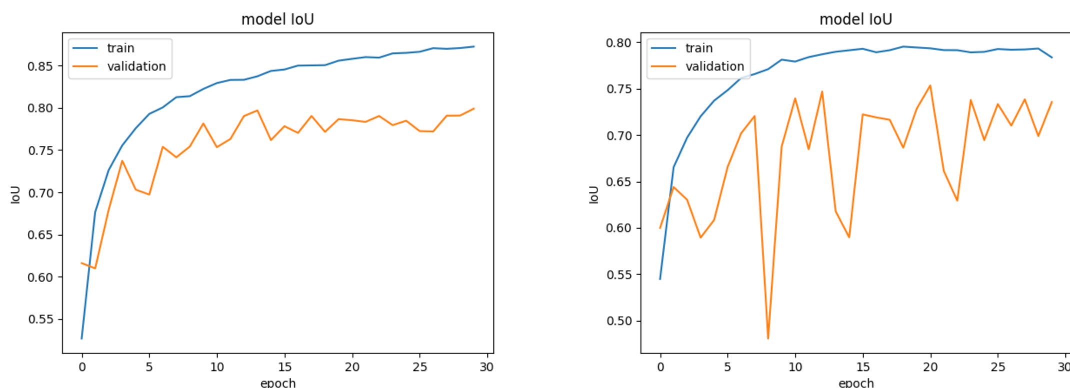
Resultados

Finalmente se entrenó la arquitectura ResNet con PReLUs utilizando *DICE Loss* y varias versiones de *Focal Loss*. En este caso se ha hecho solo un entrenamiento, salvo para FL con $\alpha = 0.5$, $\gamma = 1.1$ y $\alpha = 0.5$, $\gamma = 1.2$ por haber conseguido buenos resultados muy similares.

Tabla 5.3: Resultados obtenidos con varias funciones de loss en la arquitectura ResNet considerada y PReLUs.

Loss function	mIOU	mPrecision	mRecall	mSMC
BCE (FL $\alpha = 0.5$; $\gamma = 0$)	0.8011	0.9169	0.8644	0.8352
FL $\alpha = 0.5$; $\gamma = 1.0$	0.7899	0.8887	0.8798	0.8084
FL $\alpha = 0.6$; $\gamma = 1.0$	0.7957	0.8711	**0.9036**	0.7924
FL $\alpha = 0.5$; $\gamma = 1.1$	**0.8033**	0.9103	0.8749	**0.8357**
FL $\alpha = 0.5$; $\gamma = 1.2$	0.7950	**0.9214**	0.8543	0.8201
DICE Loss	0.7556	0.8400	0.8863	0.7526

Los resultados obtenidos con DICE son muy pobres. Aunque en teoría es una función de pérdida muy adecuada para esta aplicación, es razonable pensar que esto puede ser debido a que DICE es una función demasiado exigente, lo que implica que las evaluaciones durante el entrenamiento contendrán suficiente ruido como para dificultar obtener resultados muy precisos. En la figura 5.5 se puede apreciar la gran cantidad de ruido que aparece en la curva de IoU de validación al utilizar DICE como función de *loss*.



(a) Entropía Binaria Cruzada (BCE).

(b) DICE Loss y la pseudo-binarización descrita por 5.3.

Figura 5.5: Comparación de la evolución del IoU de entrenamiento y validación durante el entrenamiento de una arquitectura ResNet con PReLUs utilizando BCE y DICE Loss.

Los resultados obtenidos con *Focal Loss* son lo esperado, ligeramente mejores que los obtenidos con BCE por ser esencialmente igual pero más versátil. La mejor

combinación encontrada es $\alpha = 0.5$ y $\gamma = 1.1$, se tomará esta como una combinación aceptable, y se utilizará en las siguientes pruebas a realizar para optimizar este nuevo modelo. Utilizar esta función de pérdida incrementa ligeramente el tiempo de entrenamiento del modelo por requerir más cálculos pero no influye en su velocidad de inferencia al no influir en el propio modelo.

5.2.3. Ratio de aprendizaje

Es bueno en las primeras etapas del entrenamiento que el modelo sufra cambios significativos para facilitar la exploración de su espacio de soluciones. Hacia el final del entrenamiento conviene que el modelo varíe poco de un paso de *gradient descent* a otro para minimizar las fluctuaciones finales debidas al ruido y así facilitar la convergencia a un mínimo local.

Como ya se ha mencionado, este comportamiento se puede conseguir utilizando un tamaño de *batch* grande [Shen, 2018]. La idea fundamental es que al principio del entrenamiento muchas predicciones serán mediocres y el valor del *loss* será elevado, al final del entrenamiento muchas predicciones serán buenas y el valor del *loss* se reducirá. El valor del *loss* determina, junto con el ratio de aprendizaje el tamaño del paso que dará el modelo. Utilizar un tamaño de *batch* grande no es una opción por las limitaciones de memoria en GPU disponible, que obligan a utilizar un tamaño de *batch* muy reducido.

Una forma de forzar este comportamiento es utilizar un *learning rate* que varíe a lo largo del entrenamiento de tal forma que sea grande al principio y pequeño al final. Esto se puede conseguir de forma sencilla con un planificador adecuado.

Se ha decidido probar con cuatro tipos de planificadores en cinco pruebas diferentes. El más básico es un planificador lineal, que decae un valor de *learning rate* inicial a un valor final de forma lineal. Se harán dos pruebas con planificadores exponenciales escalonados, que decaen el *learning rate* en un factor constante cada época en este caso. Se utilizará también un planificador cosenoidal que permite utilizar fácilmente valores grandes y pequeños de *learning rate*. Por último se utilizará un planificador que interpole entre un valor inicial y un valor final con una raíz cuadrada, lo que permite tener un ratio de aprendizaje elevado hasta el final del entrenamiento. Los valores concretos de las curvas se han obtenido a partir de escoger valores iniciales y finales. Los valores iniciales no deben ser tan altos como para que el modelo diverja al inicio del entrenamiento, y los valores finales no deben ser demasiado pequeños como para impedir que el modelo siga aprendiendo. En la figura 5.6 se pueden ver los cinco planificadores escogidos para realizar pruebas junto con el planificador constante utilizado hasta ahora.

Los resultados obtenidos de estos entrenamientos se muestran en la tabla 5.4, y son mejores por lo general. Algo común en todos los casos es que se ha reducido en gran medida el ruido presente en las curvas del rendimiento en validación en las últimas épocas del entrenamiento; véase la figura 5.7.

Con el planificador cosenoidal se han obtenido los mejores resultados, que son

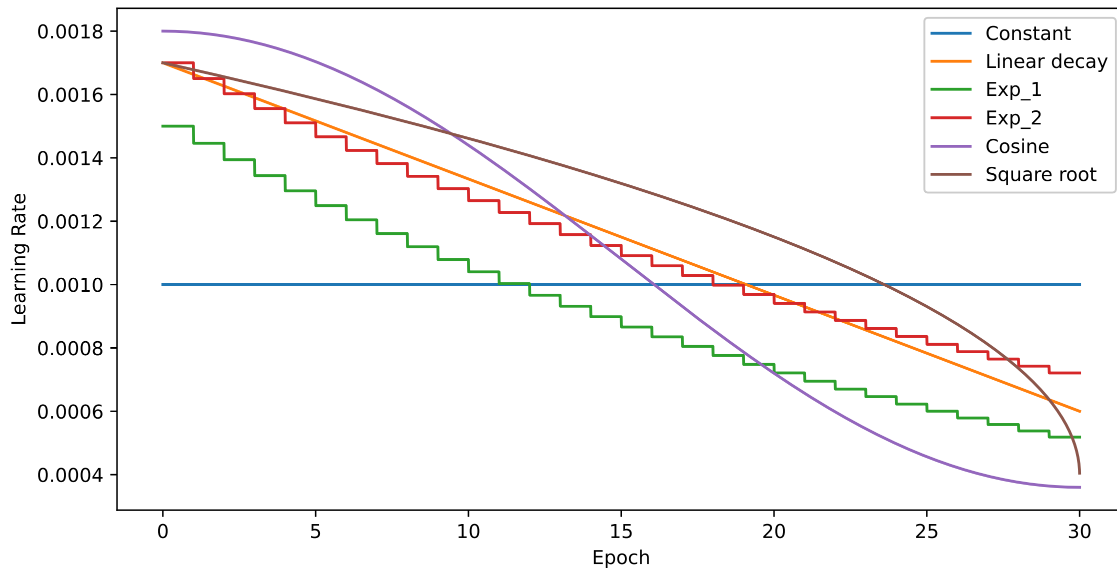
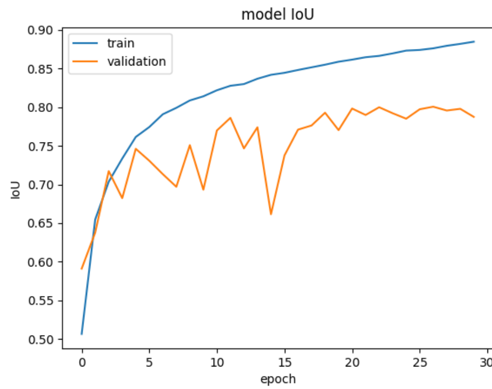


Figura 5.6: Evolución del learning rate con varios planificadores.

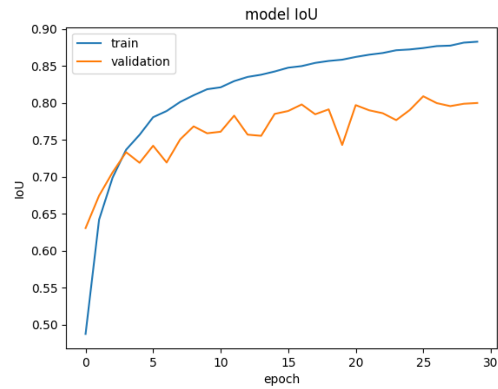
Tabla 5.4: Resultados obtenidos con varias planificaciones del ratio de aprendizaje.

Learning Rate Schedule	mIOU	mPrecision	mRecall	mSMC
Constant	0.8033	0.9103	0.8749	0.8357
Linear decay	0.7943	0.9144	0.8605	0.8335
Exp_1	0.7853	**0.9254**	0.8393	0.8336
Exp_2	0.8037	0.9078	**0.8782**	0.8394
Cosine	**0.8064**	0.9147	0.8736	**0.8403**
Square root	0.8038	0.9187	0.8659	0.8372

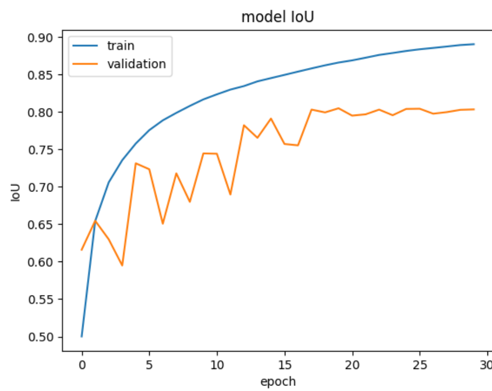
significativamente mejores que los obtenidos con un ratio de aprendizaje constante. Este será el modelo escogido, fruto de una optimización simple de hiperparámetros. Idealmente este proceso se iteraría hasta no obtener variaciones significativas de los hiperparámetros, pero eso incrementaría el sesgo de los conjuntos de datos de validación y *test*, y el margen de mejora probablemente ya sea demasiado estrecho como para justificarlo.



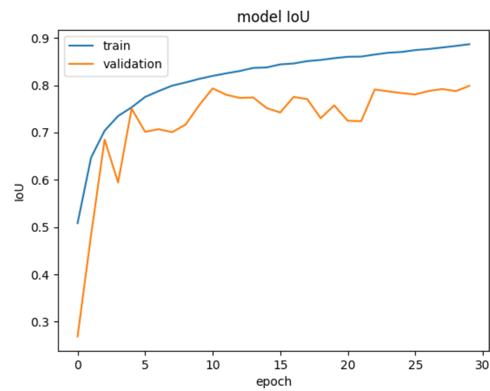
(a) Linear decay



(b) Exp_2



(c) Cosine



(d) Square root

Figura 5.7: IoU de entrenamiento y validación con cuatro planificaciones diferentes del learning rate; véase la figura 5.6. Se puede apreciar una reducción clara del ruido en las curvas de validación en todos los casos como era de esperar.

Capítulo 6

Resultados

El objetivo de este capítulo es mostrar los resultados obtenidos con el modelo diseñado y compararlos con los resultados obtenidos por otros investigadores sobre el mismo conjunto de datos. Se hará además un pequeño análisis del funcionamiento interno de la red.

Los resultados se mostrarán sobre imágenes del *dataset* de *test*, que es el subconjunto de imágenes con menos sesgo debido al sobreajuste.

6.1. Visualización de los resultados

Las segmentaciones obtenidas son relativamente buenas por lo general. Salvo en los casos más exigentes, se cumple el objetivo de obtener máscaras independientes para cada mano en la segmentación, y las propias segmentaciones son precisas y tienen buena definición. Cuatro imágenes del conjunto de datos de *test* bien segmentadas se muestran en la figura 6.1.

Haber obtenido máscaras independientes en las segmentaciones se debe al procesado del *dataset* realizado en el capítulo §3.1 y al uso de funciones de pérdida ponderadas con los mapas de pesos obtenidos. Dos ejemplos de máscaras independientes conseguidas en casos exigentes se muestran en las figuras 6.1a y 6.1c, donde se puede apreciar que el modelo ha aprendido a dejar un margen considerable entre instancias diferentes.

Esta separación no se ha conseguido obtener en todos los casos; ejemplos de esta clase de fallos se muestran en la figura 6.2. Haber entrenado el modelo teniendo en cuenta esta restricción reduce además el mIoU esperado porque no solo se dificulta así el problema a resolver, sino que la propia métrica IoU deja de representar fielmente el objetivo considerado. Por este motivo se ha diseñado una métrica que pudiese complementar a otras métricas como IoU para evaluar el cumplimiento de los requisitos; véase el apéndice appendix B para más información sobre esta métrica.

Se puede apreciar en las imágenes mostradas en las figuras 6.1 y 6.3 que el modelo diseñado tiende a predecir áreas más pequeñas que las definidas por el *ground truth*, en especial por los bordes, como es de esperar. Esto concuerda con los valores de *Precisión* y *Recall* obtenidos, que confirman que el modelo tiende a realizar

predicciones relativamente conservadoras. Tener más *Precisión* que *Recall* es algo que se ha buscado durante el diseño del modelo porque facilita obtener máscaras disjuntas para cada instancia, aunque una diferencia grande entre ellas tendría un efecto negativo sustancial sobre el mIoU.

En la figura 6.3 se muestran dos ejemplos de segmentaciones obtenidas de dos imágenes con instancias de manos diferentes solapadas en las que se han obtenido máscaras independientes para cada instancia. Esto representa el caso más difícil, y aun así el modelo ha sido capaz de proporcionar el resultado deseado.

6.2. Comparación con otros modelos

En este apartado se busca contextualizar el rendimiento obtenido con el modelo diseñado comparándolo con los resultados obtenidos por otros investigadores sobre el mismo conjunto de datos. En concreto se compararán los resultados con los obtenidos por Khan y Borji utilizando RefineNet [G. Lin *et al.*, 2017], una arquitectura moderna.

Tabla 6.1: Comparación de los resultados obtenidos sobre EgoHands con los obtenidos por Khan y Borji [Khan y Borji, 2018]. En ambos casos se han utilizando la misma cantidad de imágenes originales para entrenar, 3600 imágenes del dataset original. En este proyecto se han utilizado además otras 1222 imágenes para entrenar obtenidas a partir del dataset de entrenamiento original mediante data augmentation.

Loss function	mIOU	mPrecision	mRecall	mSMC
Khan y Borji	**0.814**	0.879	**0.919**	—
Modelo obtenido	0.806	**0.915**	0.874	0.840

Los resultados mostrados en la tabla 6.1 son muy comparables con los obtenidos por el modelo escogido en el capítulo §5. En base a los valores de *Precisión* y *Recall* obtenidos por Khan y Borji, se puede ver que este modelo tiende a predecir máscaras de segmentación más grandes, por lo que, aunque sean resultados mejores en general, son peores para la aplicación considerada en este trabajo.

Se puede concluir fácilmente a partir de estos resultados, que el modelo diseñado es un éxito. El mIoU conseguido es significativamente menor que lo encontrado en la literatura, pero la diferencia es despreciable si se tienen en cuenta las restricciones adicionales consideradas y las limitaciones impuestas por los recursos disponibles.

La velocidad de inferencia del modelo es de 68ms/imagen o 14fps en el *hardware* utilizado, lo que es ligeramente inferior a lo comúnmente considerado como *tiempo real*. El principal cuello de botella es la cantidad de memoria en GPU disponible, por lo que es fácil conseguir una velocidad de inferencia mayor utilizando un *hardware* más potente.

6.3. Análisis de los resultados

Las redes convolucionales consiguen detectar objetos a partir de construir un conjunto jerárquico de representaciones de la imagen de entrada, y utilizar estas representaciones para detectar objetos. Esta jerarquía de representaciones se obtiene a partir de extraer características de bajo nivel de la imagen y combinarlas para obtener características de medio y, finalmente, alto nivel.

En este apartado se mostrarán algunas de las características inferidas en una imagen por la red para aportar una intuición sobre su funcionamiento interno. Esto se conseguirá mostrando resultados intermedios obtenidos por la red, en concreto la salida de las funciones de activación en diferentes niveles de la arquitectura.

La figura 6.5 muestra la salida de una de las primeras capas de la red, y se puede apreciar que la red está aplicando operaciones básicas para detectar bordes de objetos en varias direcciones y colores. En esta etapa del procesado la imagen original aún es reconocible.

La figura 6.6 muestra la salida de una de las capas intermedias del codificador. Se pueden apreciar ciertas formas de la imagen original, así como rectas horizontales y bordes de objetos. Estas son representaciones más abstractas de la imagen original, pero aún se puede reconocer la imagen.

En la figura 6.7 se puede ver la salida del codificador de la red, donde se puede esperar tener la representación más abstracta y con menos información posicional de la imagen original. Lo poco que se puede distinguir fácilmente son ciertas formas y líneas horizontales, verticales y oblicuas, además de bordes de objetos.

La figura 6.8 muestra la salida de una de las capas intermedias del codificador. Se pueden distinguir perfiles y contornos de objetos del *foreground* y objetos que están asociados, en concreto manos y brazos. Cabe destacar que estos objetos aparecen mucho más aislados y descontextualizados que en representaciones de más bajo nivel.

En la figura 6.9 se muestra la salida de una de las capas finales del decodificador. Aquí aparecen las representaciones de más alto nivel, como era de esperar, por lo que es fácil distinguir objetos. A partir de estas representaciones se puede intuir fácilmente el resultado final. Aparecen varios tipos de representaciones de manos, y se puede asumir que la predicción se obtendrá a partir de un consenso de todas ellas.

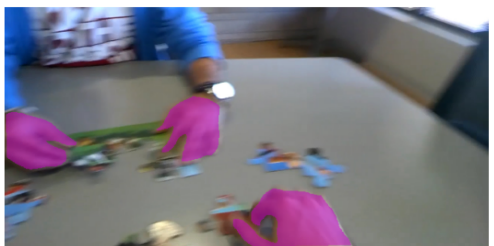
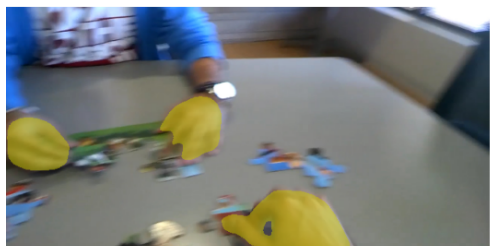
Por último se muestra la salida de la red, y la máscara se segmentación obtenida a partir de binarizar la salida en la figura 6.10.

Este conjunto de resultados intermedios muestra la jerarquía de representaciones de bajo, medio, y alto nivel que la una red neuronal aprende a inferir para hacer predicciones.

(a) Varias manos bien segmentadas con máscaras individuales.

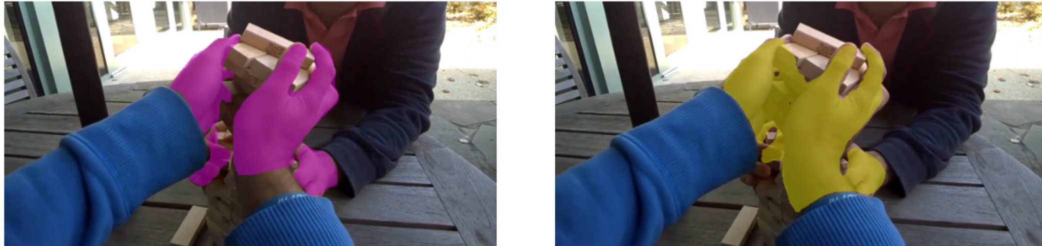
(b) Fondo complicado.

(c) Manos juntas bien segmentadas individualmente.

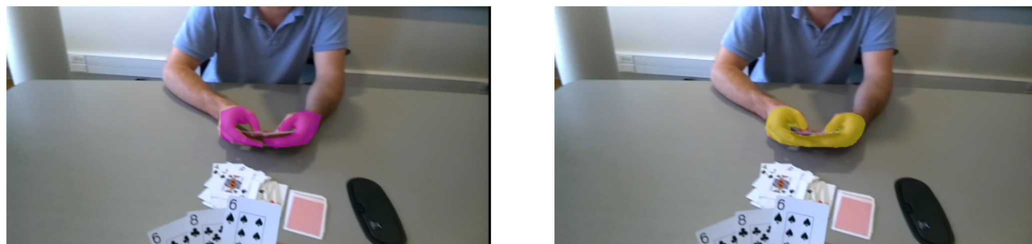
(d) Imagen borrosa

Figura 6.1: Ejemplos de buenas segmentaciones (IoU > 0.8) incluso en casos exigentes. En magenta se muestra el *ground truth* y en amarillo las máscaras inferidas.



(a) Dos manos bajo una única máscara de segmentación.

(b) Máscara de segmentación de una instancia fragmentada.



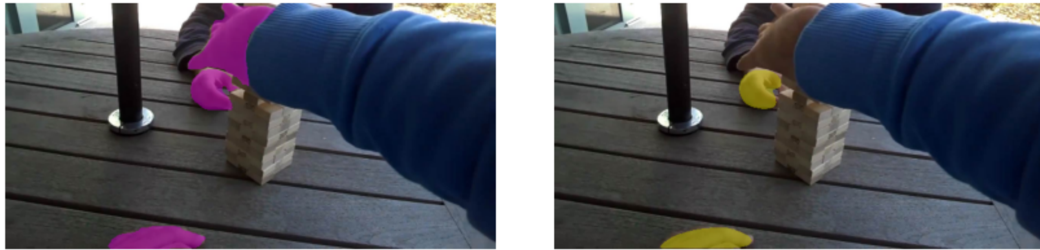
(c) Dos manos bajo una única máscara de segmentación.

Figura 6.2: Segmentaciones con buen IoU pero que no han proporcionado una máscara de segmentación para cada instancia. En magenta se muestra el ground truth y en amarillo las máscaras inferidas.

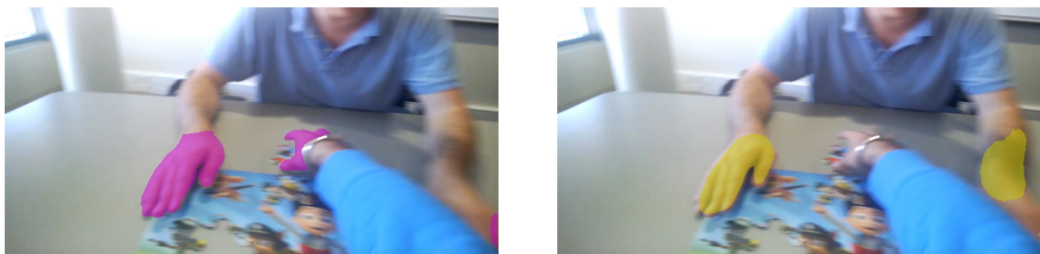
(a)

(b)

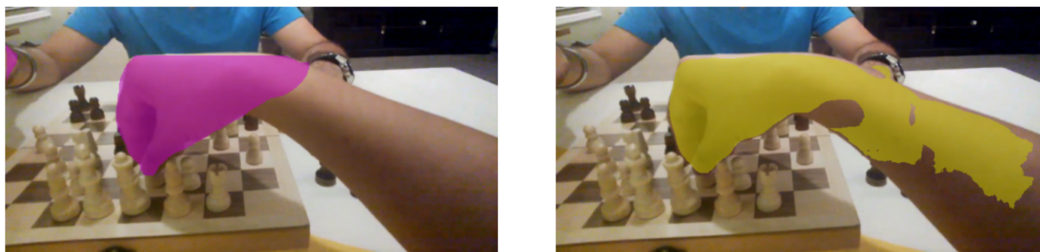
Figura 6.3: Manos correctamente segmentadas con una máscara para cada instancia en dos casos difíciles. En magenta se muestra el ground truth y en amarillo las máscaras inferidas.



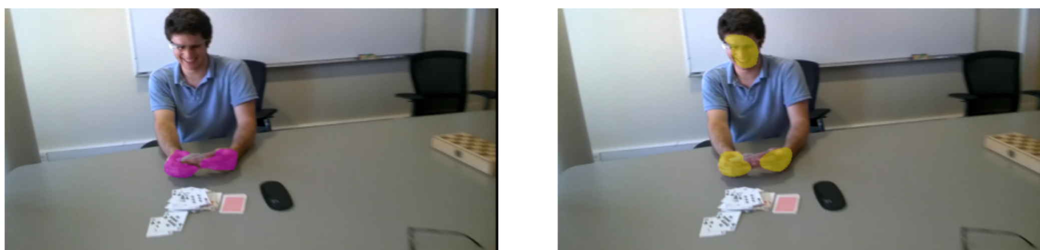
(a) Mala iluminación.



(b) Imagen borrosa.



(c) Dos tonalidades en la piel.



(d) Persona alejada.

Figura 6.4: Ejemplos de malas segmentaciones (IoU < 0.6) en casos difíciles por diversos motivos. En magenta se muestra el ground truth y en amarillo las máscaras inferidas.

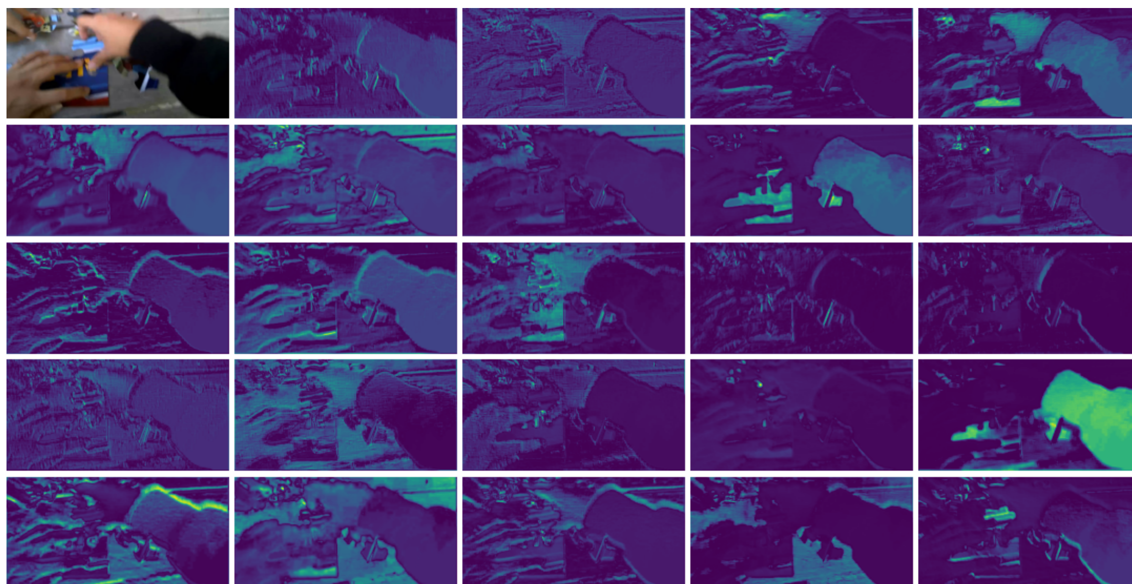


Figura 6.5: Salida del primer bloque convolucional del encoder.

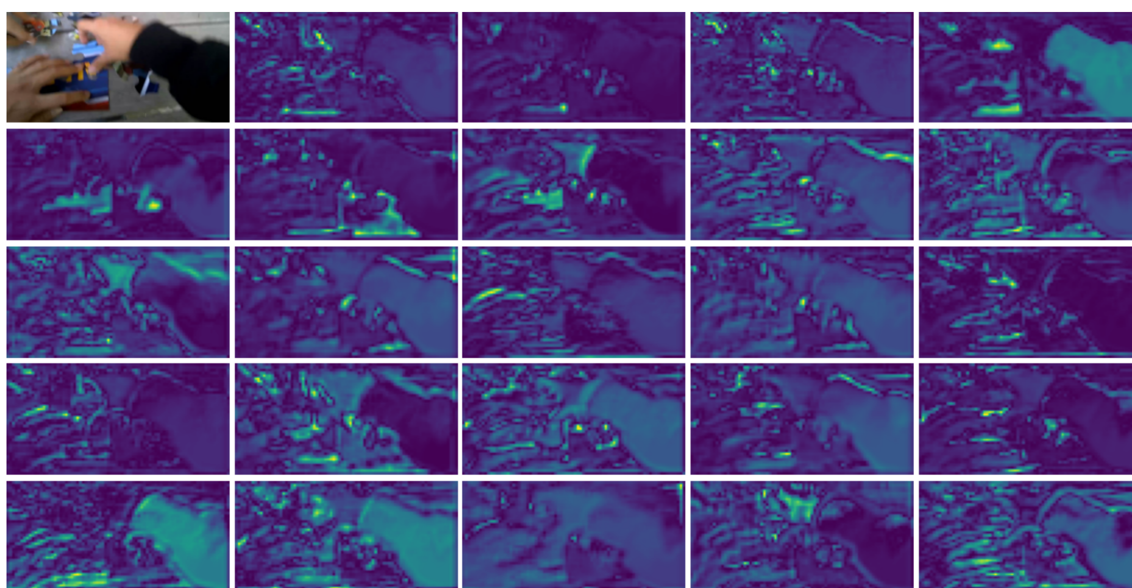


Figura 6.6: Salida de un bloque convolucional intermedio del encoder.

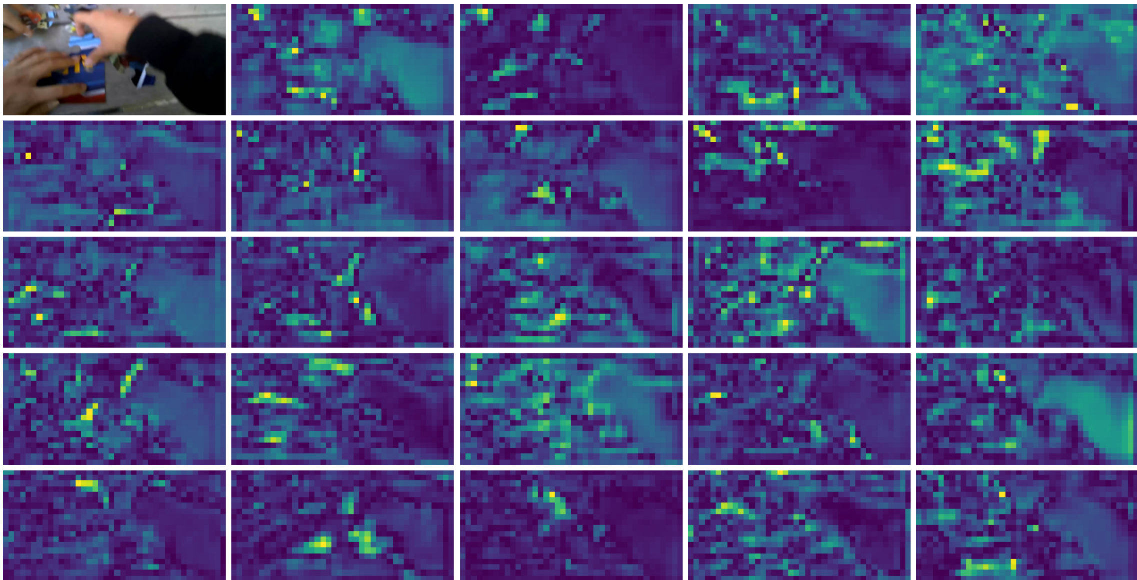


Figura 6.7: Salida del cuello de botella de la arquitectura, último bloque convolucional del encoder, y entrada del decoder.

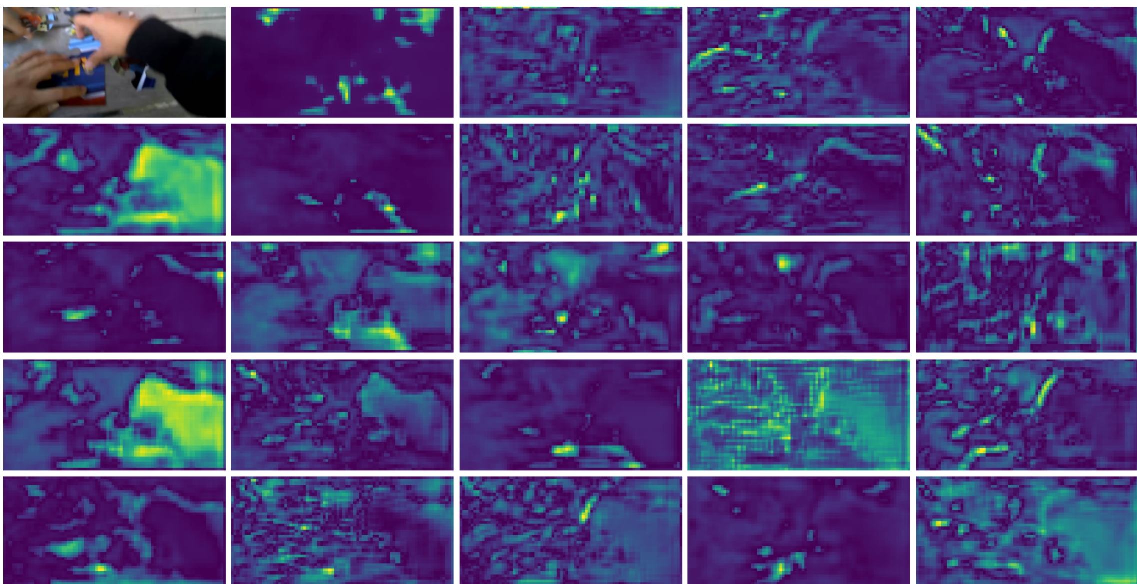


Figura 6.8: Salida de un bloque convolucional intermedio del decoder.

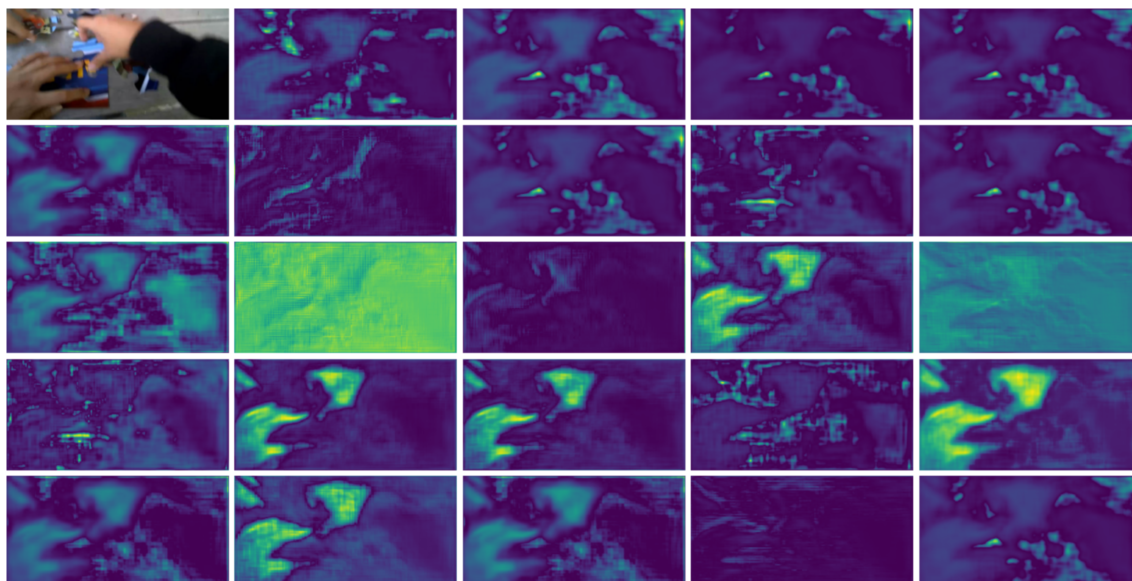


Figura 6.9: Salida del bloque final del decoder.

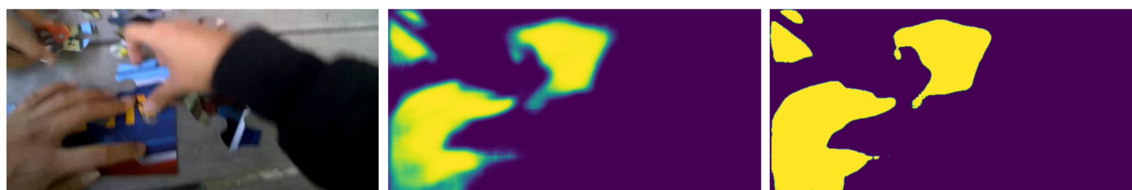


Figura 6.10: Salida de la red y máscara de segmentación obtenida de binarizar la salida.

Capítulo 7

Conclusiones

Por último, se evaluarán los resultados obtenidos en función de los objetivos de este proyecto, y se comentarán líneas de trabajo que pueden derivar de este proyecto.

7.1. Conclusiones

El modelo desarrollado es capaz de producir resultados muy precisos, comparables con los encontrados en la literatura. Se ha conseguido cumplir además, por lo general, el objetivo establecido inicialmente de obtener máscaras de segmentación independientes para cada instancia, lo que complica sustancialmente el problema de la segmentación semántica.

Uno de los objetivos centrales de este proyecto ha sido explorar la posibilidad de utilizar las arquitecturas de redes neuronales sencillas utilizadas en segmentación semántica para obtener una aproximación razonable a segmentación de instancias. El objetivo es realizar segmentación semántica en imágenes pero consiguiendo *blobs* independientes para cada instancia. Esto se ha conseguido con un preprocesado del conjunto de datos en el que se detectaban los píxeles más importantes de cada segmentación, en este caso los píxeles que separan dos instancias, para poder darles más importancia durante el entrenamiento. La metodología expuesta puede generalizar a cualquier aplicación, apenas complica el diseño y entrenamiento de los modelos, y solamente requiere un postprocesado sencillo de la segmentación inferida para detectar instancias individuales.

De esta forma se han obtenido resultados muy precisos, comparables con los proporcionados por los algoritmos de segmentación más recientes, y además se proporcionan máscaras individuales para cada instancia.

En este caso es relativamente sencillo forzar al modelo a separar instancias diferentes porque en las imágenes aparece, por lo general, una cantidad reducida de manos y estas tienen un tamaño considerable. Aun así el problema no es trivial porque la frontera entre dos manos no es fácilmente reconocible.

Es posible que en otro tipo de aplicaciones que introduzcan nuevas restricciones, no sea una opción segmentar instancias de esta manera. Esto se debe a que este método complica el problema original de la segmentación semántica, debiendo llegar a

un compromiso entre segmentación semántica y segmentación de instancias.

Con los resultados obtenidos, podemos concluir que se ha logrado el objetivo final del proyecto. El cumplimiento del resto de objetivos considerados en el capítulo §1 deriva de haber mostrado y comentado el proceso de diseño del modelo, justificando las decisiones tomadas en base a los resultados obtenidos durante el proceso.

7.2. Líneas futuras

Hay ciertos aspectos a tener en cuenta para completar este proyecto que no se han podido llevar a cabo, y por ello se sugieren como líneas futuras. Se consideran tanto formas de mejorar el proyecto, como lo necesario para aprovechar los resultados. Estas tareas son:

- Continuar el ajuste fino del modelo probando optimizadores más sofisticados como AdamW [Loshchilov y Hutter, 2019].
- Estimar la precisión del modelo final una vez se haya desplegado el modelo. Se sugiere comprobar la precisión del modelo sobre otro conjunto de datos de segmentación de manos como el *Georgia Tech Egocentric Activities (GTEA) dataset* [Fathi *et al.*, 2011]. Interesan los resultados obtenidos sin entrenar con este conjunto de datos y después de haber reentrenado el modelo con muy pocas imágenes de este conjunto. Esto se debe a que se debería entrenar el modelo con imágenes del entorno objetivo antes de desplegar el modelo.
- Diseñar e implementar el postprocesado necesario para filtrar y aprovechar las máscaras de segmentación inferidas por el modelo.
- Desplegar el modelo integrándolo en el sistema de realidad aumentada donde se utilizará.
- Estimar la velocidad de funcionamiento del modelo sobre el *hardware* objetivo y modificar la arquitectura del modelo si es necesario.

El sistema de detección de manos obtenido se ha diseñado de forma genérica, por lo que no está restringido a la aplicación considerada, en un sistema de rehabilitación de mano potenciado por el uso de realidad aumentada. Este proyecto puede contribuir por lo tanto a futuros proyectos similares, como por ejemplo un sistema de rehabilitación de muñeca.

Bibliografía

1. Apache MXNet. Consultado el 24 de junio de 2022, desde https://mxnet.apache.org/versions/1.9.1/
2. Bambach, S., Lee, S., Crandall, D. J. & Yu, C. (2015). Lending A Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions. En *The IEEE International Conference on Computer Vision (ICCV)*.
3. Boser, B. E., Guyon, I. M. & Vapnik, V. N. (s.f.). A training algorithm for optimal margin classifiers. En *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory* (pp. 144-152).
4. Cai, Z. & Vasconcelos, N. (2017). Cascade R-CNN: Delving into High Quality Object Detection. *arXiv:1712.00726 [cs]*. arXiv: 1712.00726. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/1712.00726
5. Ciresan, D., Meier, U. & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. En *2012 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3642-3649). doi:10.1109/CVPR.2012.6248110
6. Cortes, C. & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273-297. doi:10.1007/BF00994018
7. Dai, J., He, K., Li, Y., Ren, S. & Sun, J. (2016). Instance-sensitive Fully Convolutional Networks. Number: arXiv:1603.08678 arXiv:1603.08678 [cs]. arXiv. Consultado el 17 de mayo de 2022, desde http://arxiv.org/abs/1603.08678
8. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. & Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. En *CVPR09*.
9. Diserud, O. H. & Ødegaard, F. (2007). A multiple-site similarity measure. *Biology Letters*, *3*(1), 20-22. doi:10.1098/rsbl.2006.0553
10. Fathi, A., Ren, X. & Rehg, J. M. (2011). Learning to recognize objects in egocentric activities. En *CVPR 2011* (pp. 3281-3288). doi:10.1109/CVPR.2011.5995444
11. Girshick, R. (2015). Fast R-CNN. En *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
12. Girshick, R., Donahue, J., Darrell, T. & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv:1311.2524 [cs]*. arXiv: 1311.2524. Consultado el 20 de abril de 2022, desde http://arxiv.org/abs/1311.2524

13. He, K., Zhang, X., Ren, S. & Sun, J. (2015a). Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*. arXiv: 1512.03385. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/1512.03385

14. He, K., Zhang, X., Ren, S. & Sun, J. (2015b). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*. arXiv: 1502.01852. Consultado el 1 de mayo de 2022, desde http://arxiv.org/abs/1502.01852

15. Hinton, G., Srivastava, N. & Swersky, K. Lecture 6a Overview of mini-batch gradient descent. Consultado desde http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

16. Ioffe, S. & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*. arXiv: 1502.03167. Consultado el 2 de mayo de 2022, desde http://arxiv.org/abs/1502.03167

17. ITAP Medical Robotics. Consultado el 22 de junio de 2022, desde https://roboticamedica.itap.uva.es/projects/robhand

18. ITAP Medical Robotics - Research. Consultado el 20 de junio de 2022, desde https://roboticamedica.itap.uva.es/research/

19. Keras: the Python deep learning API. Consultado el 24 de junio de 2022, desde https://keras.io/

20. Khan, A. U. & Borji, A. (2018). Analysis of Hand Segmentation in the Wild. arXiv:1803.03317 [cs]. arXiv. Consultado el 28 de junio de 2022, desde http://arxiv.org/abs/1803.03317

21. Kingma, D. P. & Ba, J. (2017). Adam: A Method for Stochastic Optimization. Number: arXiv:1412.6980 arXiv:1412.6980 [cs]. arXiv. Consultado el 7 de junio de 2022, desde http://arxiv.org/abs/1412.6980

22. LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, *1*(4), 541-551. doi:10.1162/neco.1989.1.4.541

23. LeCun, Y. & Cortes, C. (2010). MNIST handwritten digit database. Consultado desde http://yann.lecun.com/exdb/mnist/

24. Li, H., Xu, Z., Taylor, G., Studer, C. & Goldstein, T. (2018). Visualizing the Loss Landscape of Neural Nets. Number: arXiv:1712.09913 arXiv:1712.09913 [cs, stat]. arXiv. Consultado el 19 de junio de 2022, desde http://arxiv.org/abs/1712.09913

25. Lin, G., Milan, A., Shen, C. & Reid, I. (2017). RefineNet: Multi-path Refinement Networks for High-Resolution Semantic Segmentation. En *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 5168-5177). doi:10.1109/CVPR.2017.549

26. Lin, T.-Y., Goyal, P., Girshick, R., He, K. & Dollár, P. (2018). Focal Loss for Dense Object Detection. *arXiv:1708.02002 [cs]*. arXiv: 1708.02002. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/1708.02002

27. Loshchilov, I. & Hutter, F. (2019). Decoupled Weight Decay Regularization. Number: arXiv:1711.05101 arXiv:1711.05101 [cs, math]. arXiv. Consultado el 19 de junio de 2022, desde http://arxiv.org/abs/1711.05101

28. Papers with Code - COCO test-dev Benchmark (Instance Segmentation). Consultado el 25 de junio de 2022, desde https://paperswithcode.com/sota/instance-segmentation-on-coco

29. Papers with Code - PReLU Explained. Consultado el 26 de junio de 2022, desde https://paperswithcode.com/method/prelu

30. Phan, H. L., Le, T. H., Lim, J. M., Hwang, C. H. & Koo, K.-i. (2022). Effectiveness of Augmented Reality in Stroke Rehabilitation: A Meta-Analysis. *Applied Sciences*, *12*(4), 1848. doi:10.3390/app12041848

31. PyTorch. Consultado el 24 de junio de 2022, desde https://www.pytorch.org

32. Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640 [cs]*. arXiv: 1506.02640. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/1506.02640

33. Redmon, J. & Farhadi, A. (2016). YOLO9000: Better, Faster, Stronger. *arXiv:1612.08242 [cs]*. arXiv: 1612.08242. Consultado el 2 de mayo de 2022, desde http://arxiv.org/abs/1612.08242

34. Redmon, J. & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. *arXiv:1804.02767 [cs]*. arXiv: 1804.02767. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/1804.02767

35. Ren, S., He, K., Girshick, R. & Sun, J. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(6), 1137-1149. doi:10.1109/TPAMI.2016.2577031

36. Ronneberger, O., Fischer, P. & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv:1505.04597 [cs]*. arXiv: 1505.04597. Consultado el 9 de mayo de 2022, desde http://arxiv.org/abs/1505.04597

37. Ruder, S. (2017). An overview of gradient descent optimization algorithms. Number: arXiv:1609.04747 arXiv:1609.04747 [cs]. arXiv. Consultado el 25 de junio de 2022, desde http://arxiv.org/abs/1609.04747

38. Saposnik, G., Levin, M. & for the Stroke Outcome Research Canada (SORCan) Working Group. (2011). Virtual Reality in Stroke Rehabilitation: A Meta-Analysis and Implications for Clinicians. *Stroke*, *42*(5), 1380-1386. doi:10.1161/STROKEAHA.110.605451

39. Shen, K. (2018). Effect of batch size on training dynamics. Consultado el 19 de junio de 2022, desde https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e

40. Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*. arXiv: 1409.1556. Consultado el 20 de abril de 2022, desde http://arxiv.org/abs/1409.1556

41. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, *15*(56), 1929-1958. Consultado desde http://jmlr.org/papers/v15/srivastava14a.html

42. Stallkamp, J., Schlipsing, M., Salmen, J. & Igel, C. (2011). The German Traffic Sign Recognition Benchmark: A multi-class classification competition. En *IEEE International Joint Conference on Neural Networks* (pp. 1453-1460).

43. Szegedy, C., Ioffe, S., Vanhoucke, V. & Alemi, A. (2016). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *arXiv:1602.07261 [cs]*. arXiv: 1602.07261. Consultado el 4 de mayo de 2022, desde http://arxiv.org/abs/1602.07261

44. Team, K. Keras documentation: Image segmentation with a U-Net-like architecture. Consultado el 6 de julio de 2022, desde https://keras.io/examples/vision/oxford_pets_image_segmentation/

45. TensorFlow. Consultado el 24 de junio de 2022, desde https://www.tensorflow.org/?hl=es-419

46. U-net, dropout, augmentation, stratification. Consultado el 6 de julio de 2022, desde https://kaggle.com/code/phoenigs/u-net-dropout-augmentation-stratification

47. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention Is All You Need. *arXiv:1706.03762 [cs]*. arXiv: 1706.03762. Consultado el 20 de abril de 2022, desde http://arxiv.org/abs/1706.03762

48. Zhang, H., Li, F., Liu, S., Zhang, L., Su, H., Zhu, J., … Shum, H.-Y. (2022). DINO: DETR with Improved DeNoising Anchor Boxes for End-to-End Object Detection. *arXiv:2203.03605 [cs]*. arXiv: 2203.03605. Consultado el 27 de abril de 2022, desde http://arxiv.org/abs/2203.03605

Apéndice A

Support Vector Machines

Los clasificadores basados en *Support Vector Machines* son muy interesantes por su relevancia en la historia del el aprendizaje automático, y como un fundamento del funcionamiento de los modelos utilizados a día de hoy. Estos modelos representan parte de los intentos exitosos de implementar técnicas de aprendizaje automático, antes de ser reemplazados por métodos más potentes fruto de los avances en hardware y computación de los últimos años. Considero que son una versión menos abstracta de las redes neuronales actuales, e implementan un algoritmo suficientemente simple como para aportar una intuición muy útil sobre el funcionamiento de métodos más modernos.

Support Vector Machines (SVMs) [B. E. Boser *et al.*, s.f.] son modelos capaces de aprender mediante entrenamiento supervisado una función de decisión para clasificar muestras en una de dos categorías. Para poder clasificar un objeto con este sistema es necesario extraer un conjunto predefinido de características que lo definen. Cada una de estas muestras representa un punto o un vector en el espacio de características, y se clasificará en base a su localización en este espacio.

La idea es inferir a partir de conjunto de muestras ya clasificadas, el hiperplano óptimo[1] que divide este espacio en dos regiones, una para cada una de las dos categorías. Que la función de decisión sea un hiperplano implica que los dos grupos deben ser linealmente separables en el espacio de características. La función de decisión tiene propiedades que permiten gran capacidad de generalización, y por lo general es tanto mejor cuanto más ancha sea la frontera entre los dos grupos [Cortes y Vapnik, 1995].

Este hiperplano representa la función de decisión, y define una frontera entre las dos clases. Véase la figura A.1. La división es óptima respecto al conjunto de datos utilizado, por lo que está limitado por la cantidad de datos y por la calidad de las características que definen las muestras.

Un algoritmo de clasificación basado en SVMs (Conocido como *Support vector Classifier* o SVC) es similar a *k-means* y *k-NN* (*k-nearest neighbours*) puesto que clasifica nuevas muestras a partir de un conjunto clasificado de datos. Una diferen-

[1] El hiperplano óptimo se define como la función de decisión lineal con más margen entre los puntos de las dos clases.

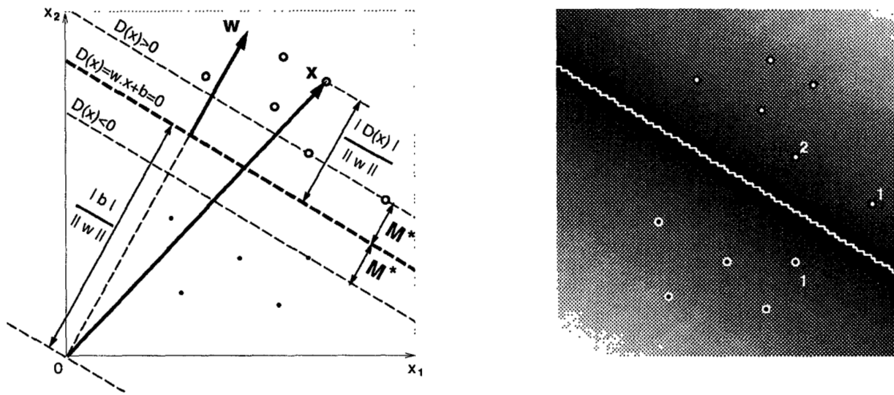


Figura A.1: Ejemplo gráfico de la función de decisión obtenida a partir de un conjunto de muestras en un espacio de dos dimensiones. $x1$ y $x2$ representan las dos características que definen una muestra en este sistema. Podrían representar propiedades tangibles del objeto que representan o algo más abstracto. [Cortes y Vapnik, 1995]

cia fundamental con estos dos algoritmos es que un SVC tiene parámetros internos, por lo que necesita un entrenamiento. Además, como se utiliza un hiperplano para definir la frontera entre las dos categorías, al menos en el algoritmo básico, solo se pueden clasificar dos categorías a la vez. Otra diferencia fundamental es que k-means y k-NN filtran fácilmente cualquier *outlier*, mientras que las SVMs tienen en cuenta todas las muestras. Pero, aunque se tengan en cuenta todas las muestras, solo los llamados *support vectors* definen la superficie de decisión. Estos *support vectors* son las muestras que caen justo en el margen de la frontera, y son los puntos más restrictivos por su distancia al otro conjunto de datos.

En 1995 Cortes y Vapnik generalizaron en este concepto a datos de entrenamiento no linealmente separables con el concepto de *Support Vector Networks* ibíd. La idea es mapear no linealmente los puntos de datos desde el espacio de características original a un espacio de características de muy alta dimensión utilizando un mapeado escogido a priori. A continuación, es en este espacio de características donde se obtiene la frontera. Al aplicar una transformación no lineal, se pueden conseguir dos conjuntos linealmente separables, aunque no lo fuesen en el espacio original. Pasar a un espacio de muy alta dimensión facilita que exista un hiperplano que pueda separar los dos conjuntos de puntos, y con este objetivo se introduce además el concepto de *soft margin*, que permite ciertos errores de clasificación en los datos de entrenamiento.

R-CNN es una *Support Vector Network* generalizada a un número arbitrario de clases por utilizar múltiples clasificadores basados en SVMs. R-CNN utiliza una red convolucional para extraer características abstractas de cada región. Además de escoger previamente regiones interesantes para concentrar la búsqueda.

Es interesante la comparación entre una red neuronal densa sencilla y un SVC. En una red formada por densas, cada una de las neuronas implementa una frontera definida por un hiperplano. El conjunto de neuronas implementa una superficie lineal a trozos para dividir el espacio de características. Esto permite un comportamiento

mucho más complejo que un simple SVM. Un SVC implementa una función lineal, pero al transformar no linealmente los vectores de entrada, consigue implementar una función de decisión no lineal en el espacio original.

Apéndice B

Métrica de la similitud media entre centroides

Obtener máscaras de segmentación independientes para cada mano es importante para el problema que se explora en este trabajo, y por ello se ha diseñado una métrica que pueda cuantificar de forma imparcial la conformidad con este requisito. En este texto se referirá a esta métrica como la *Métrica de la similitud media entre centroides (SMC)*.

Suponiendo que al inferir en una imagen se obtiene una máscara de segmentación muy buena, que contenga un *blob* por cada instancia en la imagen, entonces estos *blobs* se podrán asociar uno a uno con los *blobs* en la máscara de segmentación objetivo[1] . Se busca una métrica que en este caso base mida la desviación media entre los centroides de los *blobs* inferidos y los centroides de los *blobs* objetivo. Esta métrica debe penalizar además los casos que se desvíen del caso base considerado por ser este el caso ideal, donde el modelo infiere un *blob* para cada instancia. Se debe penalizar tanto el caso en el que el número de *blobs* en ambas segmentaciones no coincida, como el caso en el que hay algún *blob* que no tenga una correspondencia directa en la otra segmentación[2]

En la métrica diseñada se analizan por separado los casos en los que se hayan obtenido el mismo número de máscaras individuales a las que hay en el objetivo o no. Esto permite más flexibilidad a la hora de interpretar cada caso. El algoritmo es simétrico, y por lo tanto no importa qué máscara es la inferida y qué máscara representa el *ground truth*.

Si hay el mismo número de *blobs* en ambas segmentaciones, estos se asocian en parejas de ta forma que se minimice la distancia total entre los elementos de las parejas. No asociar cada *blob* al más cercano en la otra segmentación permite evaluar mejor los casos en los que haya *blobs* sin correspondencia directa.

Si el número de *blobs* no coincide, se asocia para cada *blob* en la segmentación

[1] Suponiendo que cada instancia tiene un blob individual asociado. Lo que en este caso se ha asegurado con un postprocesado en el dataset como se explica en la sección §3.1

[2] Como ejemplo se propone el caso en el que haya un *blob* en cada segmentación pero su IoU sea 0.

con más cantidad, la distancia al *blob* más cercano en la otra segmentación. Y, para penalizar este caso, se calcula la distancia media como la suma de estas distancias dividida ente el número de *blobs* en la segmentación con menos. Esto permite penalizar menos un blob cuanto menos se aleja de de alguno en la otra segmentación, y penalizar en función de la disparidad en la cantidad de blobs obtenidos en cada segmentación.

Esta distancia media obtenida se normaliza respecto a la máxima separación posible, la diagonal de la imagen, para obtener un resultado entre 0 y 1 si hay el mismo número de máscaras en ambas segmentaciones, y mayor que cero si no hay el mismo número de máscaras. Finalmente se calcula la raíz cuadrada de esta medida para aumentar su magnitud, y se resta de 1 para obtener un valor menor que 1. La implementación en Python utilizada se muestra en la figura B.1

```python
import cv2
import numpy as np
from itertools import permutations

def avg_centroid_similarity(
                target_seg: np.array,
                inferred_seg: np.array
            ) -> float:
    """
    Inputs target_seg and inferred_seg as binary matrices
    with the same shape
    """
    assert(np.shape(target_seg) == np.shape(inferred_seg))

    # Get target centroids
    target_c = []
    target_seg = np.array(target_seg, dtype=np.uint8)
    contours, _ = cv2.findContours(
                        target_seg,
                        cv2.RETR_TREE,
                        cv2.CHAIN_APPROX_SIMPLE
                    )
    for c in contours:
        M = cv2.moments(c)
        if M["m00"] != 0: #Ignore easily filterable zero-sized blobs
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])
            target_c.append([cX, cY])

    # Get inferred centroids
    pred_c = []
    inferred_seg = np.array(inferred_seg, dtype=np.uint8)
    contours, _ = cv2.findContours(
                            inferred_seg,
                            cv2.RETR_TREE,
                            cv2.CHAIN_APPROX_SIMPLE
                    )
    for c in contours:
        M = cv2.moments(c)
        if M["m00"] != 0: #Ignore easily filterable zero-sized blobs
            cX = int(M["m10"] / M["m00"])
```

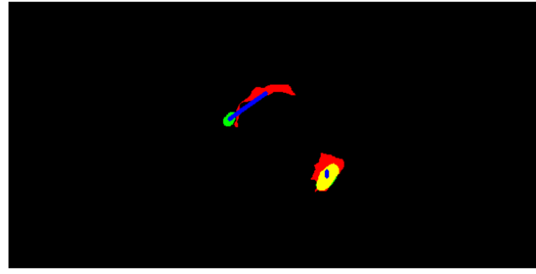
```
            cY = int(M["m01"] / M["m00"])
            pred_c.append([cX, cY])

    # Define shorter and longer centroid arrays
    s, l = sorted([target_c, pred_c], key=lambda x : len(x))

    # Handle edge cases
    if not s and l:
        return 0.
    elif not l and not s:
        return 1.

    # Handle same number of blobs
    if len(l) == len(s):
        n = len(l)
        dist_matrix = [[dist(p, q) for p in l] for q in s]
        perms = permutations([i for i in range(n)])
        dist_combinations = [
                    sum([
                        dist_matrix[j][i]
                        for j, i in enumerate(perm)
                    ])
                    for perm in perms
                ]
        total_dist = min(dist_combinations)

    # Handle different number of blobs
    else:
        total_dist = sum([
                    min([dist(p, q) for q in s])
                    for p in l
                ])

    diag = (
            np.shape(target_seg)[0]**2 +
            np.shape(target_seg)[1]**2
        )**.5

    return 1 - (total_dist/len(s)/diag)**.5
```

Listing B.1: Implementacición de la métrica de similitud media entre centroides utilizada.

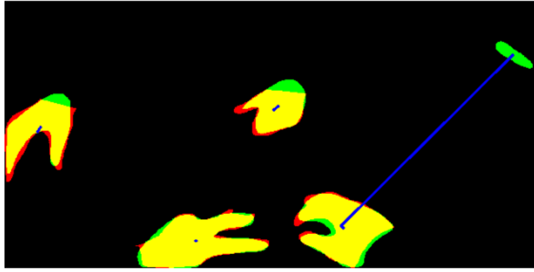
La complejidad computacional factorial de este algoritmo es asumible en esta aplicación por la cantidad reducida de *blobs* que aparecen en las imágenes. En aplicaciones diferentes, utilizar una aproximación en el caso de que haya el mismo número de *blobs* podría ser necesario o recomendable.

En la figura B.1 se muestra esta métrica calculada en varias segmentaciones obtenidas y sus respectivos *ground truths*. Un valor SMC de 1.0 corresponde con un resultado perfecto, y se debe tener en cuenta que en casos de segmentaciones excesivamente fragmentadas se pueden obtener valores negativos. Teniendo esto en cuenta, se puede decir sin mucho reparo que esta métrica compara estas segmentaciones como se busca. La imagen B.1a claramente muestra los mejores resultados; la imagen B.1b penaliza levemente que la máscara inferida para una instancia que apenas coincide con la predicción; la imagen B.1c penaliza de forma significativa

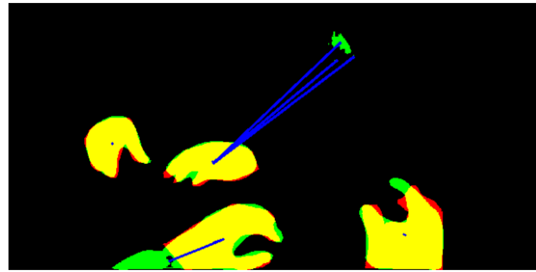
(a) Mismo número de blobs.
SMC: 0.9333



(b) Mismo número de blobs.
SMC: 0.7938



(c) Número diferente de blobs.
SMC: 0.6703



(d) Número diferente de blobs.
SMC: 0.5052

Figura B.1: Métrica calculada en cuatro segmentaciones obtenidas con distintos modelos. Se muestran las segmentaciones en rojo y verde, su intersección en amarillo, y las distancias consideradas en azul.

un *blob* predicho para lo que era un brazo en el fondo; por último, la imagen B.1d penaliza en gran medida la predicción de una máscara por estar fragmentada y por no tener correspondencia en el *ground truth*.