



UNIVERSIDAD
DE MÁLAGA



ESCUELA DE INGENIERÍAS INDUSTRIALES

TRABAJO FIN DE MÁSTER

Detección multimodal de objetos mediante deep-learning empleando la red AdapNet++ para fusión sensorial.

Multimodal object detection via deep-learning using AdapNet++ model for sensory fusion.

Máster en Ingeniería Industrial

Autor: Pablo Venema Rodríguez

Tutor: Dr. Ricardo Vázquez Martín

Cotutor: Andrés Manuel Salas Espinales

Departamento de Ingeniería de Sistemas y Automática

Áreas de Conocimiento: Ciencia de la Computación e Inteligencia Artificial

MÁLAGA, Septiembre de 2021

DECLARACIÓN DE ORIGINALIDAD DEL PROYECTO/TRABAJO FIN DE MASTER

D./ Dña.: Pablo Venema Rodríguez

DNI/Pasaporte: 77148745Y Correo electrónico: pablogenox@gmail.com

Titulación: Máster en Ingeniería Industrial

Título del Proyecto/Trabajo: Detección multimodal de objetos mediante deep-learning empleando la red AdapNet++ para fusión sensorial.

DECLARA BAJO SU RESPONSABILIDAD

Ser autor/a del texto entregado y que no ha sido presentado con anterioridad, ni total ni parcialmente, para superar materias previamente cursadas en esta u otras titulaciones de la Universidad de Málaga o cualquier otra institución de educación superior u otro tipo de fin.

Así mismo, declara no haber trasgredido ninguna norma universitaria con respecto al plagio ni a las leyes establecidas que protegen la propiedad intelectual, así como que las fuentes utilizadas han sido citadas adecuadamente.

En Málaga, a 8 de noviembre de 2021



Fdo.: Pablo Venema Rodríguez.

Resumen

Este trabajo fin de máster analiza la aplicación de técnicas de transferencia de conocimiento sobre una red neuronal artificial bimodal, utilizando para ello imágenes pertenecientes al espectro visible (RGB) e infrarrojo obtenidas del dataset UMA-SAR.

Para conseguir este objetivo se ha utilizado la arquitectura AdapNet++ con un bloque SSMA necesario para la fusión de diferentes modalidades. Inicialmente se ha utilizado uno de los modelos ya entrenados por los autores y se ha llevado a cabo un proceso de transferencia de aprendizaje añadiendo una clase adicional.

A continuación, se ha realizado un proceso de transferencia de conocimiento similar utilizando imágenes RGB y térmicas procedentes del dataset UMA-SAR. Al utilizar este conjunto de datos, se emplean imágenes dentro del ámbito de la búsqueda y rescate y se sustituye una de las modalidades originales del modelo Adapnet++ por otra modalidad, profundidad por térmica. Para realizar este proceso se ha re-entrenado una red AdapNet++ unimodal con imágenes RGB y, por otro lado, con imágenes térmicas partiendo de varios modelos proporcionados por los autores de la red.

Por último, se han empleado los mejores modelos para el re-entrenamiento de la red AdapNet++ bimodal. Los resultados de este proceso se muestran en base a métricas comúnmente utilizadas en aplicaciones destinadas a la segmentación de imágenes.

Palabras clave: Red neuronal artificial, red neuronal convolucional, convolución, AdapNet++, SSMA, segmentación, encoder, decoder, etiquetas, transferencia de conocimiento, imagen RGB, imagen térmica, mapa de profundidad, Cityscapes, UMA-SAR, bimodal, unimodal, MIoU.

Abstract

This work analizes the application of transfer learning techniques on a bimodal artificial neural network using visible (RGB) and infrared spectrum images obtained from UMA-SAR dataset.

For this purpose, AdapNet++ architecture with a SSMA block is used to fuse different modalities. Initially, a model trained by the autors is chosen and a transfer learning process is made by adding an aditional class to the original model.

After this, a similar transfer learning process is accomplished using RGB images and thermal images from UMA-SAR dataset. When using this datset, images from the serching and rescue ambit are used and one of the original modalities from AdapNet++ changes, thermal instead of depth. In order to achieve this, a transfer learning process of an unimodal AdapNet++ architecture with RGB and thermal images using models provided by the authors as starting point is carried out.

Finally, the best models have been used in the re-train of a bimodal AdapNet++. The results of this process are shown based on metrics commonly used in image segmentation applications.

Key words: Artifical neural network, convolutional neural network, convolution, AdapNet++, SSMA, segmentation, encoder, decoder, labels, transfer learning, RGB image, thermal image, depth map, Cityscapes, UMA-SAR, bimodal, unimodal, MIoU.

Índice general

1. Introducción	1
1.1. Motivación	3
1.2. Objetivos	4
1.3. Estructura de la memoria	5
1.4. Estado del arte	6
2. Materiales y métodos	11
2.1. Plan de trabajo	11
2.2. Recursos utilizados	12
2.2.1. Tensorflow 1	12
2.2.2. Datasets: Cityscapes, Freiburg forest y UMA-SAR.	15
2.2.3. Google colab	18
2.2.4. CVAT (Computer Vision Anotation Tool)	19
3. Principios teóricos	21
3.1. Unidad básica. La neurona	22
3.2. Proceso de entrenamiento	23

3.3.	Función de pérdida y backpropagation	24
3.4.	Infraajuste y sobreajuste	26
3.5.	Redes neuronales convolucionales (CNN)	28
3.5.1.	Hiperparámetros de una capa convolucional	30
3.6.	Capas de pooling	31
3.7.	Campo receptivo efectivo	32
3.8.	Mejoras en el proceso de entrenamiento de una red neuronal artificial	33
3.9.	Regularización del error	34
3.9.1.	Regularizadores L1 y L2	35
3.10.	Dropout	35
3.11.	Normalización por lotes (Batch normalization)	36
3.12.	Avances adicionales para mejorar el rendimiento de una red neuronal	36
3.13.	Uso de CNN en segmentación de imágenes	38
3.13.1.	Arquitecturas encoder-decoder	40
3.14.	Transferencia de conocimiento, ajuste fino y aumento de datos	43
4.	AdapNet++	45
4.1.	Estructura general de AdapNet++	46
4.1.1.	Módulo eASPP	48
4.1.2.	Decoder de AdapNet++	49
4.1.3.	Poda de neuronas	49
4.2.	Bloque SSMA	50

ÍNDICE GENERAL	V
5. Transferencia de conocimiento con AdapNet++	53
5.1. Metodología	53
5.2. Transferencia de conocimiento utilizando el dataset de Cityscapes	54
5.2.1. Etiquetado de las imágenes del dataset de Cityscapes	55
5.2.2. Proceso de re-entrenamiento sobre el dataset de Cityscapes	60
5.3. Transferencia de conocimiento utilizando el dataset UMA-SAR	63
5.3.1. Etiquetado y tratamiento de las imágenes del dataset UMA-SAR	63
5.3.2. Experimentos previos al re-entrenamiento	71
6. Resultados	77
6.1. Métricas	77
6.2. Resultados obtenidos sobre el dataset de Cityscapes	78
6.3. Resultados obtenidos sobre el dataset UMA-SAR	85
6.3.1. Resultados obtenidos en imágenes RGB partiendo del modelo de Cityscapes RGB	86
6.4. Resultados sobre imágenes térmicas utilizando el modelo de cityscapes	89
6.5. Resultados obtenidos utilizando el modelo general de AdapNet ++	89
6.6. Consideración del fondo en el cálculo del error	91
6.7. Índices obtenidos con las diferentes estrategias de entrenamiento	94
7. Conclusiones y trabajos futuros	105

VI

ÍNDICE GENERAL

Bibliografía

108

Índice de figuras

1.1.	Evolución de las técnicas de visión por computador. Fuente: Goodfellow y col., 2016	2
1.2.	Frecuencia de aparición de la expresión neural network (red neuronal) además de otros nombres que le han dado distintos autores. Fuente: Goodfellow y col., 2016	2
1.3.	Comparación entre imagen RGB (a) y térmica (b). (Fuente: Elaboración propia). Imágenes obtenidas de Morales y col., 2021	4
1.4.	Diferentes arquitecturas de FCNs. (Fuente: Planche y Andres, 2019)	7
1.5.	Arquitectura de U-Net. (Fuente: Planche y Andres, 2019) . . .	7
2.1.	Se importan los archivos a partir de los cuales se construye la red. (Fuente: Elaboración propia).	15
2.2.	Creación de la red, grafo y placeholder del tipo de entrada. (Fuente: Elaboración propia).	15
2.3.	Carga de los valores del checkpoint en la red construida. (Fuente: Elaboración propia).	15
2.4.	Ejemplos de imágenes tomadas del dataset de Cityscapes. RGB (a). Disparity (b). Máscara (c). Fuente Cordts y col., 2016 . .	17
2.5.	Ejemplo de imágenes del dataset de las jornadas de rescate de la UMA. RGB (a). Térmica (b).	18

3.1.	Representación de una neurona en IA. (Fuente: (Planche y Andres, 2019))	22
3.2.	Funciones de activación. Sigmoide (1). Tangente hiperbólica (2). ReLU (3). (Fuente: University, s.f.)	23
3.3.	Representación de una red neuronal. Se representan las entradas de la red como X y las salidas como Y (Fuente: Elaboración propia)	24
3.4.	Ilustración del proceso de descenso de gradiente. (Fuente: Planche y Andres, 2019)	26
3.5.	Ejemplo de sobreajuste. (Fuente: https://towardsdatascience.com/dont-overfit-ii-how-to-avoid-overfitting-in-your-machine-learning-and-deep-learning-models-2ff903f4b36a)	27
3.6.	Ejemplo de infraajuste (Fuente: Goodfellow y col., 2016)	28
3.7.	Operación de convolución. (Fuente: Elaboración propia)	29
3.8.	Aplicación de la máscara de Sobel. Imagen original a la izquierda y resultado a la derecha. (Fuente: https://towardsdatascience.com/magic-of-the-sobel-operator-bbbcb15af20d)	29
3.9.	Deslizamiento de la máscara con un stride de 2. (Fuente: Elaboración propia)	31
3.10.	Orlado de una matriz. (Fuente: Elaboración propia)	31
3.11.	Capa max-pooling con k y stride de 3. (Fuente: Elaboración propia)	32
3.12.	Campo receptivo de una capa tras la aplicación de dos máscaras 3x3. (Fuente: Elaboración propia)	32
3.13.	Ejemplo de imagen tratada por la red YOLO. (Fuente: https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/)	38
3.14.	Ejemplo de segmentación de una imagen. (Fuente: Elaboración propia)	39

ÍNDICE DE FIGURAS

IX

3.15. Convolución traspuesta. (Fuente: (Planche y Andres, 2019))	41
3.16. Ejemplo de unpooling. (Fuente: (Planche y Andres, 2019))	41
4.1. Arquitectura de la red AdapNet++. (Fuente: (Valada y col., 2019)	47
4.2. Ejemplo de bloque residual. (Fuente: Elaboración propia)	47
4.3. Estructura del decoder al introducir las pérdidas auxiliares. (Fuente: Elaboración propia)	49
4.4. Arquitectura del encoder para segmentación multimodal. (Fuen- te: (Valada y col., 2019)	50
4.5. Arquitectura del bloque SSMA. (Fuente: (Valada y col., 2019)	51
5.1. Metodología seguida durante el proceso de transferencia de conocimiento	54
5.2. Ejemplo de aplicación de aumento de datos. a) Imagen origi- nal. b) Scale. c) Flip horizontal. d) Crop	57
5.3. Ejemplos de mapas de profundidad en formato disparity (iz- quierda) y JET (derecha). (Fuente: Elaboración propia).	59
5.4. Comparación entre dos mapas de profundidad en formato JET. Sin aplicar la técnica de relleno(izquierda). Aplicando la técnica de relleno(derecha). (Fuente: Elaboración propia)	59
5.5. Fragmento de código utilizado para no cargar todas las capas. (Fuente: Elaboración propia).	62
5.6. Ejemplo de imagen etiquetada con CVAT. (Fuente: Elabora- ción propia).	65
5.7. Creación de una tarea en CVAT. (Fuente: Elaboración propia)	67
5.8. Interfaz de CVAT. (Fuente: Elaboración propia).	67

5.9.	Botón para cargar modelos en CVAT. (Fuente: Elaboración propia).	68
5.10.	Botón para dibujar polígonos en CVAT. (Fuente: Elaboración propia).	68
5.11.	Misma imagen en CVAT. Sin usar tecla control (a). Usando tecla control (b).	69
5.12.	Etiquetado rápido. (Fuente: Elaboración propia)	69
5.13.	Estructura del archivo a importar en CVAT. (Fuente: Elaboración propia).	70
5.14.	Predicciones efectuadas por AdapNet++ cargada con el modelo original de Cityscapes sobre imágenes RGB del dataset UMA-SAR. (Fuente: Elaboración propia).	73
5.15.	Predicciones efectuadas por AdapNet++ cargada con el modelo original de Freiburg-forest sobre imágenes RGB del dataset UMA-SAR. (Fuente: Elaboración propia)	74
5.16.	Predicciones efectuadas por AdapNet++ cargada con el modelo original de Cityscapes sobre imágenes térmicas del dataset UMA-SAR(Fuente:Elaboración propia)	75
5.17.	Predicciones efectuadas por AdapNet++ cargada con el modelo original de Freiburg sobre imágenes térmicas del dataset UMA-SAR. (Fuente: Elaboración propia)	76
6.1.	Evolución del valor de pérdida de la red durante el proceso de entrenamiento. Cada punto del eje horizontal corresponde a 10 iteraciones de entrenamiento. (Fuente: Elaboración propia).	79
6.2.	Evolución del valor del MIoU sobre el conjunto de evaluación para los pesos de la red calculados en distintas etapas del entrenamiento. (Fuente: Elaboración propia).	79

6.3. Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 calses checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia)	80
6.4. Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia)	81
6.5. Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia)	81
6.6. Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia)	82
6.7. Resultados tras aplicar ajuste fino sobre el chekpoint 409.	85
6.8. . Resultados obtenidos tras realizar un reentreno sobre el modelo de Cityscapes con las imágenes del dataset UMA-SAR. (Fuente: Elaboración propia)	88
6.9. Predicciones de AdapNet++ para el modelo de Cityscapes RGB utilizando imágenes en escala de grises. (Fuente: Elaboración propia)	90

6.10. Evolución del error. a) Reentrenamiento a partir del modelo de Cityscapes rgb sin tener en cuenta el fondo de la imagen. b) Reentrenamiento a partir del modelo de Cityscapes RGB c) Reentrenamiento a partir del modelo de Cityscapes de mapas de profundidad con imágenes térmicas. d) Reentrenamiento a partir del modelo de propósito general con imágenes térmicas. e) Reentrenamiento a partir del modelo de Cityscapes RGB con imágenes térmicas. f) Reentreno a partir del modelo de propósito general con imágenes RGB.	93
6.11. Valores del IoU de cada clase al entrenar con imágenes RGB sobre el modelo de Cityscapes RGB teniendo en cuenta el fondo en el cálculo del error.	98
6.12. Valores del IoU de cada clase al entrenar con imágenes RGB sobre modelo de AdapNet++ general teniendo en cuenta el fondo en el cálculo del error	98
6.13. Valores del IoU de cada clase al entrenar con imágenes térmicas sobre modelo de Cityscapes RGB teniendo en cuenta el fondo en el cálculo del error.	99
6.14. Valores del IoU de cada clase al entrenar con imágenes térmicas sobre el modelo de Cityscapes para mapas de profundidad teniendo en cuenta el fondo en el cálculo del error.	99
6.15. Valores del IoU de cada clase al entrenar con imágenes térmicas sobre el modelo de AdapNet++ general teniendo en cuenta el fondo n el cálculo del error.	100
6.16. Resultados obtenidos con los diferentes modelos si se tiene en cuenta el fondo al calcular el error.	102
6.17. Carga de los bloques de una red AdapNet++ bimodal. (Fuente: Elaboración propia).	103

Índice de tablas

6.1.	Resultados obtenidos para imágenes RGB sin tener en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia) .	94
6.2.	Resultados obtenidos para imágenes térmicas sin tener en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)	95
6.3.	Resultados obtenidos con imágenes RGB teniendo en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia) .	95
6.4.	Resultados obtenidos con imágenes térmicas teniendo en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)	95
6.5.	Resultados desglosados por clase obtenidos con imágenes RGB utilizando el modelo de Cityscapes teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)	95
6.6.	. Resultados desglosados por clase obtenidos con imágenes rgb utilizando el modelo de AdapNet++ general teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)	96
6.7.	Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de Cityscapes RGB teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)	96

6.8.	Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de Cityscapes para mapas de profundidad teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)	97
6.9.	Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de AdapNet++ general teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)	97
6.10.	Resultados obtenidos para AdapNet++ bimodal.	104

Capítulo 1

Introducción

A finales del siglo XX vio la luz uno de los mayores avances en el campo del aprendizaje automático (machine learnig), la aparición del aprendizaje profundo (Deep-learning) y de las redes neuronales artificiales. Previo a este avance, las técnicas de aprendizaje automático utilizadas en visión por computador consistían en que, un operador humano, decidía cuales eran los patrones más importantes a detectar en una imagen y desarrollaba una aplicación que fuese capaz de detectar dichos patrones. La principal ventaja que ofrecen las redes neuronales artificiales es la gran cantidad de parámetros de la que disponen y que se ajustan durante el entrenamiento de dichas redes, lo cual le permite superar el desempeño de las aplicaciones de aprendizaje automático clásico. La evolución de las técnicas de aprendizaje profundo (machine learning) puede observarse en la figura 1.1.

Pese a que la primera aparición de las redes neuronales artificiales se remonta a 1940 no ha sido hasta la última década cuando su uso ha comenzado a crecer de manera importante. Esto se debe a que la gran cantidad de parámetros de los que disponen hace necesario, por un lado, una gran cantidad de ejemplo para entrenar dichas redes, y por otro, a la potencia de computo que hace falta para realizar dicho entrenamiento. En el momento de su primera aparición (1940), estos dos obstáculos no eran salvables, sin embargo, la aparición de internet y los avances en hardware de los ordenadores han provocado que las redes neuronales artificiales se posicione como uno de los protagonistas en técnicas de visión por computador (figura 1.2).

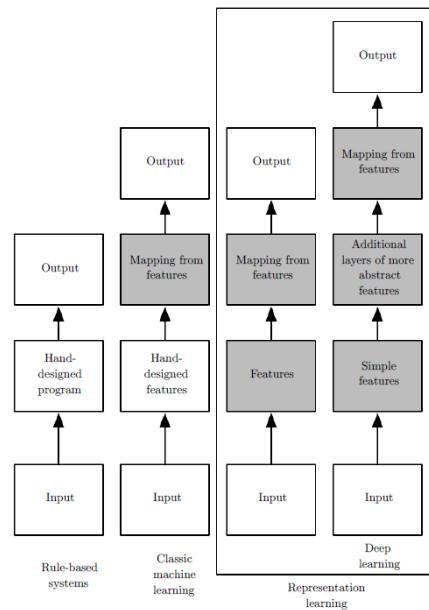


Figura 1.1: Evolución de las técnicas de visión por computador. Fuente: Goodfellow y col., 2016

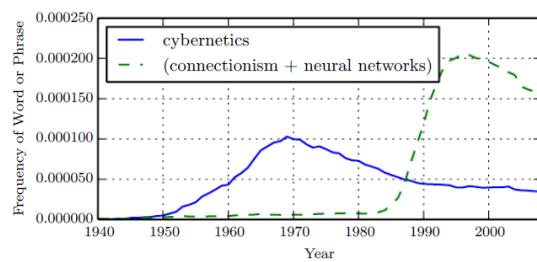


Figura 1.2: Frecuencia de aparición de la expresión neural network (red neuronal) además de otros nombres que le han dado distintos autores. Fuente: Goodfellow y col., 2016

Las redes neuronales artificiales pueden usarse para una gran variedad de aplicaciones, una de ellas y la que es objeto de este trabajo es la segmentación. En este ámbito, las redes neuronales convoluciones, comúnmente llamadas CNN(Convolutional Neural Network) y las DCNN(Deep Convolutional Neural Network) han demostrado unos grandes resultados en aplicaciones específicas. Por ejemplo en conducción autónoma de vehículos, los resultados obtenidos en detección de peatones son excelentes Sermanet y col., 2012 e incluso llegan a superar la capacidad humana en cuanto a clasificación de señales de tráfico se refiere Ciresan y col., 2012. Esta precisión hace que las redes neuronales artificiales se usen cada vez más en aplicaciones que requieren de una rápida y correcta identificación del entorno.

Una de las últimas contribuciones a la segmentación de imágenes y las DCNN es la arquitectura AdapNet++ (Valada y col., 2019) que es la que se ha estudiado y con la que se ha trabajado en este proyecto. La principal contribución que aporta esta red a la segmentación de imágenes es la fusión de características obtenidas de dos formatos de imágenes distintos, lo cual permite que la red arroje predicciones correctas cuando uno de los canales de entrada no proporciona información relevante, o mejorar la predicción resultante de la utilización de un solo canal gracias a la información adicional que proporcionan el resto. Esto permite a AdapNet++ arrojar una predicción correcta en situaciones en las que una red neuronal artificial que trabaja con una sola modalidad como entrada falla.

1.1. Motivación

Cada año, la Universidad de Málaga organiza unas jornadas de rescate en las que se cuenta con escenarios tales como escenarios de catástrofe natural o atentado. Durante estas jornadas se cuenta con la participación de diferentes equipos de rescate, como por ejemplo la UME. Una de las labores llevadas a cabo es la de tomar varias imágenes de los entornos preparados en dos formatos, espectro infrarrojo (imágenes térmicas) y espectro visible (color o RGB) toda la información referente a este dataset puede consultarse en (Morales y col., 2021).

En estas situaciones de emergencia, la identificación rápida de víctimas

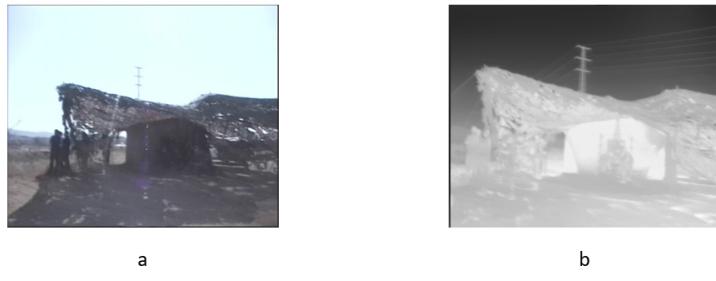


Figura 1.3: Comparación entre imagen RGB (a) y térmica (b). (Fuente: Elaboración propia). Imágenes obtenidas de Morales y col., 2021

y de personal de rescate es crucial, lo que hace que la utilización de una red neuronal artificial sea muy conveniente ya que pude identificar de manera temprana y con precisión a estas dos categorías. Esto permite enviar la ayuda necesaria al lugar adecuado lo antes posible.

El principal motivo por el que se utiliza AdapNet++ reside en su capacidad para extraer información de varios canales, en este caso, un canal corresponde a imágenes en espectro visible (RGB) y otro a imágenes del espectro infrarrojo (térmicas). La combinación de estas dos modalidades resulta muy interesante ya que el uso de las imágenes RGB puede resultar contraproducente cuando, por ejemplo, se tienen escenarios con una mala iluminación. Sin embargo, estas situaciones no afectan a las imágenes térmicas. En la figura 1.3 puede comprobarse como personas que no se ven en la imagen RGB por falta de iluminación sí que son reconocibles en la imagen térmica, esta situación puede ser aún más evidente en situaciones en las que es de noche o por ejemplo al atravesar un túnel que no está iluminado.

1.2. Objetivos

El objetivo principal de este trabajo consiste en analizar el funcionamiento de la red AdapNet++ al aplicar técnicas de transferencia de conocimiento utilizando para ello imágenes del dataset UMA-SAR. Para conseguir este objetivo principal se establecen una serie de objetivos específicos.

1. Evaluar el funcionamiento del modelo AdapNet++ de mapas profundidad utilizando como entrada imágenes térmicas. Con esto se pretende comprobar si la información que se extrae de un mapa de profundidad puede resultar útil en la segmentación de imágenes térmicas. Los mapas de profundidad representan una estimación de la distancia a la que se encuentra un elemento a partir de las imágenes tomadas por dos cámaras que están separadas una cierta distancia. La estimación realizada se proyectará en una imagen en escala de grises, asociando al elemento en cuestión un nivel de intensidad concreto. Por ejemplo, un nivel de gris muy alto para elementos cercanos y muy bajo para los que se sitúen más lejos.
2. Etiquetar un número reducido de imágenes del dataset UMA-SAR (Morales y col., 2021) y aplicarles técnicas de aumento de datos.
3. Aplicar técnicas de transferencia de conocimiento sobre uno de los datasets con los cuales los autores de AdapNet++ han entrenado la red.

1.3. Estructura de la memoria

La memoria se divide en 7 capítulos que se explicarán brevemente en este apartado.

1. Materiales y métodos: En este capítulo se explica el plan de trabajo que se ha seguido además de las herramientas utilizadas durante la elaboración del mismo.
2. Principios teóricos: Aquí se explican brevemente los principios teóricos que ha sido necesario estudiar antes de comenzar a realizar el trabajo.
3. AdapNet++: Se presenta a estructura de la red neuronal artificial utilizada en un capítulo separado de los principios teóricos más generales para indicar las contribuciones principales de la arquitectura AdapNet++.
4. Transferencia de conocimiento con AdapNet++: Se explicarán los pasos seguidos para etiquetar las imágenes utilizadas durante el proceso de

entrenamiento así como experimentos previos a dicho proceso y cómo se ha llevado a cabo el proceso de re-entrenamiento.

5. Resultados: En este capítulo se presentan los resultados obtenidos con las diferentes estrategias de re-entrenamiento.
6. Conclusiones y trabajos futuros

1.4. Estado del arte

El rápido desarrollo de las redes neuronales ha llevado a que estas se utilicen en un gran número de aplicaciones, la más importantes se explican en el capítulo 3. Este capítulo se centrará en aquellas redes utilizadas para segmentación de imágenes.

Las redes utilizadas en segmentación de imágenes suelen utilizar arquitecturas encoder-decoder, como se explica en el capítulo 5. Una de las primeras redes desarrolladas con esta morfología y que se sigue utilizando es la familia FCN (Long y col., 2014) representada en la figura 1.4. Basan su arquitectura en VGG-16 (Simonyan y Zisserman, 2015) con ligeras modificaciones además de añadir un decoder. En esta arquitectura se fusionan mapas de características de las capas finales con capas situadas en la parte media de la red. Existen diferentes versiones de la misma dependiendo de que capas se utilicen para realizar esta fusión. La versatilidad de esta red hace que continúe usándose para diferentes aplicaciones, así como base para el desarrollo de nuevas arquitecturas.

Una de las primeras arquitecturas desarrolladas a partir de FCNs fue U-Net, (Ronneberger y col., 2015). Al igual que las FCNs, cuenta con un encoder y fusión de características detectadas en diferentes capas de la red. Las principales diferencias son, por un lado, su arquitectura en forma de U (1.5), y por otro, que la fusión de características no se hace mediante una suma elemento a elemento de los mapas de características sino concatenando (se “apilan” unos mapas sobre otros). Al igual que con las FCNs, esta arquitectura ha inspirado muchas otras.

Otra red ampliamente utilizada en segmentación es Mask R-CNN (He y col., 2018), basada en Faster R-CNN (Ren y col., 2016). La segunda red

1.4 Estado del arte

7

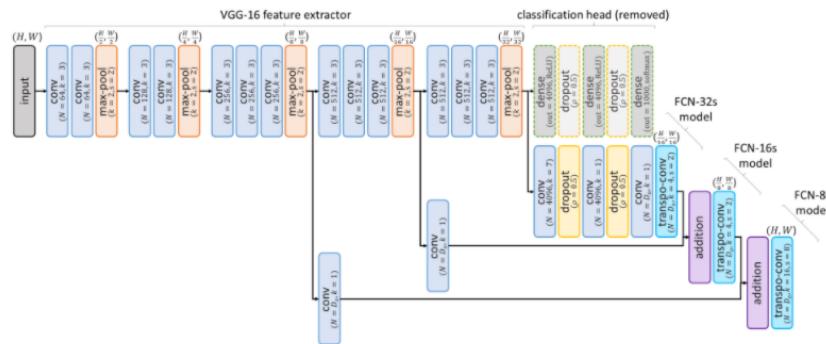


Figura 1.4: Diferentes arquitecturas de FCNs. (Fuente: Planche y Andres, 2019)

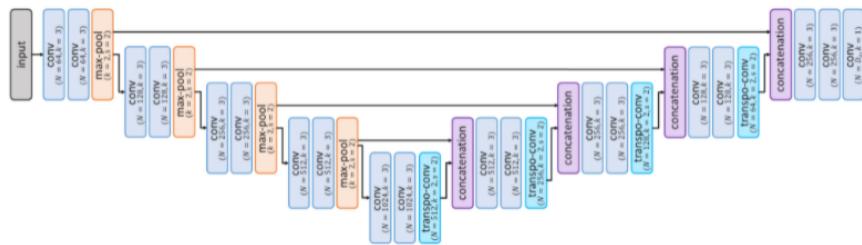


Figura 1.5: Arquitectura de U-Net. (Fuente: Planche y Andres, 2019)

es utilizada para detección de objetos (capítulo 3), de modo que dibuja una caja alrededor de cada objeto detectado en una imagen. Mask R-CNN toma esto y segmenta el objeto dentro de la caja. Aquí aparece un nuevo concepto, la segmentación de instancias, y es que esta red no solo detecta que hay en una imagen sino también su cantidad. En las redes anteriores, un grupo de personas era clasificado como persona, pero no se sabía cuántas había. Mask R-CNN al separar previamente cada una en una caja, puede determinar su número.

Volviendo a la segmentación semántica (no se sabe el número de elementos), las redes convolucionales (capítulo 3) han demostrado un desempeño más que satisfactorio y por ello son las más utilizadas en este campo, de hecho, U-Net y la familia FCN son redes convolucionales. A partir de las FCNs surge DeconvNet (Noh y col., 2015), que mejora los resultados de su antecesor gracias a un up-sampling (proceso en el que se recupera el tamaño de la imagen original) en el que se utiliza una red de deconvolución profunda pero haciendo el proceso de entrenamiento más complejo.

Con SegNet (Badrinarayanan y col., 2016) se elimina la necesidad de aprender el proceso de up-sampling ya que se reutilizan los índices de las capas del encoder. ParseNet (Liu y col., 2015) basa sus predicciones en el contexto global de la imagen en lugar de hacerlo a partir de un campo receptivo muy amplio. De esta manera se pretenden evitar predicciones erróneas en determinadas zonas de la imagen. Dicho esto, sigue habiendo problemas de resolución.

Para solucionar esto se introducen las convoluciones dilatadas y diferentes tipos de capas de pooling, explicadas en el capítulo 3, surgiendo DeepLab (Chen y col., 2017) y PSPNet (Zhao y col., 2017).

Todas las redes mencionadas se basan en segmentar un solo tipo de imagen, pero una de las peculiaridades de AdapNet++ con el bloque SSMA (Valada y col., 2019), es que utiliza lo que se conoce como fusión multimodal, parte de imágenes en más de un formato para obtener información. En la mayoría de casos, se utilizan dos canales de extracción de información distintos, alguno de ellos puede incluso utilizar técnicas de machine learning en lugar de ser una red neuronal. Cada uno extraerá unas características determinadas y posteriormente se combinarán para arrojar una predicción.

Las técnicas de fusión entre salidas de más de un tipo de imagen se pueden dividir en fusión temprana, jerárquica y tardía. La primera de ellas consiste en agrupar varias modalidades en una sola entrada y pasar esto a la red, sin embargo, esto no consigue que la red aprenda a diferenciar características de cada modalidad. La fusión jerárquica extrae características de varios encoders en diferentes etapas de los mismos y las junta en un solo decoder.

Al emplear fusión tardía, primero se entrena cada encoder en una modalidad de datos de entrada y posteriormente se combinan los mapas de características de cada una. En este contexto surge el bloque CMoDE (Valada y col., 2017), desarrollado por los autores de AdapNet++ y que no solo permite fusionar mapas de características de diferentes modalidades, sino que además consigue determinar cuál de las modalidades aporta la información más relevante. Este bloque evoluciona al bloque SSMA, introducido por los autores de AdapNet++ en la red utilizada en este trabajo.

Capítulo 2

Materiales y métodos

2.1. Plan de trabajo

Para realizar el trabajo, se han fijado diferentes fases, las cuales se han ido completando sucesivamente. En total, se cuenta con 4 fases.

1. Fase de documentación: En esta primera etapa se ha estudiado los principios teóricos de las redes neuronales. Además de esto, se ha seguido el curso de Google sobre Tensorflow y se han utilizado otros recursos como fuente de información adicional. El objetivo principal de esta fase es comprender los principios teóricos, así como programar redes neuronales artificiales sencillas.
2. Estudio de la red AdapNet++: Utilizando los artículos acerca de los bloques SSMA (Valada y col., 2019) y la propia AdapNet, en el que también se habla del bloque CMoDE (Valada y col., 2017).
3. Funcionamiento de la red AdapNet++.
 - Poner en funcionamiento un modelo de AdapNet++ y comprobar los resultados obtenidos.
 - Re-entrenar uno de los modelos de AdapNet++ introduciendo una nueva clase y aplicando técnicas de aumento de datos y de transferencia de conocimiento.

4. Estudio del funcionamiento de la red con imágenes procedentes del dataset UMA-SAR (Morales y col., 2021).
 - Etiquetado de las imágenes del dataset.
 - Comparación del funcionamiento al utilizar imágenes térmicas en lugar de mapas de profundidad.
 - Re-entrenamiento de la red con el dataset UMA-SAR.

2.2. Recursos utilizados

En este apartado se presentarán los programas utilizados para llevar a cabo las fases descritas.

2.2.1. Tensorflow 1

Tensorflow es una herramienta desarrollada por Google y que actualmente es muy utilizada en aplicaciones de aprendizaje profundo (Deep-learning), para lo cual ofrece una gran variedad de funciones además de la posibilidad de modelarlas para adaptarlas a propósitos más específicos. De esta manera, Tensorflow permite el desarrollo desde aplicaciones simples hasta las más complejas.

Tensorflow cuenta con varios niveles de abstracción, el primero de ellos es la capa C++. Esto es porque la mayor parte de las operaciones de cómputo se hacen en este lenguaje. Para que sea posible utilizar la GPU del ordenador a la hora de entrenar, evaluar o utilizar redes neuronales, NVIDIA desarrolló la librería CUDA. Esta librería permite ejecutar las aplicaciones desarrolladas con Tensorflow en la GPU.

El siguiente nivel es la capa API de bajo nivel. En este caso la capa permite “traducir” a C++ las aplicaciones desarrolladas en Python, que será el lenguaje que se utilizará en este trabajo.

Por último, se tiene la capa API de alto nivel, compuesta por Keras, que no se utiliza en este trabajo, y otra herramienta que cuenta con métodos

ampliamente utilizados ya implementados, de modo que se facilita su uso pues solo hay que invocarlos a través de comandos.

Una de las principales características de Tensorflow es que trabaja con tensores, de ahí su nombre, que pueden ser vectores o matrices N-dimensionales. Estas matrices se utilizan para guardar valores matemáticos de las distintas variables. Cada tensor estará compuesto por tres valores, si uno de ellos no está definido, podrá tomar cualquier cantidad, los valores son:

- Tipo de variable.
- Forma del vector o matriz.
- Rango, el número de dimensiones.

Otro de los elementos importantes de Tensorflow es la utilización de grafos. Un grafo es una manera de representar una o varias operaciones, de este modo un conjunto de operaciones iguales tendrá el mismo grafo, aunque varíen sus parámetros. Esto permite guardar, por ejemplo, la arquitectura de una red neuronal artificial en un grafo que contendrá las distintas capas que la forman y como estas interactúan. Los grafos permiten dividir las operaciones entre CPU y GPU, ejecutar diferentes partes de la red en máquinas distintas, mejorar la velocidad de respuesta de la red evitando operaciones innecesarias y, como cada grafo puede ejecutarse en cualquier dispositivo, facilitan la portabilidad del modelo.

Cada vez que se realice una operación, esta se guardará en un tensor, aunque este tensor no contendrá el resultado, solo información referente al tipo de operación (suma, multiplicación...), forma del vector de salida y tipo de dato. Para conocer el resultado es necesario iniciar una sesión. La sesión permite que la aplicación haga uso de los recursos disponibles. Por ejemplo, la operación $c = a + b$, no guardará el resultado de la suma, sino la operación suma, la forma de c y si es entero, float... El resultado de c se podrá ver con el comando `tf.Session.run(c)`. Si c es la red neuronal y a y b las entradas, puede ir variándose el valor de a y b y llamando al comando `tf.Session.run()` para obtener las distintas predicciones. Para conseguir esto se definen a y b como marcadores de posición (placeholders) y se siguen los siguientes pasos.

1. Se definen a y b con placeholders.

$$a = tf.placeholder(dtype = tf.int32, shape(None,))$$

Esto crea un vector de int32 con longitud indefinida, pero no especifica el valor de sus componentes.

2. Se define la operación $c = a + b$. Esto creará un grafo para c cuyas entradas a y b tendrán las características especificadas en sus placeholders.
3. Se crea la sesión y se pasan las variables mediante el comando feed_dict:

$$\text{Resultado} = \text{Session.run}(c, \text{feed_dict} = [a1, a2, a3,], [b1, b2, b3])$$

Se realizará la suma de a y b con los valores especificados en feed_dict. De esta manera, puede hacerse un bucle en el que a y b tomen los valores de todas las entradas disponibles y calcular c en cada caso, si se está entrenando una red neuronal, c podrían ser las pérdidas.

Otro componente muy importante de Tensorflow 1 son los checkpoints. Un checkpoint es un archivo en el cual se guardan los parámetros de una red que ha sido entrenada y que permite volver a cargarlos para continuar dicho entrenamiento o realizar predicciones una vez finalizado. Está formado a su vez por otros tres archivos.

- Archivo data: Contienen el valor de los parámetros de la red.
- Archivo meta: Contiene el grafo de la red.
- Archivo index.

De esta manera se crean tres archivos, si el nombre del checkpoint es ckpt_model, los archivos serán ckpt_model.data_00000_00001, ckpt_model.meta y ckpt_model.index. Cuando quiera utilizarse esta red ya entrenada, primero habrá que crear su grafo, en este paso es necesario que las capas del nuevo grafo creado tengan los mismos nombres que las capas contenidas en el checkpoint. Con el grafo creado, se especifica que variables se quieren cargar, y posteriormente se carga el checkpoint en la sesión. En este último paso

se carga el archivo ckpt_model, para que así se asocien los 3 archivos antes mencionados.

En las figuras 2.1, 2.2 y 2.3 puede verse el proceso seguido para cargar una red entrenada en Tensorflow 1. En este caso ‘models.’ (figura 6) es la carpeta donde se encuentra el archivo que construye la red y AdapNet_pp el nombre del archivo. En la figura 2.3, import_variables es una lista con las variables del grafo creado y path la ruta hacia el archivo que contiene el checkpoint. Con estos dos comandos se indica que se quieren cargar las variables del checkpoint que se encuentren dentro de la lista import_variables. Es importante que el grafo tenga los mismos nombres que el checkpoint. Si se trata de cargar una capa cuyo nombre no aparece en el grafo, Tensorflow arrojará un error.

```
module = importlib.import_module('models.' + 'AdapNet_pp')
model_func = getattr(module, 'AdapNet_pp')
```

Figura 2.1: Se importan los archivos a partir de los cuales se construye la red. (Fuente: Elaboración propia).

```
model = model_func(num_classes=8, training=False)
images_pl = tf.placeholder(tf.float32, [None, 384, 768, 3])
model.build_graph(images_pl)
```

Figura 2.2: Creación de la red, grafo y placeholder del tipo de entrada. (Fuente: Elaboración propia).

```
saver = tf.train.Saver(import_variables)
saver.restore(sess, path)
```

Figura 2.3: Carga de los valores del checkpoint en la red construida. (Fuente: Elaboración propia).

2.2.2. Datasets: Cityscapes, Freiburg forest y UMA-SAR.

En este apartado se explicarán brevemente los datasets utilizados en este trabajo.

Cityscapes

Cityscapes es un dataset que puede descargarse de manera gratuita de la página oficial de sus autores¹. Este dataset está formado por imágenes de diferentes ciudades de Alemania, y puede encontrarse en formato RGB, disparidad (mapa de profundidad) o vídeo entre otros.

Para este trabajo se han utilizado los archivos leftImg8bit, correspondiente a las imágenes RGB, disparity_trainvaltest, correspondiente a los mapas de profundidad y gtFine, que contiene los archivos necesarios para generar las etiquetas de las imágenes, el proceso de etiquetado de Cityscapes se explica en el capítulo 5. En total, este dataset cuenta con más de cinco mil imágenes en varios formatos y sus etiquetas, además de un repositorio de github con varias herramientas para manipular el dataset. Una de las características a tener en cuenta antes de utilizar estas imágenes en cualquier proceso de transferencia de conocimiento es que tienen una resolución 2048x1024. Ejemplos de imágenes extraídas del dataset se presentan en la figura 2.4.

Todas estas características hacen que sea un dataset ampliamente utilizado en aplicaciones de segmentación, sobre todo en lo que respecta a benchmarking, y para entrenar redes destinadas a conducción autónoma. La gran cantidad de imágenes que lo forman contienen una gran variedad de contextos distintos en lo que se refiere a condiciones meteorológicas, estaciones e iluminación.

Cabe destacar que en la figura 2.4, la imagen c no es la imagen etiquetada que se utiliza para entrenar la red. La imagen etiquetada contiene valores entre 0 y N, siendo N el número de clases por lo que sería muy difícil para una persona diferenciar los distintos objetos, ya que todos ellos tienen un tono de gris muy oscuro. Por este motivo se presenta la imagen coloreada.

Freiburg-forest

Este dataset ha sido realizado en la misma universidad en la que trabajan los autores de AdapNet++, para su realización han montado una cámara sobre un robot y han tomado imágenes RGB y mapas de profundidad del bosque con el mismo nombre (Valada y col., 2016).

¹<https://www.cityscapes-dataset.com/>

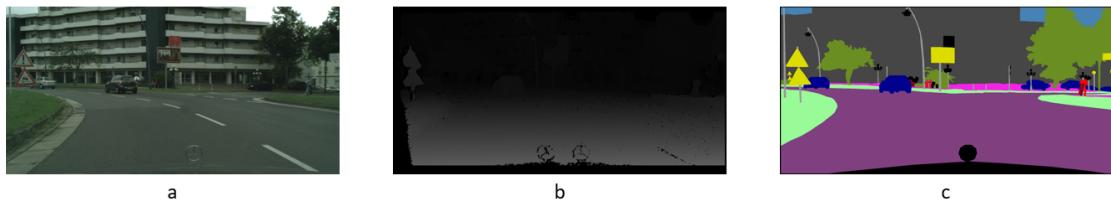


Figura 2.4: Ejemplos de imágenes tomadas del dataset de Cityscapes. RGB (a). Disparity (b). Máscara (c). Fuente Cordts y col., 2016

En este trabajo, este dataset solo se ha utilizado en la última etapa del trabajo para comprobar el funcionamiento de un modelo de AdapNet++ entrenado con el mismo, utilizando como entrada las imágenes del dataset UMA-SAR.

UMA-SAR

Cada año la Universidad de Málaga organiza unas jornadas de rescate en colaboración con otras organizaciones como la UME. En estas jornadas se prueban varios dispositivos desarrollados por la universidad para ayudar en labores de rescate tales como robots. Una de las tareas llevadas a cabo durante estas jornadas es la de colocar dos cámaras sobre un robot, una de ellas toma imágenes en RGB y la otra térmicas, todo este proceso puede encontrarse en (Morales y col., 2021) y las imágenes tomadas durante el mismo en el enlace a pie de página².

Estas imágenes son las que se han utilizado en la última etapa del trabajo. Cuentan con una resolución de 704 x 576. Las imágenes térmicas no coinciden al 100 % con las RGB, esto debido a que la cámara que trabaja en el espectro infrarrojo tiene un campo de visión horizontal de 44 ° mientras que en la dedicada al espectro visible es de 57,8 °(Morales y col., 2021). Esto puede observarse en la figura 2.5, en la imagen b no aparece el edificio de la izquierda que si se ve en la imagen a. Lo mismo ocurre por el lado derecho de la imagen y los límites inferior y superior.

²<https://www.uma.es/robotics-and-mechatronics/cms/menu/robotica-y-mecatronica/datasets/>



Figura 2.5: Ejemplo de imágenes del dataset de las jornadas de rescate de la UMA. RGB (a). Térmica (b).

2.2.3. Google colab

Google colab es una plataforma desarrollada por Google para programar aplicaciones de aprendizaje profundo. Para acceder a ella basta con buscarla a través de internet e iniciar sesión con una cuenta de Google.

Google colab cuenta con todos los recursos necesarios, incluidos Tensorflow y Python con todas las librerías que puedan hacer falta. Una vez se ejecute el código, este lo hará en la nube, haciendo uso de los recursos de Google. Para hacer uso de dichos recursos se ofrece la opción de ejecutar el código en una CPU, GPU o TPU. En caso de las dos últimas el tiempo está limitado a 8 horas diarias.

El formato de la plataforma está basado en jupyter notebooks, de hecho, el mismo código funciona en ambas aplicaciones. De esta manera, Google-colab guardará los archivos en forma de cuadernos en una ubicación, llamada Colab Notebooks, que creará automáticamente en el drive de la cuenta utilizada para iniciar sesión.

Esta plataforma es la que se ha utilizado para la segunda fase de trabajo. Concretamente, se ha utilizado con la versión 1.15.2 de Tensorflow y

ejecución en GPU. Se ha elegido esta versión ya que los autores de AdapNet++ utilizan Tensorflow gpu 1.4 y Google colab no ofrece ninguna versión anterior a la 1.15.2.

2.2.4. CVAT (Computer Vision Anotation Tool)

CVAT es una herramienta desarrollada por Intel para el etiquetado de imágenes. Este programa cuenta tanto con versión online como una que se ejecuta de manera local. La diferencia reside en que en la segunda ofrece la posibilidad de cargar modelos de red neuronal artificial propios, y utilizarlos sobre el dataset para facilitar el proceso de etiquetado si es que las categorías que se pretende etiquetar están también contempladas en la red cargada. En la versión online, solo pueden usarse los modelos cargados por defecto, pero no existe la posibilidad de cargar uno nuevo, además la cantidad de tareas que pueden crearse está limitada. Se ha decidido utilizar la herramienta online ya que las categorías que se pretendían etiquetar están contempladas en uno de los modelos cargados por defecto. El proceso seguido para etiquetar imágenes utilizando esta herramienta se detalla en el capítulo 5.

Capítulo 3

Principios teóricos

En este capítulo se explicarán los principios teóricos en los que se fundamentan las redes neuronales, comenzando por su unidad más básica, la neurona, hasta los diferentes tipos de topologías que surgen de la combinación de las mismas y algoritmos utilizados por las redes neuronales durante su proceso de entrenamiento. Todos estos principios se explicarán brevemente, si se desea profundizar más puede consultarse (Planche y Andres, 2019), (Ng, s.f.), o (Goodfellow y col., 2016).

El objetivo principal de las aplicaciones que utilizan redes neuronales artificiales es el de detectar una serie de patrones en una imagen que permitan identificar algunos de los elementos que las componen. Dichos elementos que se persigue detectar se llaman clases y son propios de cada aplicación. Por ejemplo, una aplicación destinada a la conducción autónoma de vehículos buscará identificar elementos como peatones o señales de tráfico por lo que estas serán sus clases. Para detectar con exactitud dichas clases las redes neuronales artificiales cuentan con una serie de elementos que se explicarán en este capítulo.

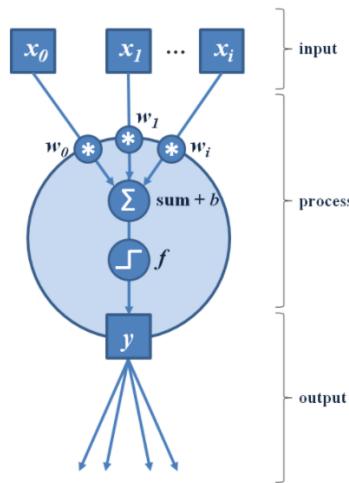


Figura 3.1: Representación de una neurona en IA. (Fuente: (Planche y Andres, 2019)).

3.1. Unidad básica. La neurona

Al igual que un cerebro humano, las redes neuronales utilizadas en IA (Inteligencia Artificial) se componen de neuronas agrupadas de diferentes formas (cabe destacar en este punto que el cerebro humano es mucho más profundo y tiene un funcionamiento mucho más complejo que cualquier red neuronal artificial, que son solo una aproximación). Estas neuronas, transforman entradas, formadas por vectores de números, en unas salidas mediante operaciones matemáticas, el proceso aparece representado en la figura 3.1.

El proceso que se lleva a cabo en cada neurona es el siguiente:

$$z = x \cdot w + b \quad (3.1)$$

- Donde el término x representa las entradas de la neurona.
- w representa los pesos de la neurona. Estos pesos son propios de cada neurona y se ajustan durante el entrenamiento.
- b representa el sesgo, se añade opcionalmente y que también se ajusta durante el entrenamiento.

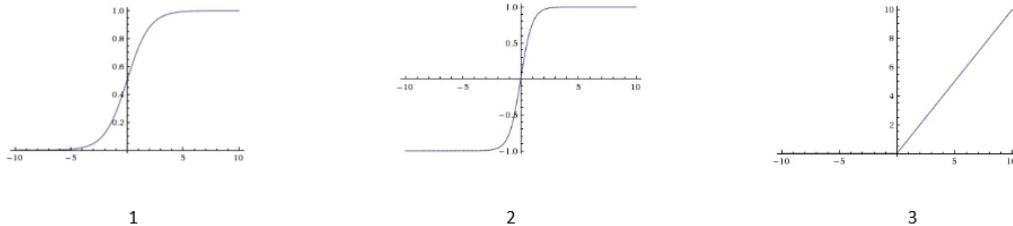


Figura 3.2: Funciones de activación. Sigmoide (1). Tangente hiperbólica (2). ReLU (3). (Fuente:University, s.f.)

Una vez calculada esta variable z se aplica sobre la misma una función de activación.

$$y = f(z) \quad (3.2)$$

En el caso de la neurona de la figura 3.1 la función de activación consiste en un escalón. Sin embargo, en las redes neuronales artificiales más recientes se utilizan otras funciones que permiten no linealidades y por tanto modelos más complejos. Las más utilizadas son sigmoide, tangente hiperbólica y ReLu (figura 3.2).

Una vez desarrollada la neurona es necesario realizar agrupaciones de las mismas para poder formar una red neuronal. Dichas agrupaciones se llaman capas y en cada una de ellas se extraen un tipo diferente de información. Normalmente en las primeras capas se extraen las características más generales mientras que en las últimas se extraen las más específicas o los detalles. En la figura 3.3 se representa una red simple. En esta red las salidas de las neuronas de cada capa son las entradas de todas las neuronas de la capa siguiente, este tipo de capas se denominan capas totalmente conectadas (dense layer).

3.2. Proceso de entrenamiento

Este es el proceso mediante el cual se ajustan los parámetros de la red (w y b en la ecuación 3.1). Para realizar esto es necesario pasar unas

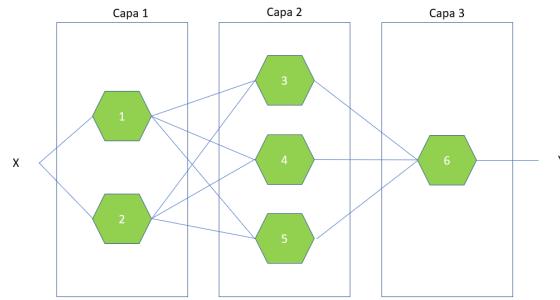


Figura 3.3: Representación de una red neuronal. Se representan las entradas de la red como X y las salidas como Y (Fuente: Elaboración propia)

entradas conocidas, evaluarlas y comparar la salida de la red neuronal con la salida que se ha establecido como la correcta, en función del error cometido se ajustarán los parámetros.

Esto implica que antes de comenzar el entrenamiento es necesario definir la salida correcta para cada entrada. Este proceso es el etiquetado. Un ejemplo del mismo puede darse en una aplicación cuyo objetivo es detectar coches. Primero se deberá de definir sobre las entradas que se utilizarán en el entrenamiento donde están los coches. De esta manera se podrá evaluar el error en el que incurre la red cuando realiza su predicción comparado la predicción realizada (salida de la red) con la imagen etiquetada correspondiente. En función de este error se ajustan los parámetros de dicha red. Este tipo de entrenamiento en concreto se conoce como entrenamiento supervisado, aunque existen otros tipos como el entrenamiento no supervisado o el entrenamiento por refuerzo (Planche y Andres, 2019).

3.3. Función de pérdida y backpropagation

Independientemente del tipo de entrenamiento utilizado, será necesario evaluar las pérdidas que se producen en las salidas durante la etapa de entrenamiento. Existen varios métodos, como pueden ser L1, L2 o entropía binaria cruzada (University, s.f.). Cada uno de ellos presenta una serie de características pero en todos ellos se evalúa la diferencia entre salida de la

red (un vector cuyas componentes representan la probabilidad, entre cero y uno, del elemento sobre el que se hace la predicción de pertenecer a cada clase) y la etiqueta correspondiente a la entrada que se está evaluando. Si por ejemplo, el objetivo de la red es diferenciar entre imágenes de gatos y perros, la predicción consistirá en un vector con dos componentes correspondientes a la probabilidad de pertenecer a una de las clases. Por otra parte, la etiqueta consistirá en un vector con valor uno en la componente correspondiente a la clase correcta y cero en la otra. En base a estos dos vectores se evalúa la pérdida o error en el que incurre la red.

Una vez se calcula la pérdida o error cometido, es necesario variar los pesos de la red para que el valor de la misma se reduzca conforme avance el entrenamiento, por lo tanto, es imprescindible saber cómo la variación en cada peso afecta a la pérdida. Para esto se utiliza el gradiente de la función de pérdida, que calcula la derivada de dicha función respecto a cada peso con el objetivo de variar dicho peso en función de si esta derivada es positiva o negativa, de esta manera se pretende alcanzar el mínimo de la función de pérdida, este proceso se conoce como descenso estocástico de gradiente (SGD) y aparece representado en la figura 3.4.

Esta derivada se calcula en cada iteración del entrenamiento aplicando para ello la regla de la cadena. Esta regla afirma que la derivada respecto a los parámetros de una capa k puede calcularse conociendo las entradas y salidas de dicha capa y la derivada respecto a los parámetros de la capa $k+1$.

Por este motivo el proceso comienza por calcular las derivadas en la última capa de la red para después ir calculando las sucesivas derivadas hasta llegar a la primera capa, esta forma de “propagar la derivada” se denomina backpropagation. El valor de la derivada de la función de pérdida en la última capa dependerá de la función de pérdida escogida y su expresión es conocida para las funciones más utilizadas. El valor de la derivada respecto a cada peso se calcula en cada iteración. En caso de que en una iteración se pase un conjunto de imágenes, se calculará la media de las derivadas obtenidas con cada imagen. El tamaño del conjunto de imágenes se conoce como batch size o tamaño de lote y es recomendable que su valor sea potencia de dos para acelerar el proceso de entrenamiento. El proceso de minimización de pérdida sobre todo el conjunto de entrenamiento una única vez se llama época (epoch).

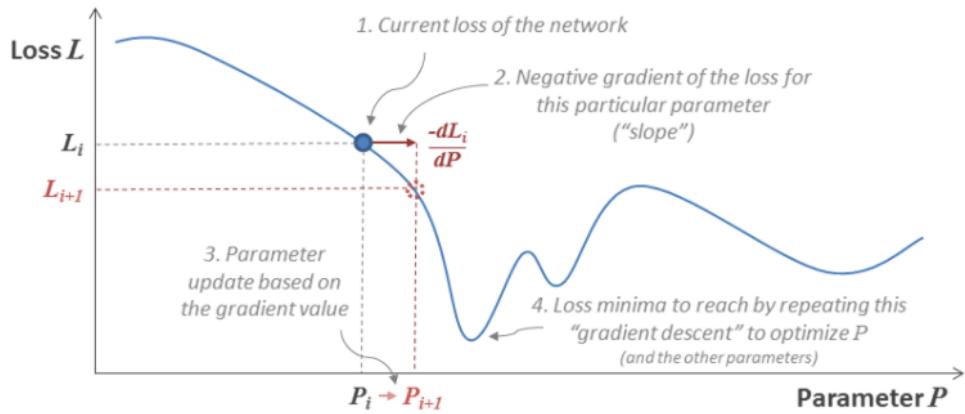


Figura 3.4: Ilustración del proceso de descenso de gradiente. (Fuente: Planche y Andres, 2019)

Sabiendo todo esto, la actualización de un peso W_i cualquiera en la iteración $i+1$ del entrenamiento será.

$$W_{i+1} = W_i - \xi \frac{dL}{dW_i} \quad (3.3)$$

- $\frac{dL}{dW_i}$ representa la derivada de la función de pérdida (L) respecto al peso W en la iteración i (W_i)
- ξ es un parámetro conocido como learning rate o tasa de aprendizaje y determina la rapidez con la que varía un peso, un valor muy elevado hará que la variación sea más rápida a costa del riesgo de no poder alcanzar el mínimo de la función. Un valor muy pequeño evita esto, pero el entrenamiento se hace más lento.

3.4. Infraajuste y sobreajuste

Cuando el proceso de entrenamiento de la red se lleva a la práctica hay dos problemas principales que pueden ocurrir. El primero de ellos consiste en que el valor de la pérdida es demasiado alto como para que la red entrenada

reconozca los patrones característicos de las clases a identificar y es conocido como infraajuste. El motivo es que los parámetros de la red no se han ajustado lo suficiente a los patrones de los ejemplos de entrenamiento.

Lo contrario sucede cuando los parámetros se ajustan demasiado a los ejemplos de entrenamiento. Esto provoca que las pérdidas durante el entrenamiento sean muy bajas, el modelo se ajusta muy bien a los ejemplos utilizados en esta etapa, pero no aprende los patrones más generales por lo que falla cuando se utilizan ejemplos nuevos. Esto suele ocurrir cuando no se dispone de suficientes ejemplos para realizar el entrenamiento o cuando se utiliza una red muy compleja para una tarea muy simple. Un ejemplo de sobreajuste se puede observar en la figura 3.5. El problema que presenta la línea verde de dicha figura es que se ajusta al ruido del conjunto de datos, esto es, elementos de la clase azul que tienen características de la clase roja y viceversa. Esto puede provocar que cuando se utilice el modelo para realizar una predicción, un elemento que pertenezca a la clase roja se identifique como azul. En la figura 3.6 se representa el caso contrario.

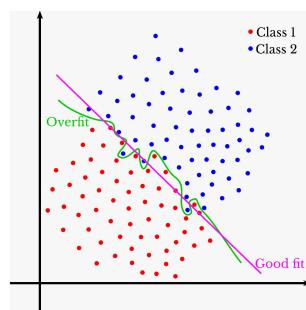


Figura 3.5: Ejemplo de sobreajuste. (Fuente: <https://towardsdatascience.com/dont-overfit-ii-how-to-avoid-overfitting-in-your-machine-learning-and-deep-learning-models-2ff903f4b36a>).

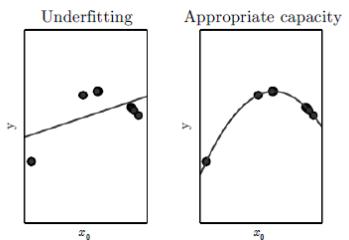


Figura 3.6: Ejemplo de infraajuste (Fuente: Goodfellow y col., 2016).

3.5. Redes neuronales convolucionales (CNN)

La red que se presentó en la figura 3.5 es una red totalmente conectada. Este tipo de redes presentan dos problemas principales. Una gran cantidad de parámetros, incluso para redes simples, y la pérdida de “visión espacial” de la red en el caso de que sus entradas sean imágenes.

Para resolver estos problemas surgen las redes convolucionales. En estas redes cada neurona está conectada a un número determinado de neuronas de la capa siguiente, de esta manera, el primero de los problemas queda solucionado pues el número de parámetros a entrenar se reduce considerablemente, además, como cada neurona está conectada únicamente a un conjunto determinado de otras se puede preservar la información espacial, ya que, como se verá más adelante, puede determinarse sobre qué región actúa cada neurona (Planche y Andres, 2019). Este tipo de capa se denomina capa convolucional.

En una capa convolucional las neuronas que la forman comparten pesos y sesgo y su funcionamiento puede verse como una sola neurona que se desliza sobre la matriz de entrada actuando sobre diferentes zonas de la misma. Esto se traduce en la operación que da nombre a este tipo de capa, la convolución. Cada grupo de neuronas que comparten parámetros se comportará como una máscara que opera sobre determinadas zonas de la matriz de entrada. En la figura 3.7 se representa esta operación, en este caso la máscara está representada por una matriz 3x3 y la entrada por una matriz 6x6. Puede observarse que la máscara cuenta con una serie de parámetros, estos parámetros corresponden a los que constituyen una máscara de Sobel

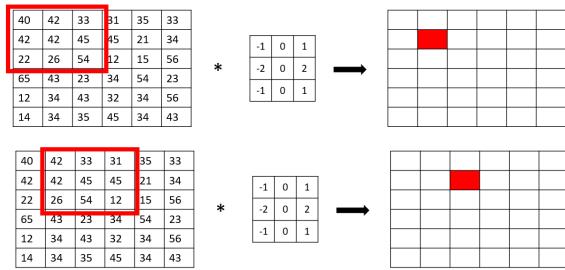


Figura 3.7: Operación de convolución. (Fuente: Elaboración propia).



Figura 3.8: Aplicación de la máscara de Sobel. Imagen original a la izquierda y resultado a la derecha. (Fuente:<https://towardsdatascience.com/magic-of-the-sobel-operator-bbbcb15af20d>).

(utilizada en aplicaciones clásicas de aprendizaje automático). Se utilizará este tipo de máscara para explicar cómo funciona la convolución.

La máscara de la figura 3.7(máscara de Sobel) aplica la operación gradiente sobre conjuntos de píxeles de tamaño 3x3, de modo que, si hay un cambio brusco en la luminosidad en ese conjunto de píxeles, el resultado de la convolución será no nulo, si la luminosidad no cambia, el resultado es cero. Cuando se aplica esta máscara para el gradiente tanto en el eje horizontal como vertical (misma matriz traspuesta), se obtiene como resultado los bordes de la imagen sobre la que se aplican ambas máscaras (figura 3.8).

Al igual que esta aplicación clásica de visión por computador, las redes convolucionales también utilizan la operación de convolución para detectar patrones en las imágenes. La principal diferencia radica en que en este se-

gundo caso, los parámetros de la máscara no son asignados al diseñar la aplicación para detectar un patrón determinado. En su lugar, las máscaras están compuestas por los pesos de las neuronas que forman la red y que se van ajustando durante el proceso de entrenamiento. El resultado de aplicar una máscara sobre la entrada se denomina mapa de características.

Puesto que las neuronas de cada capa conectadas a la misma salida comparten pesos y sesgo, cada máscara representará a un grupo de neuronas durante la convolución. Si una capa cuenta con N grupos de neuronas (cada grupo conectado a una salida), tendrá N máscaras y N mapas de características.

3.5.1. Hiperparámetros de una capa convolucional

La operación de convolución realizada en cada capa no solo depende de los pesos de la red, también influyen otros parámetros conocidos como hiperparámetros, estos están formados por:

- **Tamaño de máscara o filtro:** Es el tamaño de la máscara que realiza la operación de convolución. Las distintas máscaras de una capa convolucional no tienen por qué tener el mismo tamaño. Cabe destacar que las máscaras empleadas en redes convolucionales son cuadradas.
- **Stride:** Define sobre qué posiciones puede actuar la máscara o cuánto se desliza sobre la matriz de entrada (figura 3.9).
- **Orlado (padding):** Si se observa la operación definida en la figura 3.7, la máscara no puede actuar sobre los parámetros ubicados en los bordes de la matriz. Para solucionar esto se añaden filas y columnas de ceros como puede verse en la figura 3.10. En este caso se añade una fila y una columna aunque pueden añadirse más dependiendo del tamaño del filtro.
- **Número de filtros:** Número de máscaras distintas en una capa convolucional.

Teniendo en cuenta estos parámetros puede controlarse el tamaño

de la salida de cada capa, lo cual es especialmente útil en aplicaciones de segmentación.

40	42	33	31	35	33
42	42	45	45	21	34
22	26	54	12	15	56
65	43	23	34	54	23
12	34	43	32	34	56
14	34	35	45	34	43

40	42	33	31	35	33
42	42	45	45	21	34
22	26	54	12	15	56
65	43	23	34	54	23
12	34	43	32	34	56
14	34	35	45	34	43

Figura 3.9: Deslizamiento de la máscara con un stride de 2. (Fuente: Elaboración propia).

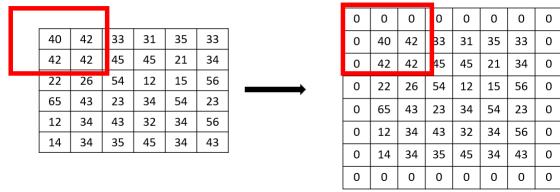


Figura 3.10: Orlando de una matriz. (Fuente: Elaboración propia).

3.6. Capas de pooling

Este tipo de capas son utilizadas junto con las capas convolucionales para reducir el número de parámetros a entrenar y para controlar el tamaño de los datos que circulan por la red. La peculiaridad de este tipo de capas es que no poseen parámetros a entrenar, simplemente constan de una máscara de tamaño $K \times K$ que devuelve un único valor para cada una de las regiones sobre las que actúa. Las más comunes son la capa max pooling y la capa average pooling. El funcionamiento de la primera se ilustra en la figura 3.11. El parámetro stride en este tipo de capas suele elegirse igual a k (tamaño de la máscara), para evitar solapes.

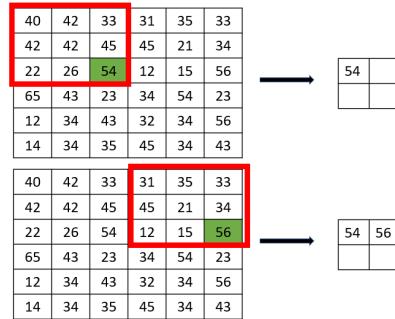


Figura 3.11: Capa max-pooling con k y stride de 3. (Fuente: Elaboración propia).

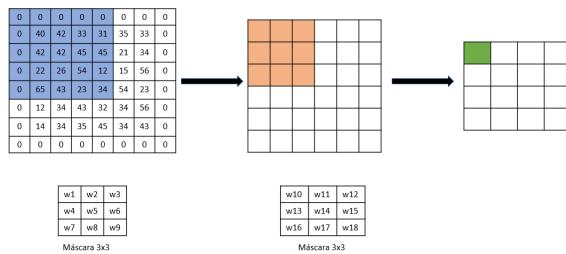


Figura 3.12: Campo receptivo de una capa tras la aplicación de dos máscaras 3x3. (Fuente: Elaboración propia)

3.7. Campo receptivo efectivo

El campo receptivo efectivo hace referencia a la región de la imagen de entrada que interviene en la activación de una neurona de cualquier capa, esta propiedad es muy importante a la hora de conservar la información espacial en una red convolucional. Esto aparece reflejado en la figura 3.12. En este caso la región en azul de la entrada afectará a cualquier neurona que actúa sobre la porción verde de la salida resultante de aplicar dos máscaras 3x3.

3.8. Mejoras en el proceso de entrenamiento de una red neuronal artificial

El proceso de entrenamiento descrito hasta ahora (SGD), presenta una serie de desafíos que se han solucionado con el avance de las técnicas de visión por computador.

El primero de ellos hace referencia al parámetro ξ , tasa de aprendizaje (learning rate), este parámetro debe ser elegido con cuidado para evitar los problemas de lentitud en la convergencia o mínimos locales. La solución a esto consiste en hacer que el parámetro varíe conforme avance el proceso de entrenamiento, siendo mayor al principio para acelerar la convergencia y más pequeño al final para encontrar un mínimo adecuado. La variable que determina como varía ξ , se denomina decaimiento de la tasa de aprendizaje (learning rate decay), y los diferentes lenguajes de programación utilizados para construir y entrenar redes neuronales tienen implementados algunos métodos para determinar su valor. En el caso de la red AdapNet++ utilizada en este trabajo se emplea el algoritmo Adam.

Otro de los problemas que presenta el proceso de entrenamiento basado en el descenso del gradiente reside en la posibilidad de alcanzar un mínimo local de la función de pérdida en lugar del mínimo absoluto. Debido al coste computacional que implica evaluar la función de pérdida para todos los valores posibles de los parámetros de la red para comprobar que el mínimo alcanzado es el absoluto, se considera que la solución es óptima cuando se alcanza un mínimo local.

Por último, la variación de la función de pérdida respecto a un parámetro no tiene por qué ser del mismo orden que la variación de la misma función respecto a otro parámetro. Algunos provocan variaciones más notables, esto hace que establecer una tasa de aprendizaje (learning rate) diferente para cada parámetro sea algo muy conveniente.

La solución a los problemas expuestos se encuentra en los optimizadores, algunos de ellos son:

- Los algoritmos basados en el momento (Polyak, 1964), consisten en variar el término de actualización de los pesos teniendo en cuenta los

términos de actualización anteriores en base a un parámetro llamado momento. El problema que presenta este método reside en la posibilidad de oscilar entorno a un mínimo sin llegar a alcanzarlo.

- Gradiente acelerado de Nesterov (Sutskever y col., 2013). Este método es parecido al anterior con la salvedad de que tiene en cuenta los valores de la pendiente de la función de pérdida en futuras iteraciones para variar el término de actualización.
- Familia de algoritmos Ada. Compuesto por:
 - Adagrad (Duchi y col., 2011), consiste en disminuir más rápidamente la tasa de aprendizaje de aquellos parámetros vinculados a capas o neuronas que detecten con mayor frecuencia la característica en la que se centran y más lentamente los de aquellas que se activen con menos frecuencia. Este algoritmo estabiliza el proceso de SGD (descenso de gradiente).
 - Adadelta (Zeiler, 2012), este algoritmo revisa los parámetros utilizados para disminuir la tasa de aprendizaje en cada iteración. Con esto soluciona uno de los problemas de Adagrad, en el que la tasa de aprendizaje puede llegar a ser tan baja que la red no mejore.
 - Adam siglas de Adaptative moment estimator (Kingma y Ba, 2017), puede verse como una combinación de Adadelta y los algoritmos basados en el momento. En este caso no solo se tiene en cuenta el valor de términos de actualización anteriores, también intervienen los valores del momento de iteraciones pasadas.

Todos estos algoritmos son ampliamente utilizados en aplicaciones de visión por computador por lo que suelen estar ya implementados en entornos como Tensorflow.

3.9. Regularización del error

Una de las situaciones a evitar durante el entrenamiento es el sobreajuste, y esto puede consagrarse deteniendo el entrenamiento en el momento

adecuado. Para saber cuál es este momento se utilizan métodos como la validación cruzada (Planche y Andres, 2019).

En la práctica se suele representar el error conforme avanza el entrenamiento y se van guardando, no solo los pesos de la última iteración, sino también los pesos de las iteraciones anteriores. Con los pesos de la red en diferentes etapas del entrenamiento se evalúa la red sobre el conjunto de validación y se determina en qué momento se alcanzó el óptimo sin incurrir en sobreajuste.

3.9.1. Regularizadores L1 y L2

Una de las formas de prevenir el sobreajuste es modificar la función de pérdida de modo que al minimizarla se alcance la regularización del error. Los regularizadores L1 y L2 modifican las expresiones de las funciones de pérdida L1 y L2 para incluir la regularización en el proceso de entrenamiento (Andrew Ng, s.f.).

La idea que subyace en el regularizador L1 es que los parámetros asociados a características poco importantes se aproximen a cero, de modo que la red reaccione únicamente a las características más importantes.

Por otra parte, el regularizador L2 penaliza a aquellos términos más grandes, manteniendo los parámetros de la red lo más bajos y homogéneos posible, ya que un conjunto de valores muy altos no cumple con el propósito general de una red neuronal (Planche y Andres, 2019).

3.10. Dropout

Otra de las técnicas para lograr la regularización de la red es el dropout. Este método consiste en desactivar neuronas de manera aleatoria en cada iteración del entrenamiento, por lo que, si se desactivan neuronas asociadas a una característica importante, las neuronas restantes aprenderán otra característica relevante que permita a la red arrojar predicciones correctas para todos los ejemplos evitando así el sobreajuste (Ng, s.f.).

3.11. Normalización por lotes (Batch normalization)

Esta técnica se basa en que el entrenamiento de una capa es más rápido cuando sus entradas están normalizadas(Ng, s.f.). No existe un consenso acerca de si es mejor normalizar la variable z o la salida de la función de activación (ecuación 3.1), aunque los más común es normalizar z . La operación que se lleva a cabo es:

$$z_{norm}^i = \frac{(z^i - \mu)}{\sqrt{(\sigma^2 + \varepsilon)}} \quad (3.4)$$

$$z_i^\wedge = \gamma z_{norm}^i + \beta \quad (3.5)$$

- μ es la media de los valores de z de la capa sobre la cual se aplica batch normalization.
- σ^2 es la desviación típica y ε se añade teniendo en cuenta el caso de que σ^2 sea nulo.
- γ y β son parámetros entrenables.

La función de activación actúa sobre el resultado de esta operación (z_i^\wedge), que al estar normalizada permitirá una convergencia más rápida además de que la red sea más robusta (Planche y Andres, 2019).

3.12. Avances adicionales para mejorar el rendimiento de una red neuronal

En este apartado se explicarán brevemente otras contribuciones aportadas a la arquitectura de las redes neuronales que junto con todo lo anterior han ayudado a mejorar su desempeño (Planche y Andres, 2019).

Empleo de convoluciones más pequeñas

Los autores de este método comprobaron que dos convoluciones 3x3 tienen el mismo campo receptivo que una convolución 5x5 pero empleando

menos parámetros. Este efecto se acentúa a la vez que más convoluciones 3x3 se agrupan de manera consecutiva. De este modo se conserva la información espacial de la imagen mientras se reduce el número de parámetros y aumenta la no linealidad.

Agrupar capas en paralelo y no en serie

Una de las formas de aumentar la eficiencia de la red es hacer que esta se componga de diferentes submódulos o subredes. Cada una de estas subredes cuenta con capas dispuestas tanto en serie como en paralelo. Las capas en paralelo, cada una con un tamaño de máscara distinto, reaccionan a características con diferentes escalas aumentando el rango de información que la red puede capturar. Una aproximación aún más exhaustiva de este método es el enfoque NiN (Network in Network) (Lin y col., 2014).

Cuellos de botella

Un cuello de botella no es más que una convolución 1x1 con un stride de 1. Esta técnica se utiliza para reducir el número de parámetros que utiliza la convolución que se encuentra a continuación del cuello de botella por lo que suele ubicarse antes de las convoluciones que precisan una mayor cantidad de parámetros a entrenar para reducir este número.

Emplear una capa de pooling al final de la red en lugar de una totalmente conectada.

Como se mencionó con anterioridad, las redes convolucionales suelen acabar con una capa totalmente conectada, que es la que computa la salida de la CNN y arroja las probabilidades finales. Al reemplazar esta capa por una capa de pooling, la red empeora un poco su capacidad de clasificación a cambio de una enorme reducción el número de parámetros a entrenar.

Pérdidas intermedias.

Uno de los problemas que afrontan redes muy grandes es la aproximación del gradiente de la función de pérdida a cero conforme el proceso de backpropagation se acerca a las primeras capas, lo cual provoca que los pesos de estas no se actualicen adecuadamente. Para solucionar esto se introducen pérdidas adicionales en varios puntos de la red.

Existen más contribuciones, algunas de ellas se utilizan en la propia

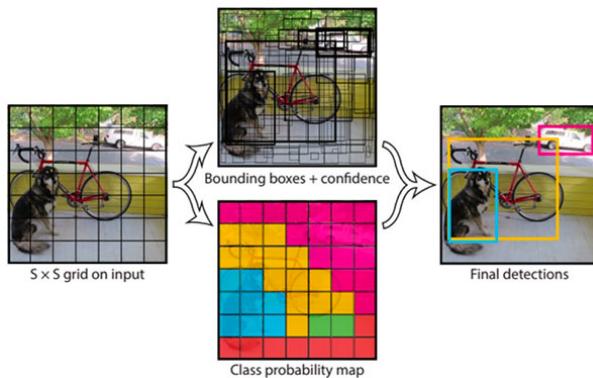


Figura 3.13: Ejemplo de imagen tratada por la red YOLO. (Fuente: <https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/>)

AdapNet++, pero se explicarán en el siguiente capítulo.

3.13. Uso de CNN en segmentación de imágenes

En este capítulo se presentarán las diferentes formas en las que una red neuronal puede actuar sobre una imagen, haciendo hincapié en la segmentación de las mismas ya que la red utilizada para realizar este trabajo se dedica a esta tarea. La operación de convolución a través de las neuronas de una red permite extraer características de una imagen, estas características pueden utilizarse con distintos objetivos, los principales son: clasificación, detección y segmentación (Planche y Andres, 2019).

Las clasificación consiste en determinar a qué categoría pertenece una imagen, por ejemplo, diferenciar si una persona en una foto es hombre o mujer. La detección va más allá y pretende determinar, dentro de una imagen, en qué zonas se encuentran las distintas clases que se desean detectar, por ejemplo, donde hay personas en una foto, un ejemplo de esta tarea se presenta en la figura 3.13.

La segmentación de imágenes consiste, no solo en detectar en que

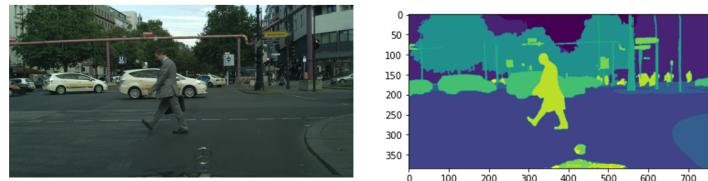


Figura 3.14: Ejemplo de segmentación de una imagen. (Fuente: Elaboración propia).

regiones de la imagen se encuentra cada clase, sino en especificar que píxeles pertenecen a dicha clase, de modo que el resultado es otra imagen en el que cada uno de los objetos presentes en la entrada aparecen ligados a una de las categorías para las cuales ha sido entrenada la red. Todo esto aparece representado en la figura 3.14. Para conseguirlo, hacen uso de arquitecturas encoder-decoder.

3.13.1. Arquitecturas encoder-decoder

Este tipo de arquitecturas se utilizan para una gran variedad de aplicaciones. En ellas el encoder se encarga de recoger información relevante de la entrada y mapearla, en forma de un conjunto estructurado de valores, en un espacio latente. Por otro lado, el decoder recoge estos valores y los proyecta en el formato de destino. La característica principal del espacio latente en el que se mapea la entrada es que este es más pequeño que la propia entrada y por lo tanto ocupa menos memoria.

Las redes neuronales convolucionales(CNN) son una gran herramienta a la hora de extraer características relevantes de una imagen y de reducir “el tamaño” de las mismas (Planche y Andres, 2019),por lo que funcionan muy bien si se utilizan como encoder de este tipo de arquitecturas. Por otra parte, para que el decoder pueda reconstruir la imagen, se introducen dos nuevos tipos de capa.

Convolución traspuesta o deconvolución

La deconvolución es una operación a través de la cual se pretende deshacer una convolución realizada previamente. Esto es, conociendo el tamaño de máscara, stride, padding... de una convolución, crear una capa que sea capaz de recuperar el tamaño original de la entrada. El proceso para conseguir esto aparece reflejado en la figura 3.15. En esta figura, los parámetros de la máscara (filter w) se ajustan durante el entrenamiento. El parámetro dilation (especifica cuantos ceros se introducen en la matriz de entrada) también puede aplicarse a las máscaras de las capas de convolución en lo que se conoce como convolución dilatada. Con esto se consigue aumentar el campo receptivo de las neuronas.

Unpooling

Al igual que la deconvolución, una capa de unpooling tiene el objetivo de deshacer el resultado de una capa de pooling. Para conseguirlo primero se modifican ligeramente las capas de pooling para que, además de su salida, proporcionen una máscara de pooling. En el caso de un max-pooling, será una

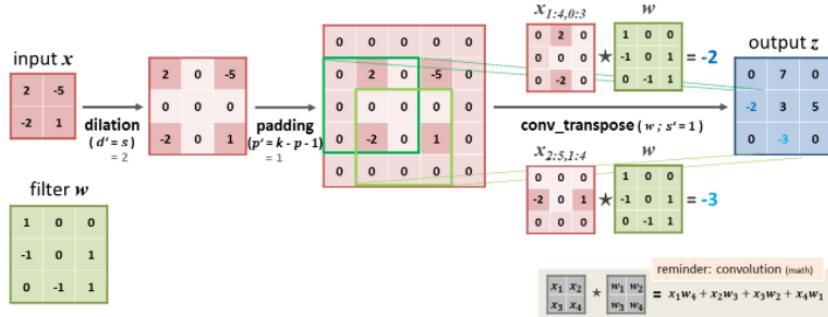


Figura 3.15: Convolución traspuesta. (Fuente: (Planche y Andres, 2019))

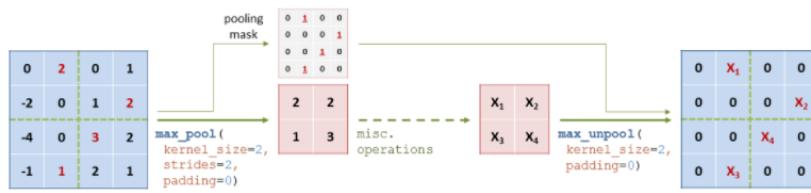


Figura 3.16: Ejemplo de unpooling. (Fuente: (Planche y Andres, 2019))

matriz en la que todos los valores son cero excepto aquellos correspondientes a las posiciones originales de los máximos(Planche y Andres, 2019). A partir de esto se reconstruye una versión aproximada del input la capa de pooling (figura 3.16). El mismo proceso aplicado a una capa average-pooling se conoce como up sampling.

Puesto que estas capas no tienen pesos, suele ubicarse tras ellas una convolución de stride 1 y con un padding que permita conservar las dimensiones. Así se consigue dotar a estas estructuras del decoder de parámetros entrenables.

Una de las peculiaridades que presenta AdapNet++ frente a la arquitectura encoder-decoder clásica es, que al efectuar una predicción a partir de dos modalidades distintas de imagen, cuenta con dos encoders, en lugar de uno, además de otros bloques que permiten la fusión de las características extraídas de ambos encoders como paso previo al decoder.

Otra de las labores que puede perseguirse con la segmentación es la segmentación de instancias. Esta técnica consiste en no solo asignar una clase a cada píxel si no en determinar el número de objetos asignados a cada clase, por ejemplo, determinar el número de personas en una imagen. Para conseguirlo hay que utilizar una serie de métodos como por ejemplo RPN(Region Proposal Network) (Ren y col., 2016) o RoIAlign (He y col., 2018).

Cálculo de la pérdidas en CNN utilizadas en segmentación

En segmentación de imágenes, además de las funciones de pérdida presentadas hasta ahora, se pueden encontrar otras que están más orientadas a esta labor, como por ejemplo el índice F1 (método precision-recall) también llamado coeficiente Dice o el índice IoU(Intersection over Union)(Planche y Andres, 2019). El objetivo de estos índices es medir cuantos píxeles se clasifican correctamente en relación al total. Por ejemplo el coeficiente Dice se calcula como:

$$Dice(A, B) = \frac{2(A \cap B)}{|A| + |B|} \quad (3.6)$$

Si la predicción A es perfecta, es decir A igual a B, el valor del coeficiente será uno, si A y B no coinciden en ningún punto será cero. Cuando se pretenden detectar varias clases se calcula la media de los coeficientes para cada clase, el valor resultante en caso de utilizar el IoU es el MIoU (Mean Intersection over Union), muy utilizado en segmentación.

Para poder incorporar estos coeficientes a la función de pérdida es necesario realizar la operación $Pérdida = 1 - Coeficiente$. Esto porque el objetivo que se persigue al entrenar la red es aproximar las pérdidas a cero lo máximo posible. Además de todo esto también suelen aplicarse técnicas como CRF (Conditional Random Fields), que toma información de la imagen de entrada para mejorar definición de la imagen resultante de la predicción.

3.14. Transferencia de conocimiento, ajuste fino y aumento de datos

Ya se han visto las diferentes aplicaciones que puede tener una red neuronal a la hora de trabajar con imágenes, así como el funcionamiento de las mismas y su proceso de entrenamiento entre otros. En este punto puede surgir una pregunta, si se tiene una red ya entrenada y se pretende utilizar para una aplicación similar, ¿es necesario volver a entrenarla desde cero? Para responder a esta pregunta surge la transferencia de conocimiento (transfer learning). Esta técnica se basa en la propiedad que tienen las primeras capas de las redes neuronales para extraer la información más general mientras que las últimas se centran en aquellas características más propias de la aplicación para la cual se va a utilizar la red. Las técnicas de transfer learning buscan conservar sin alteración o sin prácticamente alteración las primeras capas mientras que se ajustan solo los parámetros de las últimas o directamente se sustituyen por capas con un diseño distinto.

Una vez terminado el proceso descrito en el párrafo anterior, se tendrá una red compuesta por unos pesos entrenados para otra aplicación en las primeras capas, y unos pesos específicos para la nueva aplicación en las últimas. Si se desea adaptar la red un poco más a la nueva aplicación puede volver a entrenarse partiendo de los pesos que ya tiene realizando un ligero ajuste sobre los mismos. Esto se conoce como ajuste fino (fine tuning).

Las dos técnicas descritas son muy útiles cuando se quiere utilizar una red neuronal para una aplicación similar y no se dispone de una gran cantidad de ejemplos para el entrenamiento, aunque esto no quiere decir que no pueda actuarse sobre dichos ejemplos para disponer de un conjunto de datos más variado, y es que cuanta más diversidad posea este conjunto, más robusta será la red obtenida tras el entrenamiento. Esta diversidad puede conseguirse de manera artificial con técnicas de data augmentation o aumento de datos. Las más comunes, referidas a imágenes, son: aplicar una simetría horizontal o vertical, hacer zoom sobre una región de la imagen, aumentar o disminuir brillo e introducir ruido. La técnica de aumento de datos utilizada dependerá del tipo de datos con los que se trabaje.

Capítulo 4

AdapNet++

La red neuronal AdapNet++, es una red DCNN (Deep Convolutional Neural Network) utilizada para segmentación semántica de imágenes y que ha demostrado un desempeño superior al del resto de redes que actualmente pueden encontrarse sobre los mismos datasets (Valada y col., 2019). No obstante, se han utilizado varios de los avances introducidos por estas redes a la hora de definir su arquitectura. En este capítulo se explicarán brevemente estas novedades, así como las introducidas por los propios autores de la red para presentar una visión acerca de cómo está construida. Información mucho más detallada puede encontrarse en Valada y col., 2019.

Esta DCNN sigue una arquitectura encoder-decoder pero la principal diferencia con lo explicado en apartados anteriores reside en la fusión de mapas de características resultantes de varias redes AdapNet++, las cuales tienen como entrada imágenes de espectros distintos. Concretamente, utilizan dos redes AdapNet++, una que recibe como entrada una imagen RGB y otra que tiene como entrada el mapa de profundidad correspondiente a la misma escena representada en la imagen RGB. La fusión de ambos mapas se hace mediante el bloque SSMA (Self Supervised Model Adaption) que se explicará más adelante en este capítulo. El resultado de la fusión de ambas modalidades es la entrada al decoder de la red. De este modo, la red está compuesta por dos encoders, cada uno correspondiente a una red AdapNet++ entrenada para una modalidad de imagen, bloques SSMA para la fusión de ambas modalidades, y un decoder. Concretamente, los encoders de esta red

están basados en ResNet50.

El objetivo a la hora de idear esta nueva arquitectura es mejorar los resultados obtenidos por las redes existentes, en este caso no solo se busca la mejor manera de fusionar modalidades o que transformaciones aplicar para facilitar esta fusión, sino que el bloque SSMA es capaz de identificar cuál de las modalidades fusionadas aporta información más relevante.

Las contribuciones de AdapNet++ al campo de la segmentación son:

1. Utilización del bloque SSMA para fusión multimodal.
2. Nuevos bloques residuales y fusión de características de medio y alto nivel en el decoder para mejorar la detección de bordes, así como etapas de refinamiento (skip refinement).
3. Modificación del módulo ASPP en eASPP con un mayor campo receptivo y una décima parte de parámetros.
4. Una nueva manera de “poda” de capas. Esta técnica, como se verá, funciona de manera similar al dropout, con la diferencia de que la poda no elimina capas de manera aleatoria.

4.1. Estructura general de AdapNet++

La estructura de AdapNet++ parte de una arquitectura encoder-decoder a la que se añaden algunos elementos con el fin de mejorar los resultados que se pueden obtener con arquitecturas anteriores (figura 4.1), se empezará explicando la arquitectura de AdapNet++ unimodal. El encoder está basado en la CNN (Convolutional Neural Network) ResNet50 pero utilizando los bloques residuales totalmente preactivados desarrollados por los autores. Los bloques residuales están compuestos por capas, como se ha visto hasta ahora, con la peculiaridad de que la salida de uno se suma a la función z (ecuación 3.1) de las neuronas de la capa siguiente, y a esta suma se le aplica la función de activación. En la siguiente ecuación se presenta la expresión de un bloque que no es residual.

$$z_1 = W_1 \cdot input_1 + bias_1 \rightarrow output_1 = \delta(z_1) = input_2 \rightarrow$$

$$\rightarrow z_2 = W_2 \cdot input_2 + bias_2 \rightarrow output_2 = \delta(z_2) \quad (4.1)$$

- δ es la función de activación, W los pesos de la capa y $bias$ el parámetro con el mismo nombre.

Esto es porque las salidas de las neuronas de una capa solo pueden ir a las de la capa siguiente, en cambio, en un bloque residual la salida de una neurona puede “saltarse” varias capas en lo que se conoce como conexión shortcut o skip. En la figura 4.2 se representa un bloque residual.

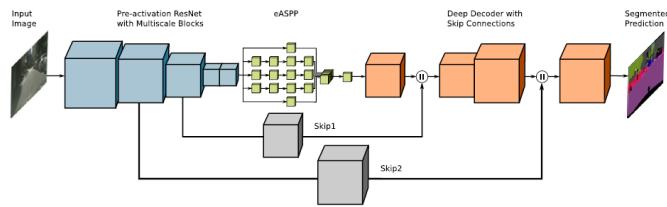


Figura 4.1: Arquitectura de la red AdapNet++. (Fuente: (Valada y col., 2019))

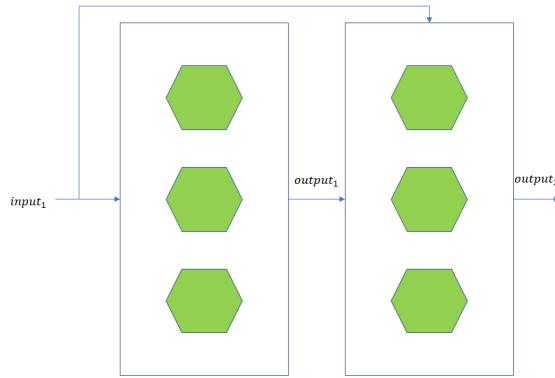


Figura 4.2: Ejemplo de bloque residual. (Fuente: Elaboración propia)

De este modo, la salida de la segunda capa será $output_2 = \delta(z_1 + input_1)$. Esta forma de configurar las capas ayuda a mejorar la propagación del error desde las capas finales hasta las iniciales, lo cual hace que sea muy útil en redes muy profundas, ya que evita la pérdida del gradiente cuando es muy cercano a cero, o el caso contrario, cuando el valor del gradiente es muy alto (Ng, s.f.).

Los bloques residuales utilizados en AdapNet++ difieren de los bloques residuales clásicos en cuestiones como el tamaño de salida del bloque, en ResNet50 el tamaño de la salida sufre un downsampling de 32, en AdapNet++ es de 16, la utilización de unidades residuales de cuello de botella debido a su mayor eficiencia computacional o la variación en el número de máscaras empleadas (Valada y col., 2019).

4.1.1. Módulo eASPP

Uno de los elementos que se pueden ver en la figura 4.1 es el módulo eASPP. Este módulo se basa en el módulo ASPP (Chen y col., 2017). El módulo original cuenta con 4 convoluciones cuyos autores llaman “atrous convolution”, y es una forma de referirse a las convoluciones dilatadas que ya se han explicado, estas se agrupan en paralelo con tasas de dilatación distintas y una capa de pooling.

Uno de los mayores inconvenientes de ASPP es la gran cantidad de parámetros a entrenar y de operaciones con punto flotante que utiliza, por lo que el primer objetivo de eASPP es reducir este número. Para ello utiliza convoluciones dilatadas en cascada y en cuellos de botella. Lo primero permite un muestreo más denso de los píxeles que con convoluciones en paralelo, con lo segundo se reduce el número de parámetros. Con todo esto se consigue una reducción del 87,87% en el número de parámetros y 89,53% de operaciones con punto flotante, mientras que se preserva información sobre el contexto de la imagen (Valada y col., 2019).

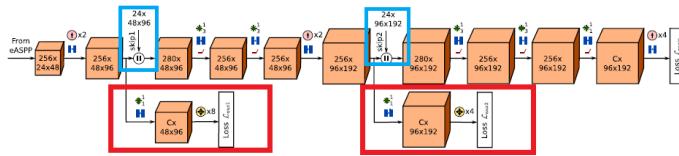


Figura 4.3: Estructura del decoder al introducir las pérdidas auxiliares.
(Fuente: Elaboración propia)

4.1.2. Decoder de AdapNet++

El decoder de AdapNet++ consta de 3 etapas. La primera duplica el tamaño de la salida del módulo e ASPP. En la segunda, se combina la salida de la primera etapa con el primer mapa de características de medio nivel, skip1 (cuadrado azul) en la figura 4.3. Al resultado de esta combinación se le aplica una convolución 3x3 para mejorar la resolución y después se efectúa una desconvolución que duplica el tamaño de la entrada. La tercera y última etapa del decoder consta de, primero, una combinación de la salida de la segunda etapa con el skip2 (cuadrado azul en la figura 4.3). A esto le siguen dos convoluciones 3x3 y por último una convolución 1x1 que iguala la profundidad de su salida al número de clases, seguida de una desconvolución que recupera la resolución original de la imagen.

En el decoder presentado en la figura 4.3 pueden observarse, en la parte inferior dos ramas que salen de la primera y segunda etapa del decoder y sobre las que se aplican varias operaciones (cuadrados rojos), en la figura se identifican como *Loss₁* y *Loss₂*. Estas ramas se introducen como pérdidas auxiliares y su función es acelerar la convergencia y ayudar al flujo del gradiente a través de las capas anteriores durante el entrenamiento.

4.1.3. Poda de neuronas

Esta estrategia funciona de manera similar al dropout. La principal diferencia reside en que la poda no elimina neuronas de manera aleatoria para después recuperarlas si no que, basándose en la contribución de cada neurona, elimina, de manera permanente, las que puntúen más bajo. El objetivo de esto es reducir el número de máscaras y por lo tanto de memoria ocupada por

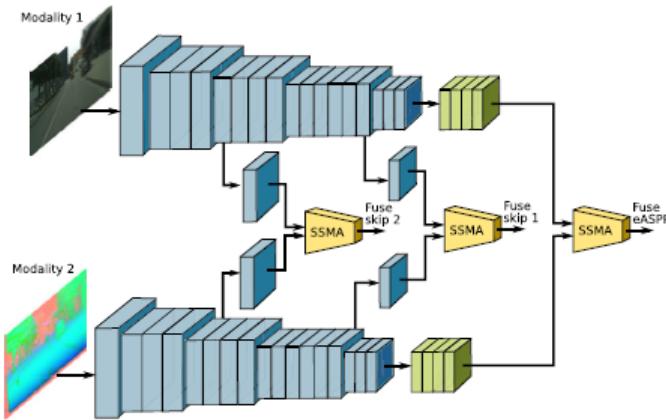


Figura 4.4: Arquitectura del encoder para segmentación multimodal. (Fuente: (Valada y col., 2019))

la red y las operaciones realizadas por la misma. La técnica utilizada por AdapNet++ se basa en una poda holística a lo largo de toda la red que no es afectado por las conexiones de cortocircuito o los skips, más detalles sobre el proceso pueden encontrarse en Valada y col., 2019.

4.2. Bloque SSMA

Hasta ahora se ha explicado cuáles son los componentes y su funcionamiento para una red AdapNet++ utilizada únicamente para imágenes de una modalidad en concreto, esto es, un único tipo de entrada. Sin embargo, los autores de esta red pretenden aportar varios tipos de entradas a la red para poder extraer de cada uno de ellos la información más relevante.

Se expondrá el caso en el que se tienen entradas de dos modalidades distintas, una imagen en RGB y un mapa de profundidad. Para poder extraer información de cada una de las imágenes se utilizan dos encoders de red AdpaNet++, uno para cada modalidad, y posteriormente se fusionan los resultados de ambos gracias al bloque SSMA que se explicará en este apartado. Todo esto puede observarse en la figura 4.4.

El bloque SSMA (figura 4.5) está compuesto únicamente por capas

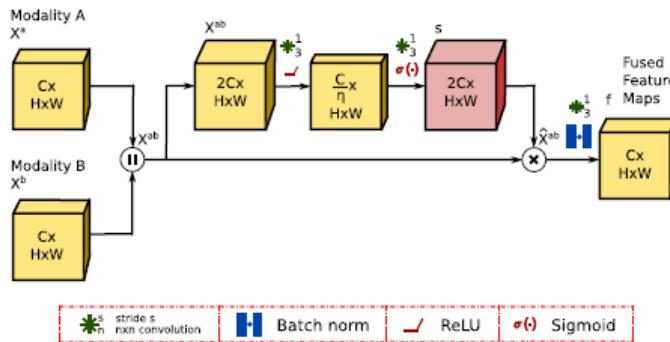


Figura 4.5: Arquitectura del bloque SSMA. (Fuente: (Valada y col., 2019))

convolucionales y es capaz de enfatizar la información más relevante dependiendo del contexto. En el caso de este trabajo, esto puede suponer que, en una escena mal iluminada, la información proporcionada por la imagen térmica se tome en cuenta antes que la de la imagen RGB. El proceso seguido dentro del bloque es el siguiente:

1. Se concatenan (se apilan) los mapas de ambas modalidades.
2. Al resultado del paso anterior se le aplican dos convoluciones 3x3.
3. Al resultado del paso 2 (cubo rojo en la imagen) se le aplica el producto Hadamard con el mapa obtenido del paso 1, esto seguido de una convolución 3x3 y una normalización por lotes.

Como puede observarse en la figura 4.4 el bloque SSMA se aplica en tres puntos, en las salidas skip1 y skip 2 explicadas previamente,y a la salida del módulo eASPP. En los tres puntos su función es fusionar dos mapas de características antes de enviarlos al decoder.

Este bloque SSMA se basa en el bloque CMoDE desarrollado por los mismos autores (Valada y col., 2017). En este caso, el bloque SSMA se utiliza únicamente para fusionar dos clases, aunque cómo se indica en el artículo mencionado, el bloque CMoDE del cual parte puede emplearse para fusionar mapas de características de todas las modalidades que se desee. Características más detalladas de cada bloque así como sus expresiones matemáticas

pueden encontrarse en el articulo citado previamente en este párrafo y en Valada y col., 2019.

Capítulo 5

Transferencia de conocimiento con AdapNet++

5.1. Metodología

En este capítulo se explicará el proceso de re-entrenamiento seguido en las dos últimas etapas del trabajo. La primera de ellas consiste en introducir una clase adicional en alguno de los datasets utilizado por los autores y re-entrenar la red para que identifique esta nueva clase. La segunda consiste en re-entrenar la red con las imágenes del dataset UMA-SAR. En ambos casos se aplicarán técnicas de transfer learning. En esta etapa del trabajo también ha sido necesario etiquetar las imágenes que se han usado para entrenar la red además de realizar una serie de tratamientos previos que también se comentarán en este apartado. El proceso seguido durante esta fase se ilustra en la figura 5.1.



Figura 5.1: Metodología seguida durante el proceso de transferencia de conocimiento

5.2. Transferencia de conocimiento utilizando el dataset de Cityscapes

La segunda etapa de este trabajo ha consistido en hacer un re-entrenamiento de la red sobre uno de los datasets utilizados por los autores. Se ha escogido el dataset de Cityscapes para 12 clases y se ha re-entrenado con el objetivo de que sea capaz de diferenciar una clase más. El objetivo de esta etapa es poder comparar los resultados del experimento con los obtenidos por los autores en unas condiciones similares. La configuración de la red para este dataset diferencia entre las siguientes clases:

1. Cielo
2. Edificio
3. Carretera
4. Acera
5. Valla
6. Vegetación
7. Poste
8. Coche/Camión/Autobús
9. Señal de tráfico
10. Peatón

11. Rider/Bicicleta/Moto

De entre estas clases, se ha elegido una que fuese fácilmente divisible, para evitar la necesidad de introducir un elemento completamente nuevo para la red. Teniendo esto en cuenta se ha dividido la clase 10 entre coches, por un lado, y autobús y camión por otro.

5.2.1. Etiquetado de las imágenes del dataset de Cityscapes

Antes de comenzar el proceso de re-entrenamiento es necesario volver a etiquetar las imágenes introduciendo la nueva clase, ya que en las máscaras de las cuales se parte, coches autobuses y camiones poseen la misma etiqueta. Una de las razones por las cuales se ha escogido Cityscapes para realizar esta fase del trabajo es por la facilidad que ofrece este dataset a la hora de etiquetar y re-etiquetar.

Cityscapes no solo cuenta con diferentes tipos de imágenes para poder entrenar una red neuronal, sino que, además, cuenta con un repositorio de github en el que pueden encontrarse varios scripts. Algunos de ellos están destinados a la creación de las máscaras con las etiquetas necesarias. Esto facilita bastante la labor de etiquetado puesto que solo hace falta aprender cómo funcionan estos scripts para luego usarlos de acuerdo al fin que se pretenda alcanzar. Para poder generar las máscaras el proceso a seguir es el siguiente:

1. Una vez descargado el repositorio de github se abre el archivo labels.py y se especifican las clases que se quiere etiquetar y un número para cada una, en caso de escoger 255, el objeto se etiquetará como conjunto vacío (posteriormente deberá de cambiarse a 0).
2. El siguiente paso consiste en abrir el archivo createTrainIdLabelImg.py, especificar donde quieren guardarse las máscaras y ejecutar el script.

Todo esto posible gracias a que los autores de Cityscapes no proporcionan las máscaras directamente, en su lugar, la descarga cuenta con varios

ficheros *.json*, uno para cada imagen. Estos ficheros contienen los polígonos de las imágenes, que no son más que los contornos de cada uno de los objetos etiquetados. Al ejecutar el archivo *createTrainIdLabelImg.py* se asigna cada polígono a una de las clases indicadas en el script *labels.py*. Se ha extraído un total de 50 imágenes etiquetadas.

Preparación de las imágenes y aumento de datos

Puesto que no se han utilizado muchas imágenes para realizar este re-entrenamiento se ha decidido aplicar técnicas de aumento de datos para aumentar la diversidad del conjunto de imágenes con el que se cuenta. Se han aplicado tres transformaciones, las mismas que los autores de AdapNet++ utilizan sobre sus datasets. Haciendo esto el dataset de 50 imágenes pasa a ser de 200. Las transformaciones se presentan en la figura 5.2 y son:

- **Random crop:** Se selecciona una región de tamaño aleatorio de la imagen y se re-escala esta región para que tenga el mismo tamaño que la imagen original. Esta operación implica el uso de imágenes con cambios de escala en los objetos presentes en el dataset.
- **Random scale:** Es igual que la técnica anterior pero la región siempre se toma sobre el centro de la imagen. El resultado es un zoom aleatorio de la imagen.
- **Flip horizontal:** Consiste en una simetría horizontal de la imagen

Por último, cabe mencionar que la técnica de aumento de datos puede aplicarse de 2 maneras, online y offline. Hacerlo online implica que, durante el proceso de entrenamiento, cada vez que se le pasa un lote de imágenes a la red, se aplican las transformaciones sobre ese lote en ese momento. Esto es muy adecuado cuando se parte de datasets muy grandes ya que el resultado de aplicarles aumento de datos ocuparía mucha memoria. Si se hace online, las imágenes transformadas solo existen durante la iteración en la que se crean. Por otro lado, el aumento de datos offline, implica realizar las transformaciones y guardar los resultados en memoria, esto suele hacerse cuando el dataset no es muy grande y es la opción que se ha escogido en este trabajo. Esta segunda opción también permite comparar los resultados obtenidos al



Figura 5.2: Ejemplo de aplicación de aumento de datos. a) Imagen original. b) Scale. c) Flip horizontal. d) Crop

entrenar distintas redes o la misma red en diferentes condiciones de entrenamiento ya que se utilizan exactamente las mismas imágenes durante dicho entrenamiento.

Preparación del las imágenes

Antes de comenzar el proceso de re-entrenamiento, es necesario preparar las imágenes que se van a utilizar. AdapNet++ acepta como entrada estas mismas imágenes, pero en formato tfrecord. Este formato permite almacenar una secuencia de cadenas binarias, ofreciendo ventajas en cuanto espacio de almacenamiento y velocidad de carga de los datos, pues permite cargar únicamente el lote que se utilice en cada momento. Esto implica realizar dos transformaciones previas al comienzo del re-entrenamiento:

- Convertir todas las imágenes que se vayan a utilizar a formato tfrecord. Para esto, primero hay que definir que estructura tienen dichos datos. No se profundizará en esta parte ya que los autores de AdapNet++ proporcionan un fichero para hacer el cambio.
- Además de lo anterior, el modelo entrando por los autores no acepta el formato disparidad de los mapas de profundidad. Antes hay que aplicarles un mapa de color y una técnica para mejorar su calidad que se explicará en este apartado.

De las dos transformaciones, la primera que hay que realizar es con-

vertir a mapa de color las imágenes en formato disparidad, esto se debe a que el fichero tfrecord debe de construirse con los archivos con los que se vaya a entrenar la red. El formato disparidad se construye a partir de dos imágenes en escala de gris, evaluando la diferencia de posición en las dos imágenes del mismo elemento. Sabiendo este valor y las características de la cámara utilizada se puede calcular la profundidad de un elemento según la ecuación:

$$\text{profundidad} = \frac{D * D_f}{disp} \quad (5.1)$$

- D: Distancia entre las cámaras utilizadas.
- Df: Distancia focal de las cámaras.
- Disp: Valor dispariad del elemento.

De este modo, la imagen disparidad se representa como una escala de grises, cuanto más oscuro sea un píxel más lejos está (menos diferencia en píxeles hay entre las posiciones del elemento al que pertenece en ambas imágenes). El formato JET no es más que la misma imagen a la cual se le ha aplicado un mapa de color tipo jet, dando como resultado una imagen RGB en la que aparecen en rojo los píxeles más cercanos y en azul los más lejanos.

Cómo puede observarse en la imagen tipo JET de la figura 5.3 hay algunos píxeles en negro, correspondientes a valores erróneos de la imagen disparidad, normalmente puntos que se ven desde una cámara, pero no desde la otra. Para solucionar esto se aplica la técnica Image Processing for Basic Depth Completion (Ku y col., 2018). El autor cuenta con un repositorio de github que puede descargarse y utilizar sus scripts para llenar las imágenes disparidad. El resultado obtenido tras aplicar esta técnica en las imágenes JET se observa en la figura 5.4.

Una vez se tienen las imágenes disparidad rellenas, se puede aplicar el mapa de color JET para obtener los mapas de profundidad que AdapNet++ acepta como entrada. El proceso para realizar la transformación se puede encontrar en el código 5.1.

En este código primero se selecciona el mapa de color tipo JET, se lee la imagen y su valor máximo invertido (variable norm). Esto se hace para

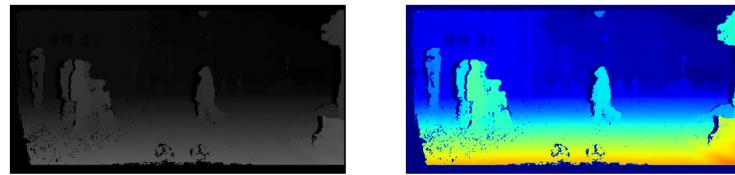


Figura 5.3: Ejemplos de mapas de profundidad en formato disparity (izquierda) y JET (derecha). (Fuente: Elaboración propia).



Figura 5.4: Comparación entre dos mapas de profundidad en formato JET. Sin aplicar la técnica de relleno(izquierda). Aplicando la técnica de relleno(derecha). (Fuente: Elaboración propia)

Código 5.1: Aplicación de mapa de color tipo JET

```
import matplotlib.pyplot as plt
import numpy as np

cmap = plt.get_cmap('jet')
image = plt.imread(Direccion de la imagen)
im=np.array(image)
norm=1/np.amax(image)
imagen_jet=cmap(im*norm)
imagen_jet=np.uint8(imagen_jet*255)
```

obtener un mapa de color normalizado, de no hacerlo, hay imágenes que están casi por completo en azul (todos los píxeles muy lejos). Para comprobar que el mapa de color obtenido era el adecuado, se ha escogido una de las imágenes que los autores proporcionan como ejemplo de segmentación, se ha buscado la imagen disparity correspondiente y se le ha aplicado la transformación que puede verse en el código 1, el resultado era el mismo que el que los autores daban como ejemplo para imagen JET del mapa de profundidad. En la figura 5.3 se pueden ver tanto la imagen disparidad como JET.

Una vez se tienen las imágenes JET y se rellenan los píxeles incorrectos, se genera el fichero tfrecord con las imágenes RGB, JET llenas y las etiquetas. Para conseguirlo se siguen los siguientes pasos:

1. Se construye un fichero .txt en el que figuren, por filas, las direcciones de formato RGB, mapa de profundidad y etiquetas de cada imagen. Esto dará como resultado un fichero con la siguiente estructura.

Dirección_rgb.png Dirección_JET.png Dirección_etiquetas.png.

Para facilitar esta labor se ha elaborado un script de Python que crea una lista con las direcciones de cada imagen y las escribe en un bloc de notas siguiendo el formato explicado.

2. Los autores de AdapNet++ proporcionan un script para pasar las imágenes a formato tfrecord por lo que no es necesario hacer uno nuevo. Este script tiene como entrada el bloc de notas del paso 1. Para ejecutarlo hay que abrir un terminal de Python moverse a la carpeta donde se encuentra el script y ejecutar el comando:

```
python convert_to_tfrecords.py {file localización_bloc_de_notas.txt  
--record nombre_archivo_tfrecord}
```

5.2.2. Proceso de re-entrenamiento sobre el dataset de Cityscapes

En muchas ocasiones no es necesario entrenar una red, principalmente si se parte de una red ya entrenada es más adecuado aprovechar su conocimiento previo y entrenar de nuevo con las nuevas clases, necesitándose menos

imágenes etiquetadas para realizar el re-entrenamiento, de esta manera se aplica la técnica de transferencia de conocimiento.

Para realizar el proceso lo primero que se ha hecho es construir una red AdapNet ++ bimodal con los pesos aportados por los autores para el dataset de Cityscapes con 12 clases, a partir de aquí se denominará este modelo como modelo de Cityscapes original. Como el cambio en cuanto a número de clases es mínimo y se pretende aprovechar el conocimiento que ya tiene la red, solo se han re-entrenado los pesos de las dos últimas capas, conv5 y conv78, los números no tienen que ver con su posición dentro de la red, son simplemente los nombres que los autores les han dado, de hecho, conv5 es la última capa del decoder y conv78 la penúltima.

Las dos capas mencionadas en el párrafo anterior no solo se han re-entrenado si no que se ha modificado su estructura. Esto es porque estas dos capas deben retirarse cada vez que se pretenda modificar el número de clases a detectar por la red. El motivo es que la estructura de ambas depende de este número. Por ejemplo, conv5, al ser la última capa del decoder, debe de proporcionar una salida cuyas dimensiones de altura y anchura sean iguales a las de la imagen de entrada de la red, y su profundidad debe de coincidir con el número de clases, esto último para poder otorgar a cada pixel la probabilidad de pertenecer a cada una de las clases. Si el número de clases cambia, también deben de cambiar las dimensiones de esta capa por lo que no pueden cargarse sus pesos para re-entrenar la red ya que la morfología de esta es diferente. Algo similar ocurre con la capa conv78. Para conseguir lo explicado se presenta un fragmento de código en la figura 5.5.

```

inspect_list = tf.train.list_variables(path)
names=[]
for item in inspect_list:
    names.append(item[0])
for var in import_variables:
    if 'conv78/' in var.name or 'conv5/' in var.name or 'conv911/' in var.name or 'conv912/' in var.name:
        print('capa no cargada: ',var.name)
    else:
        temp=var.name.split(':')
        if temp[0] in names:
            initialize_variables[temp[0]]=var
#Se dejan solo las capas conv5 y conv78 entrenables ya que son las que cambian al modificar el numero de clases
trainable_collection=tf.get_collection_ref(tf.GraphKeys.TRAINABLE_VARIABLES)
for i in range(43):
    trainable_collection.pop(0)

```

Figura 5.5: Fragmento de código utilizado para no cargar todas las capas.
(Fuente: Elaboración propia).

Lo primero que se hace es guardar en la variable `inspect_list` la lista con todas las capas de la red cargada. Posteriormente se crea una lista con solo los nombres de estas capas. Hecho esto se procede a cargar las capas. Primero se excluyen las convoluciones 78,5,911 y 912. Todas ellas dependen del número de clases y su morfología cambia al introducir una clase nueva por lo que, si se intentan cargar, el programa falla. Lo siguiente es buscar dentro de las variables del gráfico creado al construir la red(`import_variables`), las variables correspondientes en la red que se pretende cargar. Dicho con otras palabras, al construir la red se crean las capas de la misma, cada una con su nombre. Dentro de la red que se pretende cargar existen las mismas capas con el mismo nombre, pero entrenadas. Además, hay otras variables referentes al entrenamiento que no se encuentran en la red construida. Por este motivo hay que buscar los nombres que coincidan en ambas y cargar esas capas. Por último, se asigna la lista `trainable_collection` a las capas entrenables de la red y se extraen todas a excepción de las convoluciones 5 y 78, que empiezan en la posición 43 de dicha lista. Esto puede hacerse con todas las capas que se quiera.

Al no iniciar estas capas con pesos re-entrenados, estas se inician, por defecto, con pesos aleatorios. El motivo para no iniciarlos con valor nulo es que, en este último caso, la convergencia de la red es más lenta. El valor de los pesos se guardará cada diez iteraciones. Los resultados obtenidos durante este proceso de transferencia de conocimiento se presentan en el siguiente capítulo.

5.3. Transferencia de conocimiento utilizando el dataset UMA-SAR

En este apartado se explicará como se ha llevado a cabo el proceso de transferencia de conocimiento utilizando el dataset UMA-SAR(Morales y col., 2021). Se empezará por exponer la metodología seguida para etiquetar las imágenes así como los criterios seguidos para escogerlas.

5.3.1. Etiquetado y tratamiento de las imágenes del dataset UMA-SAR

La primera fase, antes de poder realizar cualquier proceso de reentrenamiento y al igual que en el capítulo anterior, es el etiquetado de las imágenes. En este caso no se cuenta con una herramienta tan cómoda como la que ofrece Cityscapes pero existen otras opciones para realizar esta tarea.

Se ha utilizado CVAT¹ (Computer Vision Anotation Tool) para completar la fase de etiquetado de imágenes. Se ha escogido utilizar la versión online ya que las categorías que se pretendía utilizar estaban contempladas por uno de los modelos cargados por defecto, además, como se verá más adelante, el modelo entrenado no era capaz de detectar las clases con precisión. Las categorías que van a etiquetarse son:

1. Personal de rescate.
2. Civiles.
3. Víctima yacente.
4. Zapatos.
5. Vehículo de recate.
6. Vehículo civil.
7. Escombro.

¹<https://cvat.org/>

Se han escogido estas categorías pues se consideran las más importantes para el robot de rescate a la hora de detectar víctimas y personal de rescate en entornos de catástrofe. Una vez seleccionada la versión a utilizar y las categorías a detectar, se eligen las imágenes que se pretende etiquetar y se crea un proyecto nuevo con esas imágenes. Las imágenes se escogieron de los datasets correspondientes a 2019². A la hora de escoger estas imágenes se ha buscado que estuviesen presentes todas las clases mencionadas. Para facilitar la labor de etiquetado se ha utilizado una tablet de sobremesa Wacom Cintiq. En la figura 5.6 se presenta una imagen etiquetada utilizando CVAT.

Antes de comenzar el proceso de etiquetado, ha sido necesario modificar ligeramente las imágenes RGB del dataset ya que uno de los problemas que presentan las imágenes térmicas respecto a las RGB, es que las características de las respectivas cámaras no son las mismas (Morales y col., 2021). El motivo principal es que AdapNet++ bimodal acepta 2 imágenes como entrada, una de cada modalidad, pero una única imagen etiquetada que debe corresponder con los elementos presentes en ambas imágenes. Las transformaciones son:

- Giro de 3 grados en sentido horario.
- Se aplica una escala de 1.34.
- Crop de la imagen escalada entre los pixeles [127,840] en el eje x y [120,696] en el eje y.
- Se devuelve la imagen RGB al tamaño original.

Con esto se observa que la mayoría de las imágenes coinciden, sin embargo, hay otras en las que los objetos aparecen ligeramente desplazados. Este desplazamiento suele ser pequeño por lo que se ha dado esta transformación por buena.

El proceso que se ha seguido para etiquetar las imágenes con CVAT se detalla a continuación:

²<https://www.uma.es/robotics-and-mechatronics/cms/menu/robotica-y-mecatronica/datasets/>



Figura 5.6: Ejemplo de imagen etiquetada con CVAT. (Fuente: Elaboración propia).

1. El primer paso es crear una tarea en CVAT y cargar las imágenes, así como establecer las etiquetas (Figura 5.7).
2. Hecho esto se entra en la tarea. Pueden realizarse varios tipos de etiquetado, ya sea con cajas para tareas de detección, o con polígonos de N lados para segmentación. Una vez abierta la tarea se tiene la interfaz mostrada en la figura 5.8.

A la izquierda se tienen varios botones, cada uno de ellos se puede utilizar para etiquetar de una manera diferente, los más importantes aparecen dentro de un cuadrado rojo. El primero de ellos permite usar modelos de redes neuronales artificiales cargados para etiquetar algunas clases. Esto se observa mejor en la figura 5.9.

Como puede verse, la pestaña Model permite seleccionar un modelo, en los cuadros de texto inferiores se asocia una categoría del modelo elegido a una de las categorías que se pretende etiquetar. Por último, se tiene la pestaña Interactor, aquí se seleccionan 4 puntos de un objeto, se le asocia una clase, y CVAT dibuja su silueta.

El otro botón es el utilizado para hacer polígonos, este sirve para etiquetar objetos que no estén contemplados en los modelos o para realizar ajustes en los objetos etiquetados por estos mismos modelos. De entre las opciones vistas en la figura 5.10, Shape etiqueta el objeto solo en la imagen presente mientras que Track lo etiqueta en todas, esto último puede ser útil al etiquetar un video pues puede haber varios frames en los que un objeto apenas varíe su posición. No es necesario especificar el número de puntos pues estos se van creando conforme se dibuja la forma del objeto.

Estas son las opciones más utilizadas en segmentación, hay otras opciones importantes como la pestaña trackers de la figura 5.9 que permite, en un video, que una etiqueta se mueva con el objeto, o el rectángulo que hay justo encima del botón para dibujar polígonos. Este rectángulo se utiliza para etiquetar imágenes destinadas a labores de detección, como por ejemplo la red YOLO (Redmon y col., 2016).

Por último, a la derecha, en el rectángulo azul, se encuentran todos los polígonos dibujados y su correspondiente etiqueta. En medio queda la imagen con las etiquetas que se van añadiendo (figura 5.7).

3. Para terminar, se mencionarán algunos “trucos” que facilitan enormemente la labor de etiquetado. El primero de ellos consiste en pulsar control para que aparezcan los puntos que forman los polígonos, como puede verse en la figura 5.11.

La ventaja que ofrece la opción b de la figura 5.11 aparece al etiquetar objetos que están juntos ya que hay que tener especial cuidado a la hora de respetar los límites de cada objeto, un píxel no puede pertenecer a 2 clases. Con la opción b basta con pulsar sobre los vértices para seguir perfectamente la silueta del objeto contiguo. Además, esto ofrece una segunda ventaja que puede verse en la figura 5.12. En la imagen (a) de dicha figura se selecciona el punto inferior izquierdo, el primer punto en el que limitan dos clases, persona y vehículo, en la imagen (b) se ha pinchado sobre el siguiente punto, que aparece en morado, en la imagen (c) se pincha sobre el último punto de la silueta y todos los puntos situados entre el morado de la imagen (b) y este último punto quedan automáticamente seleccionados. Esto permite ahorrar mucho tiempo.

4. Para terminar, se guarda el trabajo y se exporta el dataset en el formato deseado.

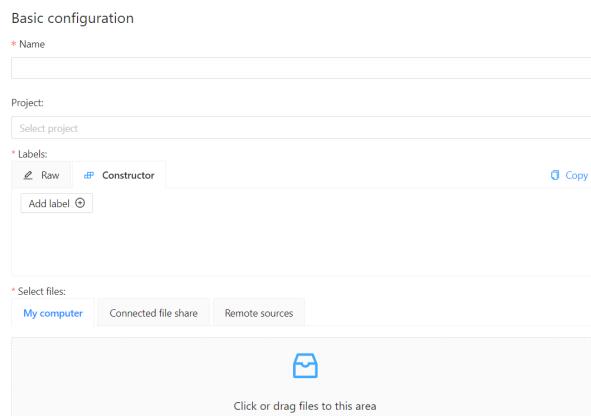


Figura 5.7: Creación de una tarea en CVAT. (Fuente: Elaboración propia)



Figura 5.8: . Interfaz de CVAT. (Fuente: Elaboración propia).

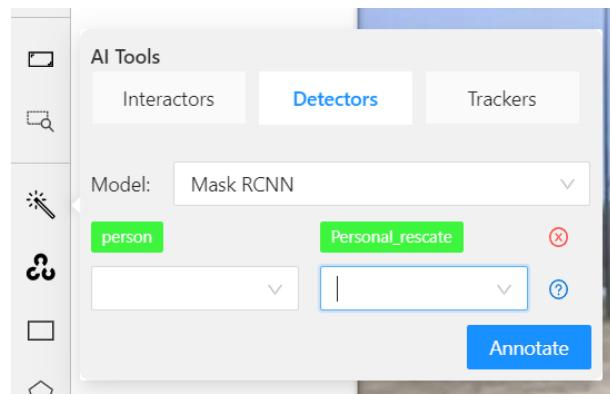


Figura 5.9: Botón para cargar modelos en CVAT. (Fuente: Elaboración propia).

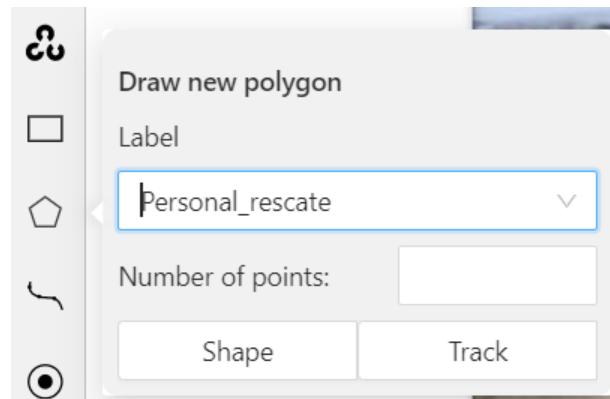


Figura 5.10: Botón para dibujar polígonos en CVAT. (Fuente: Elaboración propia).



Figura 5.11: Misma imagen en CVAT. Sin usar tecla control (a). Usando tecla control (b).

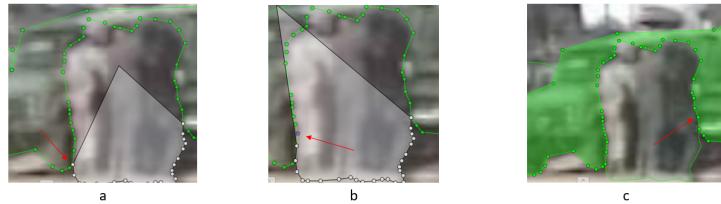


Figura 5.12: Etiquetado rápido. (Fuente: Elaboración propia)

Una vez se han etiquetado todas las imágenes, se descarga el dataset resultante en el formato deseado. Se ha escogido el formato Segmentation Mask ya que, utilizando Python, era sencillo transformar las imágenes etiquetadas al formato de escala de grises. Obtenidas las imágenes en escala de grises simplemente queda asociar cada nivel de gris a una clase. Este proceso es bastante rápido ya que el archivo descargado de CVAT especifica el color que da a cada clase. Si se utiliza la librería de open cv para pasar estas imágenes a escala de grises puede determinarse que nivel de gris se asocia a cada color mediante la fórmula:

$$Y = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B \quad (5.2)$$

En esta fórmula las letras R,G y B hacen referencia a los valores de la imagen RGB que se transforma a escala de grises. De este modo, si el color de la clase escombro se asocia al nivel de gris 165, basta con ejecutar el comando `Escombro=im==165` seguido de `Escombro=Escombro*7`.

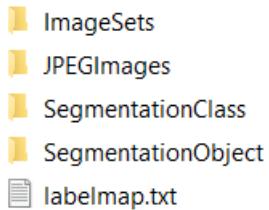


Figura 5.13: Estructura del archivo a importar en CVAT. (Fuente: Elaboración propia).

Esto en caso de que escombro se asigne a la clase 7. El procedimiento descrito es más sencillo y funciona más rápido que construir las imágenes etiquetadas directamente a partir de los archivos proporcionados por CVAT.

Cómo las etiquetas de las imágenes térmicas deben de coincidir con las de las RGB, se han cargado las etiquetas de las segundas sobre el dataset de imágenes térmicas. Par hacer esto hay que crear un archivo con la misma estructura, crear una tarea como se explicó previamente y por último importar el archivo creado. La estructura de este archivo se puede ver en la figura 5.13. El contenido de cada carpeta es:

1. **ImageSets:** Bloc de notas que contiene el nombre de las imágenes del dataset. Es importante que los nombres coincidan con los nombres de las imágenes de la tarea.
2. **JPEGImages:** Contiene las imágenes etiquetadas o que se pretenden etiquetar. Los nombres deben coincidir con los de ImageSets.
3. **SegmentationClass y SegmentationObject:** Contienen las etiquetas para segmentación semántica y de instancias respectivamente. Al igual que antes los nombres deben de coincidir con los de las otras carpetas.
4. **Labelmap:** Contiene los colores asociados a cada clase.

Para cargar las etiquetas de las imágenes RGB en la tarea creada para las imágenes térmicas, basta con copiar el archivo con las etiquetas de las imágenes RGB descargado (tiene la misma estructura que el de la figura

5.13), la carpeta JPEGImages se descarga de manera opcional. Si se tiene esta carpeta, se sustituyen las imágenes de su interior por las imágenes térmicas, que deben de tener los mismos nombres que las imágenes RGB de la tarea descargada. A la hora de crear la tarea en CVAT, las imágenes térmicas que se carguen, también tendrán que tener los mismos nombres que dichas imágenes RGB. Una vez hecho todo esto, se selecciona la opción Upload Annotations dentro de la tarea creada. La importancia de que los nombres coincidan reside en que para cada imagen se cargará la imagen etiquetada cuyo nombre coincide con la primera. Por ejemplo, si se carga una imagen con el nombre Imagen1.png, a la hora de subir las etiquetas a CVAT, se asignará a dicha la imagen la etiqueta de las carpetas SegmentationClass y SegmentationObject que se llame Imagen1.png.

5.3.2. Experimentos previos al re-entrenamiento

Antes de empezar con el proceso de re-entrenamiento, se ha examinado la respuesta de AdapNet++ ante el dataset escogido para dicho proceso. Los objetivos de esto son, comprobar si puede utilizarse algunos de los modelos ya entrenados de AdapNet++ como punto de partida para el re-entrenamiento, y analizar la respuesta de la red ante imágenes térmicas antes de realizar dicho proceso.

Para realizar estos experimentos se han seleccionado dos modelos de AdapNet++, el modelo de Cityscapes original (12 clases) y Freiburg-forest. El primero de ellos se ha escogido porque contiene las categorías de persona y vehículo, estas dos clases se dividen como se indica a continuación:

- Persona: Se divide en civil, personal de rescate y víctima.
- Vehículo: Se divide en vehículo de rescate y vehículo civil.

Si el modelo es capaz de detectar estas dos clases en el dataset UMA-SAR, el proceso de re-entrenamiento será parecido al descrito en el capítulo anterior. Aun así, hay que tener en cuenta que las imágenes de las jornadas de rescate (UMA-SAR) se tomaron en el campo mientras que las de Cityscapes representan paisajes urbanos.

El segundo de los modelos, aunque no contenga ninguna de las clases a etiquetar, tiene como objetivo la segmentación de imágenes en entornos naturales, concretamente un bosque, por lo que resulta interesante comprobar si es capaz de detectar las mismas categorías en un entorno natural distinto. Si es capaz de hacerlo con suficiente precisión, podría re-entrenarse con los nuevos ejemplos para que detecte las nuevas categorías, en una situación parecida a la que se tiene con el dataset de Cityscapes.

También se pretende comprobar si los modelos de AdapNet++ entrenados con mapas de profundidad pueden utilizarse para re-entrenar otro modelo de AdapNet++ con imágenes térmicas.

En la figura 5.14 se presenta el resultado obtenido al realizar una predicción sobre imágenes del dataset UMA-SAR partiendo de un modelo de AdapNet++ que contiene los pesos del modelo de Cityscapes original (12 clases). En este caso se esperaba que, como el modelo ya se había entrenado para reconocer personas y vehículos, fuese capaz de reconocer al menos estas dos clases. Sin embargo, se observa en la primera de las imágenes que no es capaz de segmentar a ninguna de las personas, de hecho, les asigna la clase vehículo, en la segunda sí que segmenta correctamente la parte superior y los portones de la ambulancia, aunque la red es poco precisa en las partes que están más cerca del suelo.

Por último, y a diferencia de la primera imagen, en la tercera se logran clasificar correctamente algunos píxeles correspondientes a las personas de la parte izquierda. En las tres imágenes, la red etiqueta parte de la vegetación como la clase árbol, lo cual era de esperar, y también segmenta bien la mayor parte del cielo.

Por último, también se quería comprobar si la red era capaz de asignar la clase carretera al camino lo cual está claro que no ocurre.

En la figura 5.15 se observan los resultados obtenidos con el modelo de Freiburg-forest. En este caso se pretende comprobar si la red es capaz de segmentar correctamente los caminos y la vegetación. En la primera y tercera imagen, puede observarse que se etiqueta correctamente el camino, aunque también etiqueta como camino el cielo, algunos edificios, rocas y a las personas. En la segunda imagen prácticamente todo está etiquetado como vegetación. Se han realizado pruebas con imágenes parecidas a la primera y a

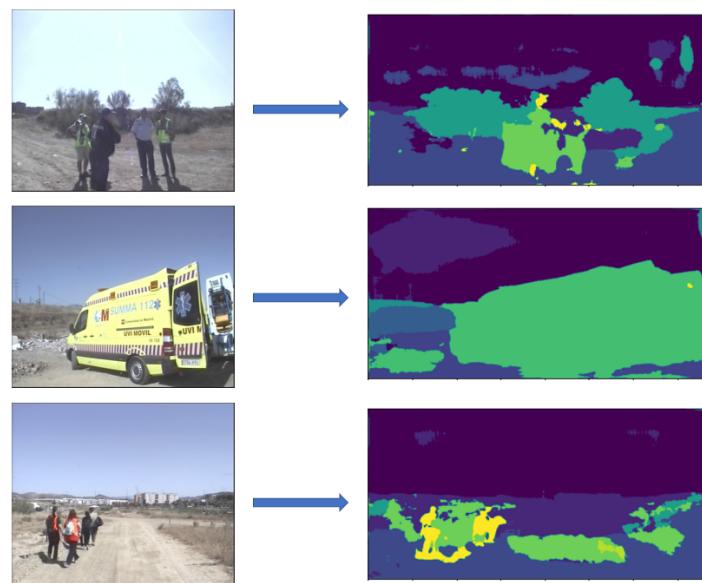


Figura 5.14: Predicciones efectuadas por AdapNet++ cargada con el modelo original de Cityscapes sobre imágenes RGB del dataset UMA-SAR. (Fuente: Elaboración propia).



Figura 5.15: Predicciones efectuadas por AdapNet++ cargada con el modelo original de Freiburg-forest sobre imágenes RGB del dataset UMA-SAR. (Fuente: Elaboración propia)

la tercera, pero en las que hubiese más vegetación, esto porque el dataset de Freiburg-forest cuenta con fotos de un bosque muy frondoso. Se ha pensado que, si el entorno de la imagen se parece un poco más al original, los resultados podrían mejorar, sin embargo, en este segundo caso, se detecta toda imagen como vegetación. De este modo puede comprobarse la importancia del contexto para ambos modelos, el primero, Cityscapes, funciona muy bien en entornos urbanos, de hecho, se ha probado con imágenes de ciudades escogidas de internet y funciona correctamente. Lo mismo ocurre con el modelo de Freiburg forest. Pero en cuanto se trata de detectar los mismos elementos en entornos distintos, el modelo falla. Dicho esto, el modelo de Cityscapes puede detectar algunos píxeles como persona o vehículo por lo que se ha escogido para realizar el proceso de transferencia de conocimiento.

Otra de las pruebas que se ha realizado es la de comprobar cómo funciona el modelo cuando su entrada es una imagen térmica. Para ello se parte de una red AdapNet++ entrenada con mapas de profundidad y se le

pasa como entrada una de las imágenes térmicas del dataset de las jornadas de rescate. Como el formato de estas imágenes es escala de grises, será necesario construir antes un mapa de color como se explicó en el capítulo anterior o pasar la escala de grises con un solo canal a la misma imagen, pero con 3 canales. Se volverán a utilizar los modelos de Cityscapes y de Freiburg-forest. Se presentan los resultados en las figuras 5.16 y 5.17.

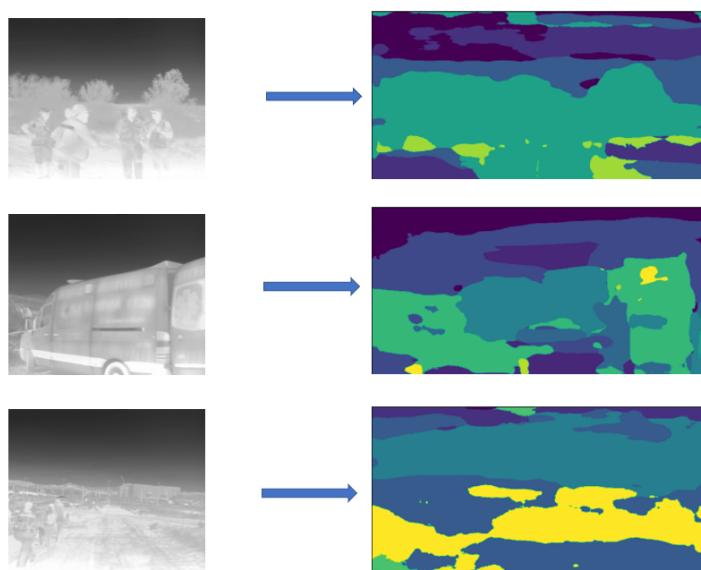


Figura 5.16: Predicciones efectuadas por AdapNet++ cargada con el modelo original de Cityscapes sobre imágenes térmicas del dataset UMA-SAR(Fuente:Elaboración propia)

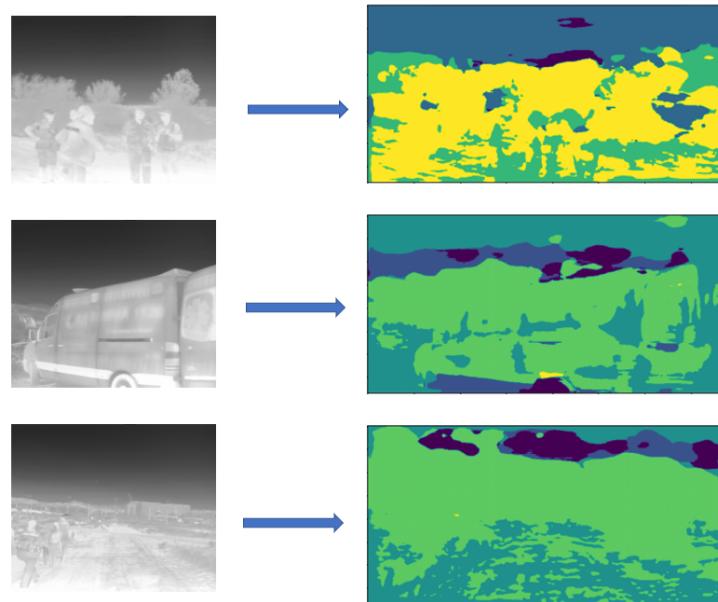


Figura 5.17: Predicciones efectuadas por AdapNet++ cargada con el modelo original de Freiburg sobre imágenes térmicas del dataset UMA-SAR. (Fuente: Elaboración propia)

Capítulo 6

Resultados

En este capítulo se expondrán los resultados obtenidos tras efectuar el proceso de transferencia de conocimiento sobre el dataset de Cityscapes y sobre el dataset UMA-SAR. Se presentarán imágenes como ejemplo de los resultados obtenidos además de métricas utilizadas normalmente en la segmentación de imágenes.

6.1. Métricas

Los diferentes métricas utilizadas para evaluar la respuesta de las CNN dependen de los objetivos que se persiga (clasificación, detección o segmentación). El índice MIoU es ampliamente utilizado para medir la precisión de redes neuronales artificiales utilizadas en labores de segmentación (Planche y Andres, 2019). En el caso de los resultados obtenidos sobre el dataset UMA-SAR se presentarán tablas indicando tanto el índice MIoU como el valor de la función de pérdida(entropía cruzada) obtenidos durante los experimentos realizados. Además de esto, se presentarán tablas en las que se indicará, para cada clase, el valor de falsos positivos, falsos negativos y positivos reales que se obtienen sobre las imágenes empleadas para el re-entrenamiento una vez acabado el mismo. No se ha empleado un conjunto de evaluación ya que el objetivo no es conseguir una red que sea capaz de segmentar las imágenes del dataset UMA-SAR sino estudiar como puede realizarse el proceso de

transferencia de conocimiento para este dataset. También se presenta para cada clase el índice IoU. Este índice junto con el índice F1 es ampliamente utilizado para evaluar la precisión de una red a la hora de segmentar cada clase(Planche y Andres, 2019). Se utilizará el índice IoU ya que en este caso no hay diferencia entre usar este o el índice F1.

Esta fase de transferencia de conocimiento se ha llevado a cabo en un equipo MSI GL65 Leopard con procesador intel core i7 y tarjeta gráfica Nvidia RTX 2070.

6.2. Resultados obtenidos sobre el dataset de Cityscapes

El objetivo de este proceso de transferencia de conocimiento es dotar a un modelo de AdapNet++ entrenado por los autores de la capacidad para detectar una clase más. En este caso se parte de un modelo que diferencia entre 12 clases y se introduce una clase más.

Los parámetros utilizados en esta fase han sido:

- Tamaño de lote=16.
- Tasa de aprendizaje= 0,01.
- 1300 iteraciones.
- Power=0,0001. Este parámetro se utiliza para actualizar los pesos en los cortocircuitos entre capas. Se ha utilizado el valor indicado por los autores de AdapNet++ en su artículo Valada y col., 2019.

Tras 1300 iteraciones, el valor de la pérdida es de 0,07 y el MIoU es de 69. Se presentan en las figuras 6.1 y 6.2 la evolución del valor de la función de pérdida sobre el conjunto de entrenamiento y la evolución de del MIoU sobre un conjunto de evaluación. El conjunto de evaluación está formado por imágenes etiquetadas que no son utilizadas para entrenar la red. El objetivo es poder comparar ambas métricas y determinar en que

momento del entrenamiento se alcanza el máximo desempeño de la red y detectar posibles casos de sobreajuste.

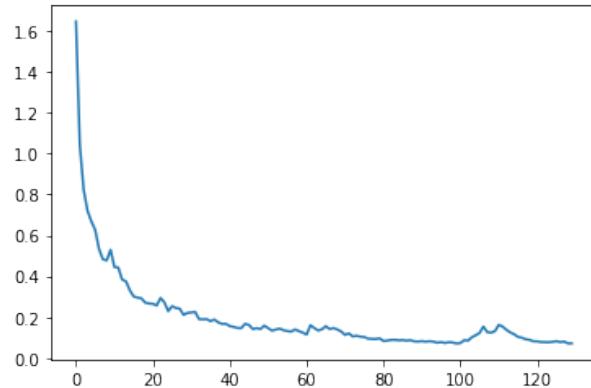


Figura 6.1: Evolución del valor de pérdida de la red durante el proceso de entrenamiento. Cada punto del eje horizontal corresponde a 10 iteraciones de entrenamiento. (Fuente: Elaboración propia).

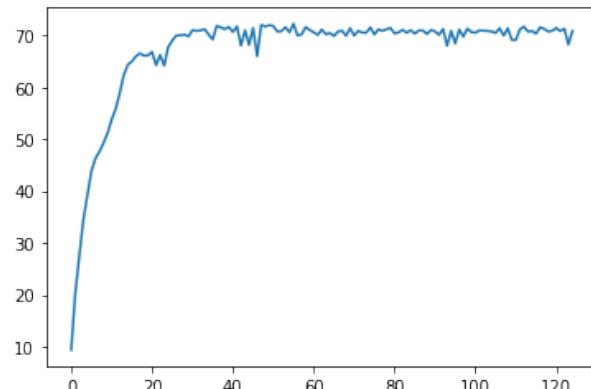


Figura 6.2: Evolución del valor del MIoU sobre el conjunto de evaluación para los pesos de la red calculados en distintas etapas del entrenamiento. (Fuente: Elaboración propia).

Al observar las gráficas puede comprobarse que mientras el valor de la pérdida baja prácticamente hasta el final del proceso de entrenamiento,

el MIoU, se estabiliza alrededor de la iteración 400. Esto puede deberse a que mientras que la pérdida se evalúa en base a la probabilidad asignada a cada píxel de pertenecer a una clase, el MIoU lo hace en base a píxeles correctamente clasificados.

Se presenta el resultado obtenido al realizar una predicción con los pesos obtenidos sobre varias imágenes distintas no utilizadas durante el proceso de entrenamiento (figuras 6.3 a 6.7).

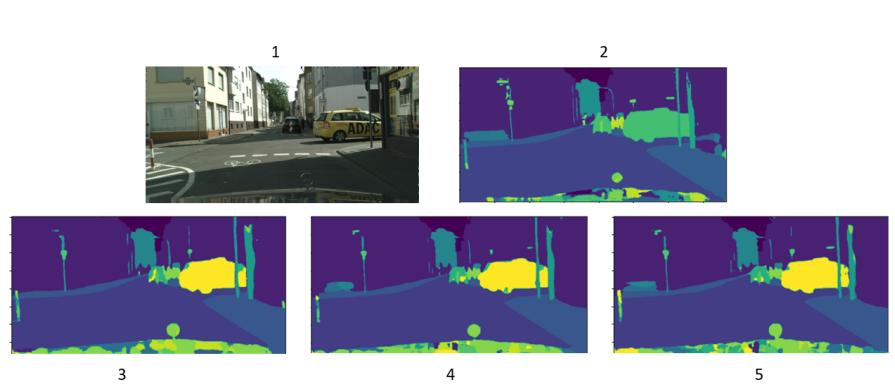


Figura 6.3: Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 calses checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia).

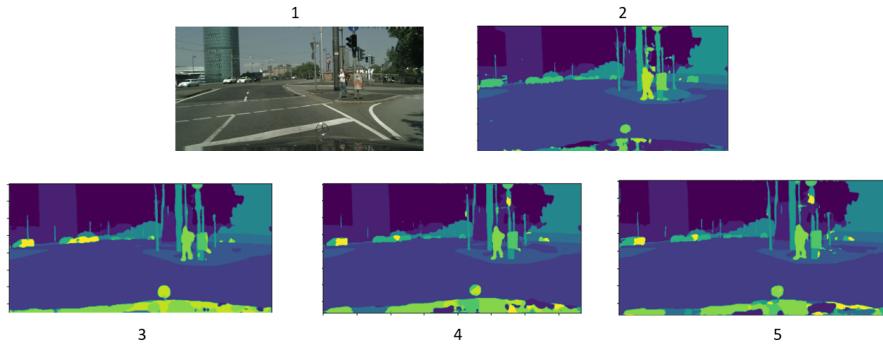


Figura 6.4: Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia).

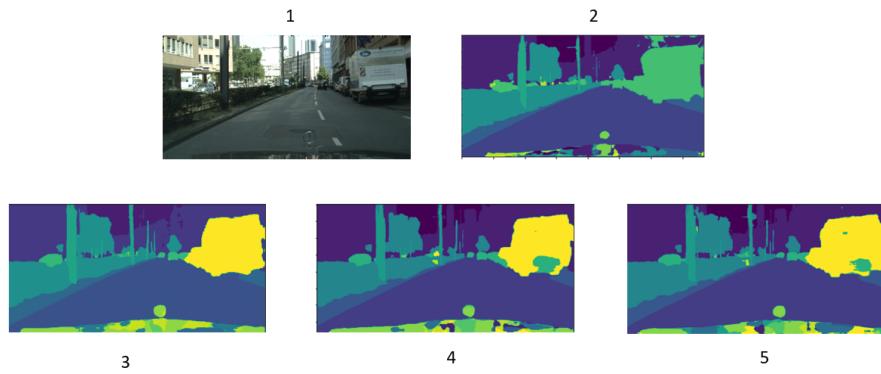


Figura 6.5: Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia).

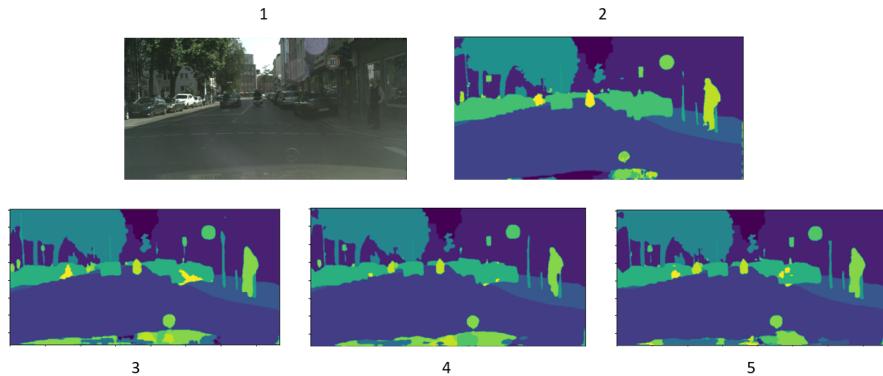


Figura 6.6: Predicciones hechas por AdapNet++ bimodal. Imagen original (1). Predicción con 12 clases (2). Predicción 13 clases checkpoint 409 (3). Predicción 13 clases checkpoint 809 (4). Predicción 13 clases checkpoint 1299 (5). (Fuente: Elaboración propia).

En la predicción de la figura 6.3 se observa que gran parte del taxi de la derecha se identifica como camión o autobús, esto se acentúa conforme avanza el entrenamiento. También se empieza clasificar parte del muro de la parte izquierda como árbol, cosa que también pasa con el modelo original entrenado por los autores de AdapNet++ (12 clases).

En la segunda de las predicciones, figura 6.4, se comprueba que la red clasifica ligeramente mejor los píxeles de los coches del fondo, al principio clasifica parte de estos como camión o autobús, esta situación mejora, pero no demasiado. En la tercera predicción, figura 6.5, se observa lo que es más interesante, pese a entrenar más la red, la clasificación del camión de la derecha empeora. Esto último hace sospechar que a partir de la iteración 400 se empieza a cometer sobreajuste.

En la última de las predicciones, en la figura 6.6, no se aprecian grandes diferencias a excepción de que los checkpoints correspondientes a etapas más avanzadas del entrenamiento segmentan algo mejor los coches, lo cual también ocurre en el resto de predicciones.

Si se analiza la gráfica del MIoU, puede concluirse que, a partir de las 400 iteraciones, se empieza a incurrir en un caso de sobreajuste. La red detecta muy bien las características particulares del conjunto de entrenamiento,

pero estas características no tienen por qué encontrarse fuera de este conjunto. Prueba de ello es que el camión se segmenta mejor en el checkpoint 409 que en el 1299.

También hay que tener en cuenta que, en todas las capas no modificadas, se conservan los pesos correspondientes al caso en el que camiones, coches y autobuses pertenecen a la misma categoría, de modo que no pueden contribuir a solucionar esto.

Otra de las cuestiones que pueden observarse es la peor definición de los bordes con respecto a la segmentación realizada por el modelo original. Esto se debe a que las dos últimas capas del decoder fueron entrenadas con más de 10 mil ejemplos por los autores del artículo, mientras que en este trabajo no se pasa de 200, por lo que el decoder del modelo original está mucho más ajustado. Incluso dentro de las predicciones resultantes del re-entrenamiento se observa en las imágenes 4 y 5 de la figura 6.6 un mejor contorno de las piernas de la persona de la derecha que en la imagen 3 de la misma figura.

Teniendo en cuenta todo lo anterior, se llega a la conclusión de que el mejor modelo es el correspondiente al checkpoint de la iteración 409 ya que en esta etapa del entrenamiento aun no se ha llegado al sobreajuste de la red. Sin embargo, la definición de los contornos es peor que en los modelos correspondientes a checkpoints de etapas posteriores. Con el objetivo de mejorar la definición de contornos se ha llevado a cabo una estrategia de entrenamiento ligeramente distinta y dividida en 2 fases.

1. La primera fase se re-entrenan las dos últimas capas la red hasta la iteración 409.
2. El segundo paso consiste en realizar 400 iteraciones más, pero esta vez re-entrenando todos los parámetros de la red y disminuyendo la tasa de aprendizaje.

Con esto se pretende al principio enseñar a las dos últimas capas del decoder. Como se pudo comprobar en el apartado anterior, el MIoU se estabilizaba entorno a la iteración 400 por lo que quitar 40 imágenes al dataset no debería de variar mucho su valor. Con el segundo paso se pretende

educar ligeramente al resto de capas de la red, ya que como se mencionó con anterioridad, estas conservan los pesos del modelo original. Se espera que con esta segunda estrategia mejore la precisión de la red y que esta no confunda clases que diferenciaba correctamente antes de realizar el re-entrenamiento. Para hacer esto se repite el proceso seguido en la figura 5.5 pero en vez de eliminar capas de la lista de capas entrenables, se añaden a esta con el comando.

```
trainable_collection.append(capa)
```

Para este segundo caso de re-entrenamiento se ha disminuido la tasa de aprendizaje puesto que lo único que se pretende es una pequeña adaptación del resto de la red a la nueva clase introducida. Además, como no se dispone de muchos ejemplos de entrenamiento no puede realizarse un entrenamiento demasiado exhaustivo. Aun con esto, la pérdida final de la red no dista mucho de la del caso anterior (0.07 en la primera estrategia, 0.12 realizando también un ajuste fino)

Los resultados obtenidos siguiendo esta nueva estrategia se presentan en la figura 6.7. Hay una ligera mejoría en cuanto a la clasificación de los coches se refiere, algunos que antes eran asignados a la clase camión o autobús ya no lo son. Sin embargo, se empiezan a cometer errores en otras zonas de la imagen. El MIoU aumenta hasta 71 en este caso, en el caso de no realizar un ajuste fino, es de 69.

Un tercer enfoque puede consistir en realizar un ajuste fino sobre el resultado obtenido al seguir la primera estrategia, pero utilizando para ello imágenes diferentes a los ejemplos de entrenamiento. Este enfoque no se ha aplicado ya que el objetivo principal es el de utilizar la red en aplicaciones de rescate, se ha dado por finalizado el re-entrenamiento en este punto con unos resultados satisfactorios. No obstante, es importante considerar esta opción si se desea mejorar el rendimiento de la red. Por último, con el objetivo de poder mejorar el MIoU, se ha decidido re-entrenar todo el decoder pero este valor a disminuido a 66,4.

Una de las preguntas surgidas al realizar todo este proceso era si se podía afinar aún más el proceso de entrenamiento disminuyendo la tasa de aprendizaje. Para comprobarlo se ha bajado su valor a 0,001. Sin embargo,

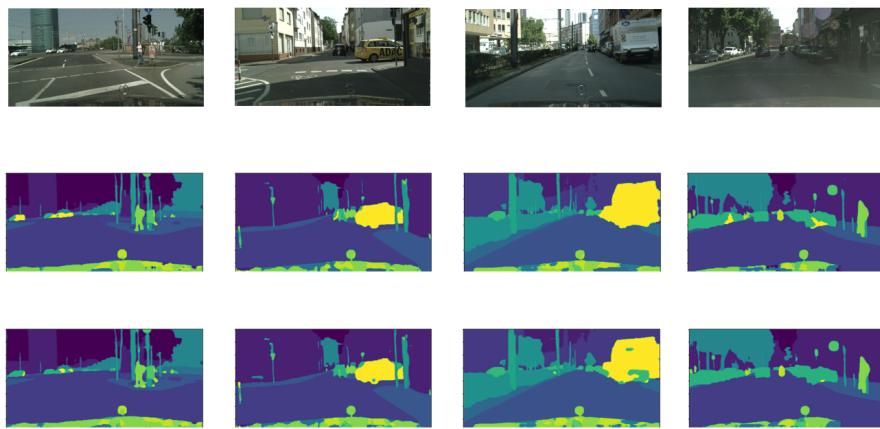


Figura 6.7: Resultados tras aplicar ajuste fino sobre el chekpoint 409.

en este último caso la función se estanca en un mínimo local entorno a un valor de 0,2 de la función de pérdida además de converger más lento.

6.3. Resultados obtenidos sobre el dataset UMA-SAR

En este apartado se presentan los resultados obtenidos tras realizar el proceso de transferencia de conocimiento sobre el dataset UMA-SAR. Se han utilizado varios modelos de partida sobre cada modalidad, concretamente, se ha partido del modelo de Cityscapes original entrenado por los autores (12 clases) pues en el capítulo anterior se pudo comprobar que era capaz de identificar algunos elementos, se utilizarán sus dos versiones (una entrenada con imágenes RGB y otra con mapas de profundidad). También se ha partido de un modelo de AdapNet++ que aportan los autores de la red como modelo más general para entrenar la red desde cero para cualquier aplicación. Estos modelos se han utilizado tanto sobre imágenes térmicas como RGB con el objetivo de comparar los resultados obtenidos y determinar que modelos utilizar a la hora de re-entrenar una red AdapNet++ bimodal.

6.3.1. Resultados obtenidos en imágenes RGB partiendo del modelo de Cityscapes RGB

Las previsiones de los modelos entrenados tanto con el dataset de Cityscapes como Freiburg-forest, no eran demasiado buenos. Por este motivo se ha reentrenado la red, concretamente la parte del decoder, con las imágenes de las jornadas del dataset UMA-SAR¹ (Morales y col., 2021). El objetivo que se persigue con esto es que el encoder extraiga los rasgos más generales, por ejemplo, aquellos que permitan diferenciar un coche de una persona, y que en el decoder se determine si esta persona forma parte de personal de rescate, civil o víctima.

Se ha partido del modelo de Cityscapes ya que este ya diferencia entre vehículos y personas, tras el entrenamiento se busca conseguir algo parecido a lo obtenido en el capítulo anterior. Para recordar, en este capítulo se seleccionó un modelo de AdapNet++ bimodal y se dividió una de sus clases, la clase vehículo, que contenía a coches, camiones y autobuses, en dos, una para los coches y otra para camiones y autobuses.

Este caso es parecido al descrito en el párrafo anterior pues se pretende dividir la clase persona en civil, víctima y personal de rescate, y la clase vehículo en vehículo civil y vehículo de rescate. La principal diferencia reside en que en el capítulo anterior se mantenía el objetivo, diferenciar elementos en una ciudad. Sin embargo, ahora tanto el objetivo como el entorno o contexto de las imágenes cambiar. Esto dificulta la labor de transferencia de conocimiento. Para realizar el re-entrenamiento se utilizaron los siguientes parámetros.

- Tasa de aprendizaje: 0,01.
- Número de iteraciones: 6000.
- Tamaño de lote: 4.
- Número de clases: 8 (incluyendo el conjunto vacío, asociado al fondo de la imagen).

¹<https://www.uma.es/robotics-and-mechatronics/cms/menu/robotica-y-mecatronica/datasets/>

El motivo de emplear un tamaño de lote (Batch size) tan pequeño es la aparición de un problema que no se observa con la red AdapNet++ bimodal. Durante el entrenamiento, en cada iteración se guardan los valores del gradiente de la función de pérdida en función de cada parámetro entrenable de la red, esto se hace para cada imagen del lote. Para lotes muy grandes, deben de guardarse una gran cantidad de valores, lo que puede provocar que la memoria llegue a su máximo y, dependiendo del software y código que se esté utilizando, se detenga el entrenamiento.

El problema descrito en esta red ocurre para un tamaño de lote manejable durante el entrenamiento de AdapNet++ bimodal pero que llena la memoria en caso de utilizar la red AdapNet++ unimodal. Hay dos formas de solucionar esto:

- Con la opción `gpu_options.allow_growth = True` de `tf.ConfigProto()`.
- Con un tamaño de lote menor.

La primera opción ya estaba implementada al ocurrir el fallo por lo que se combinó con la segunda. La aparición de este problema dependerá de la máquina que se esté utilizando para entrenar la red. Los resultados se presentan en la figura 6.8.

Puede observarse en las 3 imágenes que no logra identificarse casi nada, concretamente el MIoU sobre el propio conjunto de entrenamiento es de 2.14, por lo que muy pocos píxeles se clasifican correctamente. Esto puede deberse a que los entornos de partida son muy diferentes, una ciudad para Cityscapes y el campo para las imágenes del dataset UMA-SAR. Esto provoca que, aunque quieran identificarse las mismas clases, la red no sea capaz. El objetivo que se persigue con el entrenamiento de una red neuronal artificial en segmentación de imágenes es el de aprender una serie de patrones generales que permitan identificar los distintos elementos de una imagen de entrada. Sin embargo, cuando se realiza dicho entrenamiento, esta generalización no es tan amplia como para abarcar contextos distintos al presente en el entrenamiento realizado. Por este motivo, es necesario reentrenar la red en el nuevo contexto si se desean obtener unos resultados aceptables. Además de esto, y como se verá en el apartado 6.5, la forma en la que se calculan las pérdidas es el principal factor de este rendimiento tan bajo.

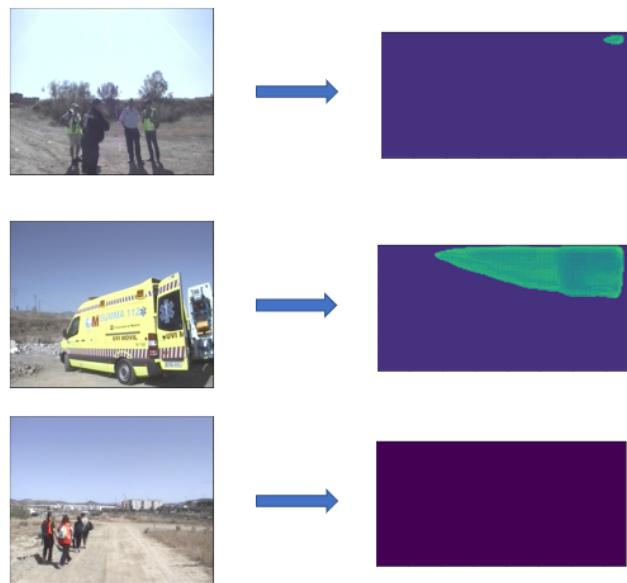


Figura 6.8: . Resultados obtenidos tras realizar un reentreno sobre el modelo de Cityscapes con las imágenes del dataset UMA-SAR. (Fuente: Elaboración propia)

6.4. Resultados sobre imágenes térmicas utilizando el modelo de cityscapes

Ante los malos resultados obtenidos al utilizar los modelos de AdapNet++ entrenados con mapas de profundidad (figuras 5.16 y 5.17), se ha probado a realizar predicciones sobre las imágenes térmicas con el modelo de AdapNet++ entrenado con las imágenes de Cityscapes RGB. Esta prueba se ha hecho porque la información proporcionada por mapas de profundidad e imágenes térmicas es muy diferente. Un valor muy alto en una imagen de profundidad implica cercanía de un objeto, en una imagen térmica implica una temperatura elevada independientemente de la distancia. Sin embargo, los contornos de las figuras en las imágenes térmicas son los mismos que en las imágenes RGB, además estos contornos están mejor definidos, la diferencia reside en que la imagen térmica es una escala de grises. Para comprobar que AdapNet++ puede identificar correctamente las categorías en escala de grises se han tomado varias imágenes del conjunto de validación del dataset de Cityscapes y se ha cambiado su formato a escala de grises. Hecho esto se ha cargado el modelo de AdapNet++ unimodal RGB de Cityscapes y se han pasado las imágenes transformadas obteniendo unas predicciones correctas. Cómo puede observarse en la figura 6.9 las predicciones son bastante buenas.

Tras comprobar que el modelo de Cityscapes para imágenes RGB puede también trabajar en escala de grises, se ha decidido realizar el proceso de transferencia de conocimiento con imágenes térmicas también con este modelo. No se presentan imágenes como ejemplo de los dos procesos de reentrenamiento ya que los resultados se asemejan a los del apartado anterior.

6.5. Resultados obtenidos utilizando el modelo general de AdapNet ++

Ante los resultados obtenidos a partir del modelo de Cityscapes se ha decidido emplear un checkpoint proporcionado por los autores de AdapNet++ el cual asigna al encoder de la red los pesos de ResNet de propósito general. Estos pesos de propósito general se suelen proporcionar con las redes

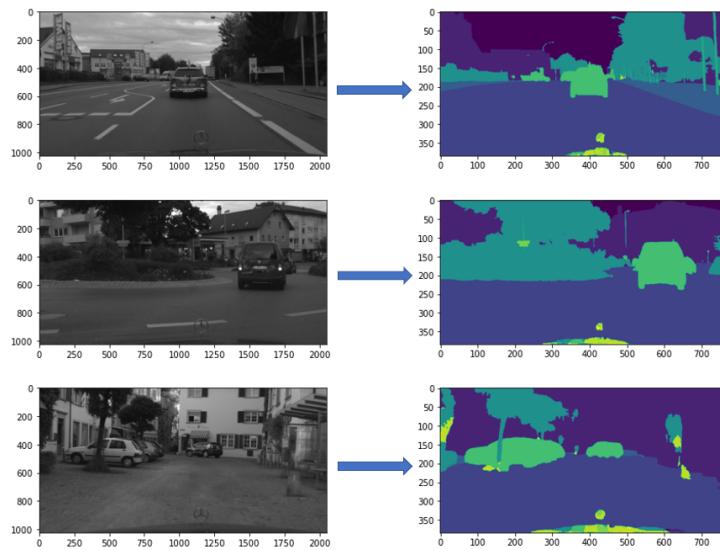


Figura 6.9: Predicciones de AdapNet++ para el modelo de Cityscapes RGB utilizando imágenes en escala de grises. (Fuente: Elaboración propia).

más empleadas, por ejemplo, ResNet o UNet entre otras, y pueden encontrarse utilizando recursos como ImageNet. El objetivo de proporcionar estos pesos es poder reentrenar las redes mencionadas desde cero para aplicaciones nuevas. En este caso, los autores de AdapNet++ cuentan con un archivo que contiene los pesos de ResNet adaptados al encoder de AdapNet++.

Con todo esto se pretende obtener unos resultados mejores que en el caso de emplear el modelo de Cityscapes original. Se ha entrenado solo la parte del decoder ya que no se dispone de una gran cantidad de ejemplos. Aunque los resultados sean algo mejores siguen siendo bastante pobres.

Para cargar este checkpoint de propósito más general hay que tener en cuenta que los nombres de las capas no son exactamente iguales a los de los modelos de Cityscapes o Freiburg forest, por lo que se elimina el comando `var.name.split('::')` (figura 5.5) a la hora de buscar dichos nombres.

Los resultados obtenidos en los tres últimos apartados pueden observarse en las tablas 1,2,3 y 4 al final del presente capítulo.

6.6. Consideración del fondo en el cálculo del error

El principal problema que se encontró en los casos explicados en los apartados anteriores consiste en que el error de entrenamiento era aceptable, concretamente se sitúa en torno a cero. Esto parecía indicar que las predicciones serían buenas, sin embargo, al evaluar el MIoU se obtenían unos resultados que no superaban el valor de tres. Este valor se obtenía sobre el propio conjunto de entrenamiento, si se hubiese evaluado sobre un conjunto de evaluación el resultado sería aún más bajo.

El motivo de que esto ocurra es el siguiente. Los autores proporcionan una serie de ficheros en Python, uno de ellos se utiliza para leer las imágenes en formato tfrecord. La peculiaridad de este fichero es que, si se asigna la clase cero al fondo de la imagen, como indican los autores de AdapNet++, este no se tiene en cuenta a la hora de calcular el error. La matriz correspondiente a la clase cero, con unos en los píxeles que pertenecen al fondo de la imagen y cero en el resto, se lee como si fuese una matriz nula.

Lo explicado en el párrafo anterior es útil cuando la mayor parte de la imagen está asignada a alguna clase, como es el caso de los datasets utilizados por los autores. Por el contrario, en la tarea objeto de este trabajo ocurre justo lo contrario, la mayoría de los píxeles pertenecen al fondo de la imagen en prácticamente todo el dataset. Esto provoca que se obtengan valores del error muy bajos, ya que gran parte de la imagen no se tiene en cuenta, por lo que, aunque la segmentación no sea buena, la actualización de los pesos de la red es muy pequeña debido a que el gradiente del error también lo es. De esta manera se obtienen una gran región mal clasificada que no se tiene en cuenta en el error y una pequeña región que en caso de estar mal clasificada no podrá influir lo suficiente en la actualización de los pesos de la red.

Esto afecta principalmente a las 2 últimas capas, las cuales se cargan con pesos aleatorios y no los del modelo de Cityscapes original. Como estos pesos no se pueden corregir lo suficiente y además son los encargados de reconstruir la imagen, ocurre algo parecido a lo observado en el apartado 6.1 cuando los checkpoints correspondientes a etapas más tempranas del entrenamiento definían peor los bordes que los asociados a etapas más avanzadas. En este caso, como el error es tan bajo, no son capaces de adaptarse

a ninguna forma.

Para solventar esto basta con no asociar ninguna clase al cero, en este caso, en lugar de crear ocho clases, de cero a siete ambos incluidos, se crean nueve, de cero a ocho, y ninguna de ellas se asigna a la clase cero. Con esto, el valor del error es mucho más alto y por lo tanto el valor de los pesos puede variar mucho más.

Para comprobar todo lo explicado se presentan en la figura 6.10 las gráficas de evolución del error para los siguientes casos.

- Reentreno con imágenes térmicas a partir de modelo de Cityscapes RGB.
- Reentreno con imágenes térmicas a partir del modelo de Cityscapes para mapas de profundidad.
- Reentreno para imágenes tanto RGB como térmicas partiendo del modelo de AdpaNet++ de propósito general.

Se presenta, con el fin de comparar el cambio en el valor del error, la gráfica para imágenes RGB cuando no se tiene en cuenta el fondo. Como puede observarse el valor no baja de 0,4 en ninguno de los casos y en el caso de la red de propósito general empieza siendo bastante alto. Este valor más alto del error permite que los pesos se actualicen mejor y mejora el desempeño de la red.

En todos los casos, a excepción de c), el error disminuye muy rápidamente al principio y posteriormente lo hace más lento. Esto puede deberse a que la red entrenada con mapas de profundidad obtiene información muy diferente a la proporcionada por imágenes térmicas, por lo que la red tarda más en adaptarse al nuevo tipo de información. Además, los pesos del encoder no se están modificando por lo que las características a partir de las cuales el decoder reconstruye la imagen siguen siendo propias de un mapa de profundidad.

El hecho de que las gráficas sigan una tendencia oscilante se debe a que, en el caso de AdapNet++ unimodal, el tamaño de lote (batch size) se ha tenido que disminuir de 16 a 4 por limitaciones de hardware, concretamente

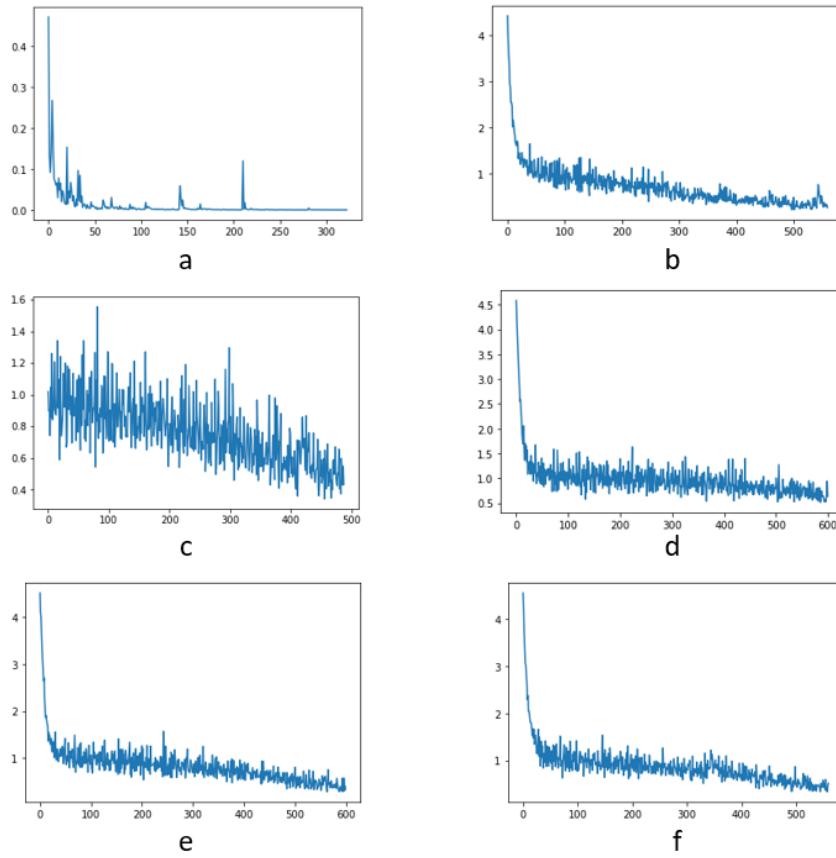


Figura 6.10: Evolución del error. a) Reentrenamiento a partir del modelo de Cityscapes rgb sin tener en cuenta el fondo de la imagen. b) Reentrenamiento a partir del modelo de Cityscapes RGB c) Reentrenamiento a partir del modelo de Cityscapes de mapas de profundidad con imágenes térmicas. d) Reentrenamiento a partir del modelo de propósito general con imágenes térmicas. e) Reentrenamiento a partir del modelo de Cityscapes RGB con imágenes térmicas. f) Reentreno a partir del modelo de propósito general con imágenes RGB.

de memoria. Normalmente cuanto mayor es el tamaño de lote, más se atenúa este efecto.

6.7. Índices obtenidos con las diferentes estrategias de entrenamiento

Ahora se presentarán con más detalle los resultados que se han obtenido al emplear las distintas estrategias, no solo se expondrán las gráficas, sino que además se proporcionarán tablas que contienen el número de falsos positivos, falsos negativos y verdaderos positivos para cada clase en todas las estrategias empleadas. También, con el fin de medir con qué precisión se segmenta cada clase, se presentará el índice IoU para cada clase, resultado de dividir los verdaderos positivos entre la suma de los valores mencionados anteriormente (ecuación 6.1).

$$IoU = \frac{\text{Verdaderos_positivos}}{\text{Verdaderos_positivos} + \text{Falsos_positivos} + \text{Falsos_negativos}} \quad (6.1)$$

Los índices obtenidos con cada estrategia de reentrenamiento se dividirán en dos partes, primero los resultados generales (tablas 6.1, 6.2, 6.3, 6.4), posteriormente divididos por clase y con el valor del IoU (tablas 6.5, 6.6, 6.7, 6.8, 6.9).

Por último y con el fin de poder visualizar con más facilidad la precisión con la que se detecta cada clase, se presentan unos gráficos de barras con el valor del IoU obtenido para cada clase en los casos expuestos en las tablas de la 6.5 a la 6.9. En el eje horizontal se encuentran las distintas clases y en el vertical el valor del índice IoU (figuras 6.11 a 6.14).

Modelo	Pérdida	MIoU
Cityscapes RGB	0,0002	2,14
AdapNet general	0,06	1,95

Tabla 6.1: Resultados obtenidos para imágenes RGB sin tener en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)

Modelo	Pérdida	MIoU
Cityscapes RGB	0,0002	1,9
Cityscapes profundidad	0,0034	2,45
AdapNet general	0,0001	1,86

Tabla 6.2: Resultados obtenidos para imágenes térmicas sin tener en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)

Modelo	Pérdida	MIoU
Cityscapes RGB	0,26	39
AdapNet general	0,51	41

Tabla 6.3: Resultados obtenidos con imágenes RGB teniendo en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)

Modelo	Pérdida	MIoU
Cityscapes RGB	0,4	33
Cityscapes profundidad	0,43	15
AdapNet general	0,64	22

Tabla 6.4: Resultados obtenidos con imágenes térmicas teniendo en cuenta el fondo en el cálculo del error. (Fuente: Elaboración propia)

Clase	Positivos reales	Falsos positivos	Falsos negativos	IoU
Escombros	10582,52	629,57	1936,28	0,80
Civil	2,71	0,36	2080,18	0,00
Personal de rescate	2625,34	1465,51	3939,58	0,33
Vehículo civil	3644,06	3,59	4108,61	0,47
Vehículo de rescate	6291,46	1706,57	3010,44	0,57
Víctima	0,00	0,00	342,00	0,00
Zapatos	0,00	0,00	351,12	0,00
Fondo	255798,22	12162,10	199,48	0,95

Tabla 6.5: Resultados desglosados por clase obtenidos con imágenes RGB utilizando el modelo de Cityscapes teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)

Clase	Positivos reales	Falsos positivos	Falsos negativos	IoU
Escombros	9488,40	1537,09	3030,40	0,68
Civil	0,00	0,00	2082,90	0,00
Personal de rescate	2509,78	1729,55	4055,14	0,30
Vehículo civil	5712,36,06	116,84	2040,30	0,73
Vehículo de rescate	6556,46	519,71	2745,43	0,67
Víctima	0,00	0,00	342,00	0,00
Zapatos	0,00	0,00	351,12	0,00
Fondo	254783,22	11958,58	1214,48	0,95

Tabla 6.6: . Resultados desglosados por clase obtenidos con imágenes rgb utilizando el modelo de AdapNet++ general teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)

Clase	Positivos reales	Falsos positivos	Falsos negativos	IoU
Escombros	9583,48	3736,27	498,49	0,69
Civil	112,07	7,32	1632,41	0,06
Personal de rescate	4801,17	3794,57	1207,39	0,49
Vehículo civil	0,00	0,00	7172,42	0,00
Vehículo de rescate	7965,68	7879,16	789,28	0,48
Víctima	0,00	0,00	361,00	0,00
Zapatos	0,00	0,00	343,48	0,00
Fondo	254575,80	2456,49	5869,35	0,97

Tabla 6.7: Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de Cityscapes RGB teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)

Clase	Positivos reales	Falsos positivos	Falsos negativos	IoU
Escombros	8051,86	17086,54	2030,11	0,30
Civil	0,00	0,00	1744,48	0,00
Personal de rescate	0,00	0,00	6008,56	0,00
Vehículo civil	0,00	0,00	7172,42	0,00
Vehículo de rescate	0,00	0,00	8754,96	0,00
Víctima	0,00	0,00	361,00	0,00
Zapatos	0,00	0,00	343,48	0,00
Fondo	259707,33	10066,27	737,82	0,96

Tabla 6.8: Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de Cityscapes para mapas de profundidad teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)

Clase	Positivos reales	Falsos positivos	Falsos negativos	IoU
Escombros	4169,15	1890,33	5912,82	0,35
Civil	0,00	0,00	1744,48	0,00
Personal de rescate	1143,75	1104,64	4864,81	0,16
Vehículo civil	0,00	0,00	7172,42	0,00
Vehículo de rescate	3697,33	2018,25	5057,63	0,34
Víctima	0,00	0,00	361,00	0,00
Zapatos	0,00	0,00	343,48	0,00
Fondo	259164,73	21723,81	1280,42	0,92

Tabla 6.9: Resultados desglosados por clase obtenidos con imágenes térmicas utilizando el modelo de AdapNet++ general teniendo en cuenta el fondo de la imagen en el cálculo del error. (Fuente: Elaboración propia)

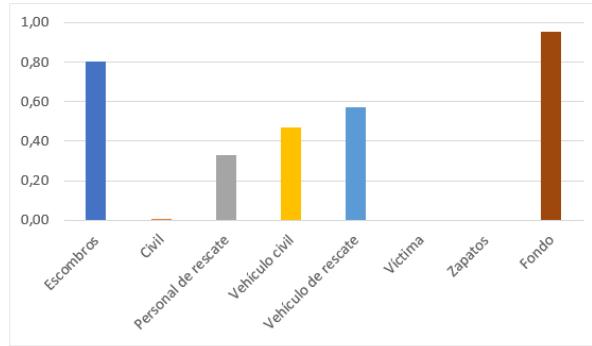
Capítulo 6. Resultados

Figura 6.11: Valores del IoU de cada clase al entrenar con imágenes RGB sobre el modelo de Cityscapes RGB teniendo en cuenta el fondo en el cálculo del error.

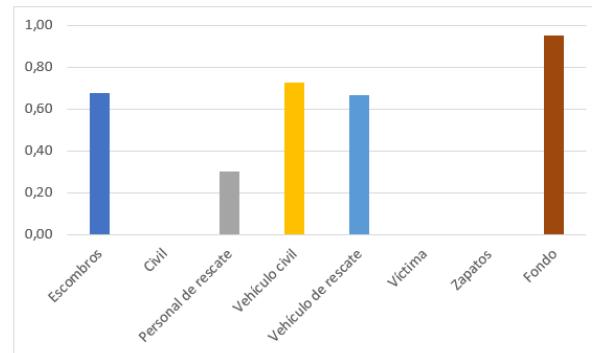


Figura 6.12: Valores del IoU de cada clase al entrenar con imágenes RGB sobre modelo de AdapNet++ general teniendo en cuenta el fondo en el cálculo del error

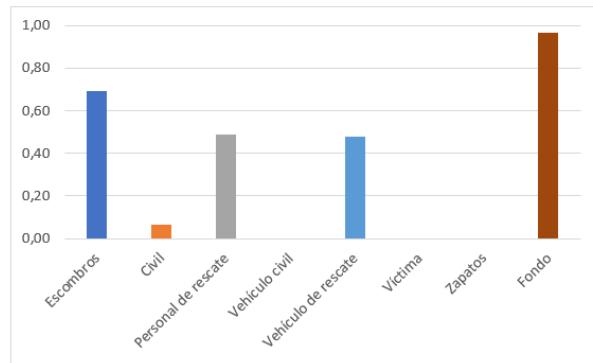


Figura 6.13: Valores del IoU de cada clase al entrenar con imágenes térmicas sobre modelo de Cityscapes RGB teniendo en cuenta el fondo en el cálculo del error.

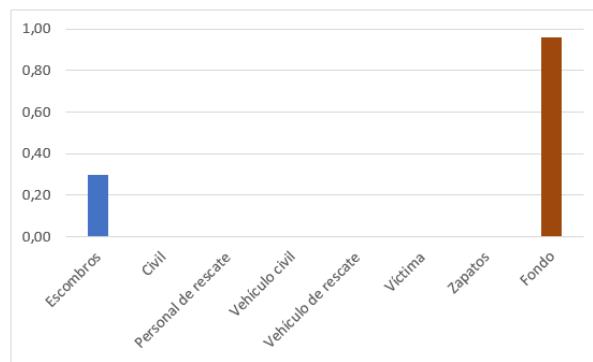


Figura 6.14: Valores del IoU de cada clase al entrenar con imágenes térmicas sobre el modelo de Cityscapes para mapas de profundidad teniendo en cuenta el fondo en el cálculo del error.

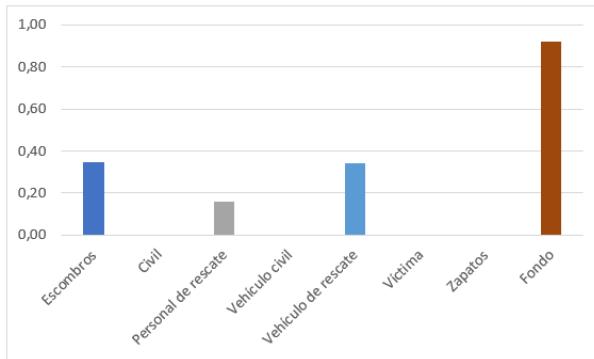


Figura 6.15: Valores del IoU de cada clase al entrenar con imágenes térmicas sobre el modelo de AdapNet++ general teniendo en cuenta el fondo n el cálculo del error.

Puede observarse que, en todos los casos, las clases de fondo y escombros son las que mejor se clasifican, esto se debe a que, por lo general, son las que más píxeles ocupan en las imágenes además de aparecer en muchas de ellas. Lo mismo ocurre con las clases de vehículo de rescate y personal de rescate, que aparecen en la mayoría de las imágenes, aunque sin ocupar tantos píxeles.

La estrategia que peores resultados presenta es aquella en la que se entrena la red utilizando como entrada imágenes térmicas y se parte del modelo de Cityscapes para mapas de profundidad. Esto se debe a los específicas que son este tipo de redes, en las que simplemente cambiando el contexto de las imágenes los resultados pueden pasar de ser muy buenos a ser malos. En este caso no solo cambia el contexto, sino que además lo hace el tipo de información que se le proporciona a la red como entrada.

Una de las cuestiones más destacables, y que ocurre en los 5 casos, es que la precisión a la hora de detectar a las víctimas es cero. Esto es bastante grave pues el objetivo es utilizar la red en labores de rescate. En estas tareas es de vital importancia identificar correctamente a las víctimas ya que un falso negativo implica no identificar a una persona que necesita ayuda y un falso positivo implica enviar recursos a donde no es necesario. En los 5 casos no hay verdaderos positivos, como puede observarse en las tablas 6.5-6.9. El motivo de este problema es que no se cuenta con demasiadas imágenes en las que aparezcan víctimas en el dataset utilizado para el entrenamiento.

La estrategia que ha mostrado mejores resultados es partir del modelo de AdapNet++ general y entrenar con imágenes RGB. Por otro lado, cabe destacar que los resultados para imágenes térmicas partiendo del modelo de Cityscapes RGB no difieren demasiado de los obtenidos con imágenes RGB y el mismo modelo. Además, en el caso de las imágenes térmicas, el modelo que da mejores resultados es Cityscapes RGB y no AdapNet++ de propósito general.

Además de todo lo anterior, puede observarse, en las imágenes en RGB, que los modelos entrenados a partir de Cityscapes detectan mejor a las personas, tanto civiles como personal de rescate. Por otra parte, los que se entrena a partir de los pesos proporcionados por los autores de AdapNet++ detectan con mayor precisión los vehículos. En cuanto a los modelos correspondientes a imágenes térmicas, el modelo que parte del modelo de Cityscapes para imágenes RGB es el que mejor detecta todas las clases.

Por último, se presentarán 3 imágenes como ejemplo para mostrar las predicciones efectuadas por los modelos cuando se tiene en cuenta el fondo en el cálculo del error (figura 6.16). En estas imágenes puede observarse lo que se ha explicado, con la excepción de que la ambulancia de la segunda imagen parece mejor segmentada en el modelo de Cityscapes. No debe olvidarse que las tablas 6.5-6.9 y sus correspondientes gráficas hacen referencia a la media obtenida para todas las imágenes empleadas en el entrenamiento y las imágenes de la figura 6.16 representan casos puntuales.

Con toda la información obtenida a partir del reentreno de AdapNet++ unimodal, se ha decidido que para reentrenar la red bimodal se partirá de dos encoders del modelo de Cityscapes RGB con 12 clases. El motivo de esto es que en el caso de las imágenes térmicas ha sido el que ha mostrado mejores resultados, en el caso de las imágenes RGB está casi a la par con el modelo de AdapNet++ general. La parte del decoder y los bloques SSMA se cargarán del modelo de Cityscapes para doce clases bimodal, es decir, entrenado para imágenes RGB y mapas de profundidad. En la figura 6.17 se presenta el procedimiento seguido para cargar los diferentes bloques de la red.

En este código, la variable “names” es una lista que contiene los nombres de las capas del checkpoint asociado a la dirección “path1”, en este caso corresponde al modelo de Cityscapes con 12 clases unimodal para imágenes

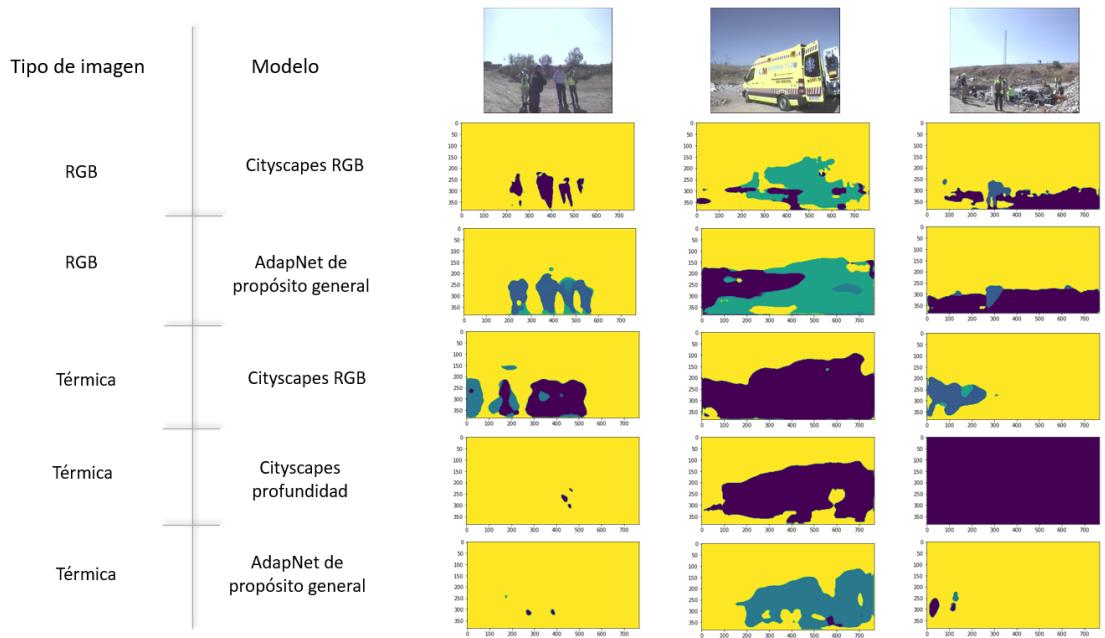


Figura 6.16: Resultados obtenidos con los diferentes modelos si se tiene en cuenta el fondo al calcular el error.

```

for item in inspect_list1:
    names.append(item[0])
for item in inspect_list2:
    names2.append(item[0])

#CARGAR ENCODERS
for var in import_variables:
    if 'depth' in var.name:
        temp1=var.name.replace('depth'+'/','')
        temp=temp1.split(':')
        if temp[0] in names:
            initialize_variables[temp[0]]=var

saver=tf.train.Saver(initialize_variables)
saver.restore(sess,path1)
initialize_variables={}

for var in import_variables:
    if 'rgb' in var.name:
        temp1=var.name.replace('rgb'+'/','')
        temp=temp1.split(':')
        if temp[0] in names:
            initialize_variables[temp[0]]=var

saver=tf.train.Saver(initialize_variables)
saver.restore(sess,path1)
#CARGAR DECODER Y SSMA
initialize_variables={}
SSMA=['conv511','conv512','conv513','conv514','conv515','conv518','conv519','conv552','conv553']
DECODER=['conv41','conv16','conv89','conv96','conv88','conv95']

for var in import_variables:
    temp1=var.name.split('/')[0]
    if temp1 in SSMA or temp1 in DECODER:
        temp=var.name.split(':')
        if temp[0] in names2:
            initialize_variables[temp[0]]=var
saver=tf.train.Saver(initialize_variables)
saver.restore(sess,path2)

```

Figura 6.17: Carga de los bloques de una red AdapNet++ bimodal. (Fuente: Elaboración propia).

RGB. Por otro lado, “names2” corresponde a los nombres de las capas del checkponit correspondiente al modelo de Cityscapes bimodal con 12 clases.

Primero se cargan los pesos del encoder correspondiente, en AdapNet++ bimodal original, a mapas de profundidad, pero en este caso se introducen los pesos de un modelo entrenado con imágenes RGB. Posteriormente se hace lo mismo con la parte del encoder correspondiente a imágenes RGB. Por último, se crea una lista con las capas correspondientes a los bloques SSMA y al decoder, para este último se excluyen aquellas capas cuya morfología depende del número de clases, y se cargan los pesos del modelo de Cityscapes bimodal de 12 clases.

Tras cargar este modelo y entrenar la parte del decoder, se obtienen unos resultados que superan a los de AdapNet++ unimodal (tabla 6.10), concretamente:

- Las pérdidas se sitúan en 0,09.
- El valor del MIoU sube hasta 52. Con esto se aumenta en casi 11 puntos el mejor valor obtenido para AdapNet++ unimodal.

Clase	Positivos reales	Falsos positivos	Falsos negativos
Escombros	8299,225	939,36	1982,87
Civil	1218,45	1246,69	610,58
Personal de rescate	3391,68	910,17	2211,81
Vehículo civil	7009,61	941,06	695,04
Vehículo de rescate	7754,52	1931,37	972,29
Víctima	0	0	361
Zapatos	0	0	343,48
Fondo	256785,43	4484,43	3275,98

Tabla 6.10: Resultados obtenidos para AdapNet++ bimodal.

Capítulo 7

Conclusiones y trabajos futuros

En este trabajo se ha utilizado la red neuronal llamada AdapNet++ con su bloque SSMA para comprobar si puede ser útil en labores de rescate en las cuales se utilizan tanto imágenes RGB como imágenes térmicas.

Para ello primero se han hecho una serie de pruebas con un dataset que ya habían utilizado los autores de la red con el objetivo de reproducir los resultados obtenidos por dichos autores, pero incluyendo una clase más. En estas pruebas se ha ampliado el número de clases para los cuales se había entrenado ya la red. Durante esta fase se ha determinado que la tasa de aprendizaje que ofrecía mejores resultados es de 0,01. Con este valor se acelera la convergencia y se reduce el error. Para un valor de 0,001 como se indica en (Valada y col., 2019) y con la cantidad de ejemplos disponibles para el entrenamiento el error se estanca en un mínimo local.

Otra de las cuestiones observadas en este proceso de transferencia de conocimiento es que a partir de las 400 iteraciones aproximadamente el MIoU se estanca a pesar de que el error continúa disminuyendo. Esto puede indicar un posible caso de sobreajuste a partir de la iteración 400. Por este motivo, el proceso de ajuste fino seguido en una de las estrategias de reentrenamiento se hizo a partir de esta iteración.

Entre las estrategias seguidas para el proceso de reentrenamiento el mejor desempeño se da cuando se reentrenan las dos últimas capas y posteriormente se lleva a cabo un ajuste fino. Para llevar a cabo este proceso,

se parte del checkpoint correspondiente a la iteración 409 del entrenamiento de las dos últimas capas, pues este corresponde al instante a partir del cual comienza el problema del sobreajuste. El objetivo de esto es reentrenar ligeramente al resto de capas de la red y mejorar la definición de los contornos que otorga el modelo correspondiente al checkpoint 409. Por otra parte, el peor resultado se obtiene al reentrenar el decoder por completo.

En cuanto al reentreno con imágenes del dataset UMA-SAR, se han seguido dos estrategias para el reentrenamiento. En la primera de ellas no se ha tenido en cuenta en fondo de la imagen. Este caso proporciona resultados bastante pobres ya que las imágenes del dataset cuentan con una gran cantidad de píxeles que no pertenecen a las clases que se pretenden detectar por lo que asocian a la clase “fondo”.

Los autores de AdapNet++ no tienen en cuenta esta clase a la hora de calcular el error por lo que, los píxeles implicados en obtener los resultados de la función de pérdida, como no son demasiados, dan valores muy bajos lo que termina provocando que los parámetros de la red no se actualicen correctamente.

Cuando no se asocia la clase cero al fondo de la imagen, el valor del error aumenta, pero también lo hace la magnitud en la que se actualizan los pesos por lo que el valor del MIoU, que es el que realmente mide los píxeles clasificados de manera correcta, es mejor que en el caso descrito en el párrafo anterior.

Debido a las pocas imágenes con víctimas de las que se dispone, la red no es capaz de identificarlas correctamente. Esto es bastante grave teniendo en cuenta el propósito para el cual se desea emplear dicha red. Como trabajo futuro puede considerarse añadir más imágenes con víctimas al dataset para que la red pueda aprender los patrones generales de esta clase.

Por último, los resultados mejoran cuando se utiliza una red AdapNet++ bimodal entrenada a partir de dos encoders correspondientes al modelo de Cityscapes para imágenes RGB. El valor de MIoU sube hasta 52 aunque sigue sin poder identificar a las víctimas, esto debido a que no se cuenta con suficientes imágenes en las que aparezca esta clase.

En cuanto a trabajos futuros, se deben de etiquetar más imágenes para generar un dataset mayor. De esta manera el desempeño de la red mejorará.

Dentro de estos datasets debería de considerarse el incluir más imágenes con víctimas.

En el caso de las imágenes térmicas, lo más recomendable es seguir partiendo del modelo de Cityscapes RGB hasta que el dataset sea lo suficientemente amplio ya que es el que proporciona mejores resultados. En el caso de imágenes RGB, lo más recomendable es utilizar el modelo de AdapNet++ proporcionado por sus autores.

También se puede considerar la posibilidad de utilizar una red con arquitectura Mask-RCNN. Este tipo de red primero detecta las clases de la imagen como haría una red tipo YOLO y posteriormente se segmenta únicamente lo que hay dentro de las “cajas” colocadas por la red. En las imágenes del dataset, en las que los píxeles correspondientes a las clases que se pretende clasificar ocupan, en muchos casos, una pequeña parte de la imagen, el utilizar este tipo de redes puede evitar tener que entrenar la red para detectar la clase “fondo” que no es interesante para la aplicación en la que se pretende utilizar la red. Este tipo de red, además, ofrece la ventaja de poder determinar el número de instancias de una determinada clase, lo cual puede ser útil en caso de haber más de una víctima. El principal problema de este enfoque es que, en principio, se pierde el carácter bimodal que proporciona AdapNet++ y que es el objetivo de este trabajo, por lo que si se desea preservar esta característica, se debe de comprobar si existe alguna forma de utilizar el bloque SSMA (Valada y col., 2019) para fusionar características de dos redes tipo Mask-RCNN.

Por último, puede considerarse cambiar la función de perdida de AdapNet++. La entropía cruzada (cross-entropy) provoca que el entrenamiento acabe dominado por las clases que más píxeles ocupan. Esto puede comprobarse en el hecho de que las categorías fondo y escombros se segmentan mucho mejor que el resto. Una función de pérdida que puede atenuar este efecto es la pérdida focal, también se puede utilizar el índice IoU para calcular la función de pérdida de la red.

Bibliografía

- Badrinarayanan, V., Kendall, A. & Cipolla, R. (2016). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation.
- Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K. & Yuille, A. L. (2017). DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs.
- Ciresan, D. C., Meier, U. & Schmidhuber, J. (2012). Multi-column Deep Neural Networks for Image Classification. *CoRR, abs/1202.2745*. <http://arxiv.org/abs/1202.2745>
- Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S. & Schiele, B. (2016). The Cityscapes Dataset for Semantic Urban Scene Understanding.
- Duchi, J., Hazan, E. & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research, 12(7)*.
- Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning* [<http://www.deeplearningbook.org>]. MIT Press.
- He, K., Gkioxari, G., Dollár, P. & Girshick, R. (2018). Mask R-CNN.
- Kingma, D. P. & Ba, J. (2017). Adam: A Method for Stochastic Optimization.
- Ku, J., Harakeh, A. & Waslander, S. L. (2018). In Defense of Classical Image Processing: Fast Depth Completion on the CPU. *2018 15th Conference on Computer and Robot Vision (CRV)*, 16-22.
- Lin, M., Chen, Q. & Yan, S. (2014). Network In Network.
- Liu, W., Rabinovich, A. & Berg, A. C. (2015). ParseNet: Looking Wider to See Better.
- Long, J., Shelhamer, E. & Darrell, T. (2014). Fully Convolutional Networks for Semantic Segmentation. *CoRR, abs/1411.4038*. <http://arxiv.org/abs/1411.4038>

- Morales, J., Vázquez-Martín, R., Mandow, A., Morilla-Cabello, D. & García-Cerezo, A. (2021). The UMA-SAR Dataset: Multimodal Data Collection from a Ground Vehicle During Outdoor Disaster Response Training Exercises. *The International Journal of Robotics Research*, 40(6-7), 835-847. <https://doi.org/10.1177/02783649211004959>
- Ng, A. (s.f.). DeepLearningAI. <https://www.youtube.com/channel/UCcIXc5%20mJsHVYTZR1maL5l9w>
- Noh, H., Hong, S. & Han, B. (2015). Learning Deconvolution Network for Semantic Segmentation.
- Planche, B. & Andres, E. (2019). *Hands-On Computer Vision with TensorFlow 2*. Packt Publishing Ltd.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4, 1-17.
- Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection.
- Ren, S., He, K., Girshick, R. & Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.
- Ronneberger, O., Fischer, P. & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation.
- Sermanet, P., Kavukcuoglu, K., Chintala, S. & LeCun, Y. (2012). Pedestrian Detection with Unsupervised Multi-Stage Feature Learning. *CoRR*, abs/1212.0142. <http://arxiv.org/abs/1212.0142>
- Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition.
- Sutskever, I., Martens, J., Dahl, G. & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. En S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th International Conference on Machine Learning* (pp. 1139-1147). PMLR. <https://proceedings.mlr.press/v28/sutskever13.html>
- University, S. (s.f.). CS231n: Convolutional Neural Networks for Visual Recognition. <https://cs231n.github.io/>
- Valada, A., Mohan, R. & Burgard, W. (2019). Self-Supervised Model Adaptation for Multimodal Semantic Segmentation [Special Issue: Deep Learning for Robotic Vision]. *International Journal of Computer Vision (IJCV)*. <https://doi.org/10.1007/s11263-019-01188-y>
- Valada, A., Oliveira, G., Brox, T. & Burgard, W. (2016). Deep Multispectral Semantic Scene Understanding of Forested Environments using Mul-

- timodal Fusion. *International Symposium on Experimental Robotics (ISER)*.
- Valada, A., Vertens, J., Dhall, A. & Burgard, W. (2017). AdapNet: Adaptive Semantic Segmentation in Adverse Environmental Conditions. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 4644-4651.
- Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. *CoRR, abs/1212.5701*. <http://arxiv.org/abs/1212.5701>
- Zhao, H., Shi, J., Qi, X., Wang, X. & Jia, J. (2017). Pyramid Scene Parsing Network.