



UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

Métodos para la detección de cambios en imágenes

Alumno: Irene María Padilla Muñoz

Tutor: Antonio J. Rivera Rivas
María J. del Jesus Díaz

Dpto: Informática



UNIVERSIDAD DE JAÉN

D./D^a Antonio J. Rivera Rivas y D./D^a María J. del Jesus Díaz, tutor(es) del Trabajo Fin de Grado titulado: **Métodos para la detección de cambios en imágenes**, que presenta Irene María Padilla Muñoz, autoriza(n) su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2021

El estudiante

PADILLA
MUÑOZ
IRENE
MARIA -
77354547
G

Firmado digitalmente por PADILLA MUÑOZ IRENE MARIA - 77354547G
Fecha: 2021.09.09 18:12:04 +02'00'

Los tutores

RIVERA
RIVAS
ANTONIO
JESUS -
26015863B

Firmado digitalmente por RIVERA RIVAS ANTONIO JESUS - 26015863B
Fecha: 2021.09.09 18:25:47 +02'00'

DEL JESUS
DIAZ
MARIA
JOSE -
26485651R

Firmado digitalmente por DEL JESUS DIAZ MARIA JOSE - 26485651R
Fecha: 2021.09.09 18:42:55 +02'00'

Irene María Padilla Muñoz

Antonio J. Rivera Rivas María J. del Jesus Díaz

Agradecimientos

En un primer momento, agradecer a Antonio Jesús Rivera Rivas, María José del Jesús Díaz, a Fernando Puentes Marchal y al equipo de investigación del grupo SIMIDAT por su ayuda y colaboración a la hora de desarrollar este proyecto. Por aportar todos sus conocimientos y recursos y que me han ayudado a adquirir numerosos conocimientos.

A mis padres y hermana por guiarme y apoyarme en esta etapa de mi vida. Por estar en los buenos y malos momentos y haberme acompañado hasta llegar a este momento creyendo en mí. Y por haberme dado todo aquello que necesitaba, amor y paciencia.

A mis peludos y no tan peludos por su compañía.

Y por último a Cactus, a Paul A., Irene M. y Javier M. por ser los mejores amigos que se puedan tener, por enseñarme que es la amistad y a como disfrutar la vida. Y en especial a Sergio B., por su paciencia, por escuchar todos mis dramas y por estar siempre cuando lo necesitaba.

Tabla de contenidos

1. INTRODUCCIÓN	1
1.1. Motivación	2
1.2. Objetivos	6
1.3. Estructura del documento	6
1.4. Planificación	7
2. ESTADO DEL ARTE	11
2.1. Ciencia de datos	11
2.1.1. Extracción de conocimiento de bases de datos	11
2.1.2. Tareas	12
2.1.3. Técnicas	13
2.2. Visión por computador	14
2.3. Detección de cambios en imágenes	15
2.4. Métodos para la Detección de cambios	16
2.5. Problemática asociada a la detección de cambios en imágenes	18
3. HERRAMIENTAS UTILIZADAS	21
3.1. Especificaciones	21
3.2. Python	22

3.3. Librerías	22
3.3.1. OpenCV	23
3.3.2. Numpy	23
3.3.3. Scikit-learn	23
3.3.4. Keras	24
3.3.5. TensorFlow	24
4. MÉTRICAS	27
4.1. Descripción de las métricas	28
4.1.1. Precisión y exhaustividad	28
4.1.2. F-Measure	29
4.1.3. Curva ROC y área bajo la curva (AUC)	30
4.1.4. Kappa	32
4.1.5. Overall Accuracy (OA)	34
4.1.6. User Accuracy (UA)	34
4.2. Interpretación de las métricas	34
4.3. Implementación de las métricas	35
5. DATASETS	37
5.1. Datasets con máscaras de cambio	37
5.1.1. SZTAKI	38
5.1.2. AICD	39
5.1.3. OSCD	40
5.1.4. HRSCD	41
5.1.5. xBD	43

5.1.6. LEVIR-CD	44
5.1.7. Hi-UCD	45
5.1.8. SpaceNet7	46
5.2. Problemas con las máscaras	48
5.3. Posibles soluciones	48
5.3.1. Máscara de segmentación	48
5.3.2. Dividir las imágenes	49
5.3.3. Data augmentation	49
5.3.4. Crear un dataset sintético	50
5.4. Pre-procesamiento	52
5.4.1. Registro de imágenes	53
5.4.2. Corrección de la iluminación	53
5.4.3. Corrección atmosférica	54
5.4.4. Aplicación sobre el dataset sintético	55
6. MÉTODOS TRADICIONALES DESARROLLADOS	57
6.1. Image Differencing	57
6.1.1. Funcionamiento	58
6.1.2. Pseudocódigo	59
6.1.3. Implementación	60
6.2. Canny	66
6.2.1. Funcionamiento	67
6.2.2. Pseudocódigo	70
6.2.3. Implementación	71

6.3. Difference PCA	71
6.3.1. Funcionamiento	72
6.3.2. Pseudocódigo	73
6.4. PCA K-Means	74
6.4.1. Funcionamiento	74
6.4.2. Pseudocódigo	76
6.4.3. Implementación	77
7. MÉTODOS DE APRENDIZAJE PROFUNDO DESARROLLADOS	79
7.1. Fundamentos de las Redes Neuronales	79
7.1.1. Elementos de una red neuronal	80
7.1.2. Tipos de capas	87
7.1.3. Aspectos a considerar	90
7.2. Red Siamesa y Encoder-Decoder	93
7.3. Fully Convolutional siamese networks for change detection	95
7.4. Deep Siamese Multi-scale Fully Convolutional Network	97
7.5. Improved UNet++	100
8. EXPERIMENTACIÓN Y RESULTADOS	103
8.1. Experimentaciones	103
8.1.1. Experimentaciones con Image Differencing	105
8.1.2. Experimentaciones con Canny	108
8.1.3. Experimentaciones con PCA K-means	112
8.1.4. Experimentaciones con Fully Convolutional siamese networks for change detection	115

8.1.5. Experimentaciones con DSMS-FCN	118
8.1.6. Experimentaciones con Improved UNet++	119
8.2. Comparación de los diferentes métodos	121
8.2.1. Análisis estadístico de los resultados	122
9. CONCLUSIONES	125
Bibliografía	v

Lista de figuras

1.1. Esquema donde se puede observar los diferentes sub-campos de la Inteligencia Artificial. Imagen recreada a partir de una Figura de Rosebrock [2017]	3
1.2. Gráficas extraídas de Research [2020] , donde se muestra la evolución financiera de la Visión por computador sobre diferentes aplicaciones. .	5
1.3. Planificación seguida para el desarrollo de este proyecto.	9
2.1. Ejemplo de lo que se pretende obtener tras aplicar los algoritmos de detección de cambios.	16
3.1. Tabla extraída de Tiobe [2021] donde se puede ver el auge de este lenguaje con respecto al último año, situándose en segunda posición. .	22
4.1. Construcción matriz de confusión.	28
4.2. Ejemplo de F-Measure.	30
4.3. Curva ROC. Figura extraída de Wikipedia wik [2021]	31
4.4. Gráfica extraída de una ejecución de MATLAB , se puede observar la curva ROC y su correspondiente AUC.	32
4.5. En esta tabla se muestra un ejemplo de matriz de confusión.	33
5.1. Tabla extraída de Long [2021] donde se puede ver algunos de los diferentes datasets existentes para el campo de la Detección de Cambios .	38
5.2. Ejemplo de los diferentes pares con sus máscaras correspondientes que se puede encontrar en este dataset. Extraído de Benedek [2013] .	39

5.3. Ejemplo de uno de los pares que se pueden encontrar, extraído de Bourdis [2020]	40
5.4. Uno de los pares incluidos en este dataset. Extraído de Daudt [2018]	41
5.5. Problemática encontrada en las máscaras.	41
5.6. Ejemplo de los pares en alta resolución. Daudt [2021]	42
5.7. Ejemplo máscara de segmentación.	43
5.8. Imagen extraída de xVi [2021] donde se pueden apreciar las localizaciones y el tipo de desastre recogido por xBD.	43
5.9. Ejemplo de un par extraído del conjunto de xBD. Imágenes tomadas de xVi [2021]	44
5.10. Extraída de Chen and Shi [2020], se pueden encontrar ejemplos de los pares con sus correspondientes máscaras.	45
5.11. Extraída de Tian [2020], muestra las máscaras de cambio y de segmentación junto con el código de color para interpretarlas.	46
5.12. Ejemplo de un par de imágenes de meses diferentes, en la parte superior se encuentran las máscaras con las edificaciones correspondientes a las imágenes inferiores. A la derecha se puede ver la máscara generada al restar las anteriores. Imágenes extraídas de Spa [2021]	47
5.13. Ejemplo de recortado de imágenes.	49
5.14. Resultado de aplicar una utilidad de Keras para <i>Data Augmentation</i> . Figura extraída de Brownlee [2019].	50
5.15. En la parte superior se puede ver el par de imágenes original y el resultado obtenido tras aplicar PCA K-means. En la parte inferior se ve el resultado de combinar la primera y la segunda imagen de tal manera que los cambios se correspondan a la máscara generada.	52
5.16. Ejemplo del nuevo par generado junto con la mascara de cambio correspondiente.	52
5.17. Ejemplo de mal funcionamiento de <i>image registration</i>	55

5.18. Un ejemplo de corrección gamma aplicado al dataset de Crop-Field con un valor de gamma de 0.5.	56
6.1. Resultado de la imagen de diferencia calculada por ambos algoritmos. .	61
6.2. Ejemplo de filtro Gaussiano extraído de Raziel [2017]	64
6.3. Ejemplo de umbralización adaptativa extraída de la documentación de OpenCV OpenCV [2021]	65
6.4. Se puede observar las claras diferencias entre ambos métodos de umbralización.	65
6.5. Dependiendo del tamaño del bloque, se obtienen diferentes resultados.	66
6.6. Resultado tras aplicar Canny a un par de imágenes de OSCD.	68
6.7. En la primera imagen se puede observar el resultado de rellenar los contornos si no se cuenta con el emborronamiento; mientras que en la parte inferior sí se ha producido dicho emborronamiento.	69
6.8. Filtro aplicado por blur.	70
7.1. A la izquierda se puede encontrar un ejemplo de los diferentes tipos de problemas lineales (AND) a la izquierda y no lineales (XOR) a la derecha. Esta figura ha sido recreada a partir del trabajo de Rosebrock [2017]	80
7.2. Comparación entre una neurona humana (superior) y una artificial (inferior).	82
7.3. Ejemplo de espacio de pérdida generado por una Red Neuronal y sus múltiples parámetros. Imagen extraída de Li et al. [2018]	84
7.4. En estas gráficas se muestra una representación del resultado de dos Redes diferentes sobre el dataset de CIFAR-10.	92
7.5. Estructura original de <i>U-net</i> , imagen recreada a partir de la mostrada en Tomar [2021]	95
7.6. Estructuras de las diferentes variantes, extraídas de Daudt et al. [2018] . Las flechas negras indican las diferentes conexiones entre entradas y salidas, las rojas son los parámetros compartidos.	96

7.7. Estructura de MFCU, capaz de extraer características espacio-espectrales a múltiples escalas. Extraído de Chen et al. [2020]	98
7.8. Arquitectura de la red DSMS-FCN.	99
7.9. Arquitectura de la red <i>UNet++</i> mejorada.	100
7.10. Módulo convolucional aplicado en <i>UNet++</i>	101
8.1. Resultado del algoritmo en el dataset LEVIR con la imagen de prueba nº19, en la parte superior se encuentran los dos pares comparados en diferentes instantes; en la parte inferior se encuentra la máscara cedida por el propio autor (izq) y la máscara obtenida por el algoritmo (dcha). .	105
8.2. Muestra del funcionamiento de estos algoritmos sobre OSCD, en la parte inferior se observa el resultado y métricas obtenidas para <code>img_diff</code> (izq) y <code>compare_ssim</code> (dcha).	107
8.3. Ejemplo de funcionamiento de Canny sobre el dataset de OSCD.	109
8.4. Aquí se muestran unos de los pasos intermedios del algoritmo, en la primera fila se mostrará el resultado de Canny sobre la imagen original, mientras que en la parte inferior se muestran los contornos ya rellenados. .	110
8.5. Ejemplo de funcionamiento de Canny sobre un par de LEVIR.	111
8.6. Ejemplo de funcionamiento de Canny sobre CROP_FIELD	111
8.7. Gráfica con los resultados de Canny sobre los diferentes datasets.	112
8.8. Funcionamiento de PCA sobre el dataset de LEVIR.	113
8.9. Funcionamiento de PCA sobre el dataset de OSCD.	114
8.10. Funcionamiento de PCA sobre el dataset de CROP_FIELD (izq) y CROP_FIELD_2 (dcha).	114
8.11. Presentación visual de los resultados de PCA K-means.	115
8.12. Resultados obtenidos de los diferentes modelos sobre el dataset de OSCD.	117
8.13. Presentación visual de los resultados de Kappa de los diferentes modelos. .	117

8.14. En orden, resultados sobre los datasets de OSCD, LEVIR, CROP_FIELD y CROP_FIELD_2 con DSMS-FCN.	119
8.15. Presentación visual de los resultados de Kappa sobre DSMS-FCN. . .	119
8.16. Errores encontrados durante la ejecución de los diferentes algoritmos. En la parte inferior se menciona el número de epoch en el que fue extraída la imagen para el dataset de CROP_FIELD.	120
8.17. Ejemplos de resultados obtenidos por UNet++ mejorada sobre el dataset de LEVIR.	121

Lista de tablas

3.1. Especificaciones de los diferentes equipos usados	21
5.1. Tabla resumen de las características de las imágenes encontradas en el dataset de SZTAKI.	38
5.2. Tabla resumen de las características de las imágenes encontradas en el dataset de AICD	39
5.3. Resumen de las especificaciones de este dataset.	40
5.4. Resumen de las especificaciones de este dataset.	41
5.5. Resumen de las características de las imágenes encontradas en el dataset.	44
5.6. Características de las imágenes del dataset de LEVIR.	45
5.7. Tabla resumen con las especificaciones de Hi-UCD.	46
5.8. Resumen de las características de las instantáneas de SpaceNet.	46
8.1. Tamaños de los conjuntos de entrenamiento, testeo y validación de los diferentes datasets.	104
8.2. Resultados de los algoritmos en sus dos variantes.	106
8.3. Se muestra el resultado de los algoritmos aplicando diferente tamaño en los bloques de la umbralización adaptativa.	108
8.4. Parámetros usados para la ejecución de Canny.	109
8.5. Tabla con los resultados en las diferentes métricas del algoritmo de Canny	112

8.6. Resultados del algoritmo de PCA K-means aplicado a los diferentes datasets	112
8.7. Resultados de los diferentes modelos aplicados. Entre paréntesis se encuentran los mejores epochs.	116
8.8. Resultados DSMS-FCN sobre los datasets. Entre paréntesis se encuentra el número de epoch del cual se ha extraído el modelo.	118
8.9. Resultados de UNet++ mejorada sobre LEVIR.	120
8.10. <i>Ranking</i> de los diferentes métodos sobre los datasets.	122
8.11. Ranking con los diferentes métodos.	123
8.12. Tabla con los resultados del test de Nemenyi.	124

Lista de listados de código

6.1. Método para la aplicación del umbral.	58
6.2. Aplicación del umbral.	59
6.3. Pseudocódigo de Image Differencing.	59
6.4. Pseudocódigo de <i>Image Differencing</i> modificado.	61
6.5. Métodos para encontrar los contornos.	62
6.6. Método para rellenar los contornos.	62
6.7. Aplicación de los métodos de buscar y rellenar.	62
6.8. Aplicación del método de umbralización adaptativa.	63
6.9. Aplicación del método de umbralización adaptativa.	64
6.10. Pseudocódigo de <i>Image Differencing</i> con Umbralización adaptativa. . .	66
6.11. Sintaxis del método de Canny de OpenCV.	67
6.12. Sintaxis del método blur de OpenCV.	69
6.13. Pseudocódigo de <i>Canny</i>	70
6.14. Pseudocódigo de <i>Difference PCA</i>	73
6.15. Pseudocódigo de PCA K-Means	76

Capítulo 1

INTRODUCCIÓN

Este trabajo se centra en el campo de la **Detección de Cambios**. Durante el desarrollo del mismo se muestra el proceso de investigación realizado, tanto sobre el Aprendizaje Profundo, como de la Detección de cambios. Se pretende mostrar los pasos, las diferentes fases que se deben de seguir para desarrollar estas disciplinas. Para ello se deben explicar los diferentes conceptos que se pueden encontrar en la literatura sobre esta temática, así como ejemplos y una revisión sobre la actualidad de muchos de estos temas. Se puede encontrar en los apartados iniciales información sobre las áreas que se ven implicadas en la Detección de cambios, como por ejemplo la Ciencia de datos, Extracción de conocimiento de Bases de datos, Visión por Computador, etc. De ellas se pretende mostrar las diferentes técnicas y aplicaciones, así como las fases que siguen para su desarrollo.

Se explican los diferentes elementos a considerar para poder trabajar sobre este área, como por ejemplo las herramientas más utilizadas para aplicar Inteligencia Artificial, su explicación y razones de su uso. También se detallan las diferentes librerías existentes y sus múltiples usos. Se documenta la investigación sobre las diferentes métricas encontradas, como se calculan e interpretan, así como dónde encontrar una implementación de las mismas. Estas serán utilizadas en el posterior apartado de análisis de los diferentes algoritmos.

Otro de los elementos más importantes a la hora de trabajar en el área de la Detección de cambios son los datasets. Se realiza un extenso análisis de todos los datasets encontrados, extraídos tanto de la documentación como de una intensa exploración por la red. Tras revisarlos se pueden extraer ciertas problemáticas, las cuales serán comentadas y para las cuales se intentará dar una solución. A parte de todo esto, se añade un apartado en el que se documenta la creación de datasets sintéticos como

posible solución a estos problemas y a la escasez de datos. Durante la experimentación, los algoritmos serán aplicados a un par de datasets sintéticos, cuya generación y problemas serán explicados en el apartado correspondiente de datasets.

Una vez explicados los procedimientos y elementos necesarios en la Detección de cambios, se procede a la explicación de los diferentes algoritmos, tanto tradicionales como de Aprendizaje Profundo. Tras un largo proceso de documentación se procedió a la selección y explicación de los que se consideraron más destacables.

Para los métodos tradicionales se puede encontrar una explicación de cada uno de ellos, su funcionamiento, fases y se muestra un pseudocódigo de su funcionamiento. Posteriormente se comenta la implementación, ya que en la mayoría de los casos se tuvo que adaptar el código para su correcto funcionamiento con el entorno local, así como la implementación de alguno de ellos.

En el caso de los métodos que aplican Aprendizaje Profundo se realiza una extensa descripción de los fundamentos de las redes neuronales, de su funcionamiento y elementos, para que la explicación de los métodos sea fácil de entender por el lector. Tras esta introducción se procede a explicar cada una de las redes aplicadas, su estructura y el funcionamiento de estas, pretendiendo destacar que es lo que las diferencia de las demás.

Finalmente se realiza un proceso de experimentación sobre los diferentes métodos, así como un análisis de los mismos. Se muestran tablas resumen con las métricas obtenidas, así como imágenes con los resultados (las máscaras obtenidas de los diferentes algoritmos). Con estos resultados se realizan *rankings* y un análisis sobre los mejores algoritmos.

1.1. Motivación

Dentro de la Inteligencia Artificial se encuentran múltiples campos de investigación, como son el **Machine Learning** y el **Deep Learning**. Estas áreas se podrían definir como:

- **Inteligencia Artificial:** consiste en generar algoritmos y técnicas que puedan ser usadas para resolver problemas intuitivos y casi automáticos para los humanos, pero que resultan complejos para las máquinas. [Russell \[2006\]](#)
- **Aprendizaje Automático:** hace referencia a enseñar a una máquina a actuar

como lo haría un humano, a través de la experiencia. Estas aprenden a través de patrones extraídos directamente de las relaciones entre los datos. Se puede encontrar más información sobre este área en [James \[2018\]](#) y [MATLAB \[2021\]](#).

- **Aprendizaje Profundo:** una especialización del **Machine Learning**, en la que se utilizan redes de estructura profunda. Esta característica es lo que les permite extraer patrones complejos en diferentes tipos de datos. Para más información se pueden consultar las siguientes referencias [Kaggle \[2021\]](#) y [Witten and Frank \[2005\]](#).

Como se puede observar en la Figura 1.1, estos campos conforman el concepto de Inteligencia Artificial. El objetivo de estos reside en desarrollar diferentes técnicas que permita a las máquinas resolver problemas que para el razonamiento humano resultarían sencillos. Sin embargo, para una máquina requiere algo más de esfuerzo. Estas disciplinas llevan mucho tiempo siendo investigadas, y aún a día de hoy se debe de seguir trabajando en estas ya que queda mucho camino por recorrer.

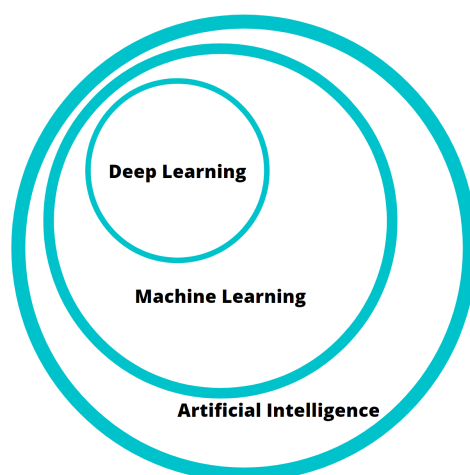


Figura 1.1: Esquema donde se puede observar los diferentes sub-campos de la Inteligencia Artificial. Imagen recreada a partir de una Figura de [Rosebrock \[2017\]](#).

Una de las disciplinas que más desafíos propone a los investigadores es la **Visión por computador**, entender y comprender el contenido de imágenes para que una máquina sea capaz de actuar ante ciertas situaciones, aprender de estas y ser capaz de reconocer patrones (definición extraída de [Rosebrock \[2017\]](#)). Estas tareas son fáciles de realizar por los humanos, ya que a simple vista somos capaces de reconocer estos patrones, de extraer el contenido de lo que se ve, entender la situación y actuar en consecuencia, algo que para un ordenador resulta una tarea mucho más compleja, la cual debe de aprender.

Para llevar a cabo esta tarea se han ido recurriendo a diferentes métodos tales como los clásicos, aquellos que se aplican a nivel de píxel ([Minu and Shetty \[2015\]](#)), o

métodos dentro del **Machine Learning** ([MATLAB \[2021\]](#)). De estos últimos se pueden destacar:

- Las **Redes Neuronales profundas**, unos algoritmos capaces de aprender de estos datos, de los patrones que encuentran en las imágenes al igual que lo haría un cerebro humano.
- Las **Máquinas de vectores de soporte (SVM)**, un modelo que representa los puntos de datos en el espacio e intenta separarlos en diferentes categorías mediante un hiperplano.

La **Visión por computador** ([Szeliski \[2011\]](#)) es de gran importancia en muchos aspectos, ya que, aunque no lo parezca, se encuentra en todo el entorno, aplicado sobre diferentes sistemas o dispositivos. Puede ser aplicada a cualquier proceso que implique analizar el contenido de información visual de manera automatizada, teniendo como objetivo facilitar determinadas tareas realizadas por los seres humanos, como por ejemplo:

- **Seguridad**: permitiendo construir sistemas automatizados que no requieran la presencia de una persona para detectar intrusiones.
- **Medicina**: con sistemas que detecten enfermedades o ayuden al seguimiento de estas a través del estudio de diferentes imágenes tales como radiografías.
- **Industria**: para la detección y eliminación de piezas defectuosas.
- **Automovilismo**: a la hora de desarrollar coches autónomos

Como se puede observar, hay numerosas aplicaciones de este campo actualmente, y a medida que crece la capacidad computacional de los diferentes dispositivos, se producen grandes mejoras en este ámbito. Y es que esta tecnología no solo se aplica en campos profesionales, sino que aparece en diferentes aplicaciones móviles o sistemas que son usados en gran medida por una gran cantidad de usuarios en su día a día.

Del artículo [Research \[2020\]](#) se pueden extraer diferentes datos financieros o de uso de estos sistemas. Más concretamente, se puede observar el impacto financiero que tendrá, así como una previsión de mercado para 2027, donde indican que se producirá una crecida anual del 7.6%. Se puede observar en la Figura 1.2, en la parte superior, una gráfica donde se muestra la cuota de mercado de las diferentes aplicaciones en 2019 comparada con la previsión de 2027, donde se muestra que el

campo de Posición y Orientación (Automovilismo), ganará una mayor importancia con respecto a la Industria. En la parte inferior se muestra una gráfica con la previsión del presupuesto invertido en la Visión por Computador, llegando a darle cada año una mayor importancia a este cambio, debido a las múltiples aplicaciones que tiene.

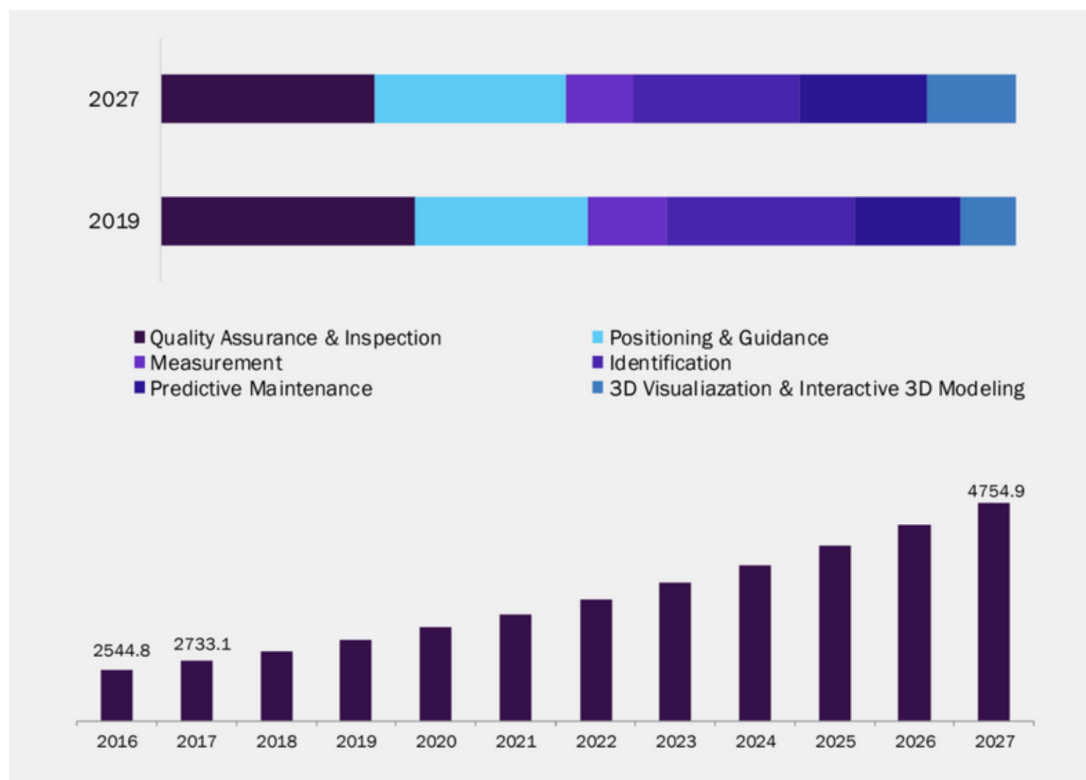


Figura 1.2: Gráficas extraídas de [Research \[2020\]](#), donde se muestra la evolución financiera de la Visión por computador sobre diferentes aplicaciones.

En este trabajo se pretende realizar una amplia revisión sobre el campo de la **Detección de Cambios en Imágenes**, el proceso de identificar cambios de interés observando un mismo escenario en diferentes periodos temporales. Es una técnica muy importante hoy en día y tiene múltiples aplicaciones, como ya se comentarán más adelante. Esta revisión comprenderá desde sus datasets, técnicas de pre-procesamiento de las imágenes, métricas aplicadas y técnicas, desde clásicas hasta algunas de las últimas propuestas en algoritmos de Aprendizaje Profundo. Para comprobar todo esto se realizará un estudio y experimentación sobre los diferentes métodos planteados para construir así una conclusión del estado actual y una visión de la evolución entre estos métodos.

1.2. Objetivos

Los objetivos seguidos para este trabajo son los siguientes:

- Estudiar el campo de la detección de cambios en imágenes.
- Identificar y analizar diferentes técnicas para la detección de cambios en imágenes, incluyendo preferentemente algunas basadas en Aprendizaje Profundo.
- Comparar y Analizar el funcionamiento de técnicas de detección para las que existan implementaciones disponibles
- Proponer posibles mejoras en el campo y estudiar su impacto.

A pesar de estos, también se ha querido orientar el trabajo para ayudar a aquellos que se quieran introducir en la investigación de este campo, por lo que se deben añadir unos cuantos objetivos más:

- Explicar los diferentes elementos que componen esta disciplina tales como conjuntos datos, tipos de imágenes, métricas...
- Explicar cómo preparar el entorno para poder investigar en este campo
- Mostrar las problemáticas existentes y en algunos casos las posibles soluciones.

Para desarrollar estos objetivos se debe aplicar una estructura concreta de los contenidos, para poder introducir al lector poco a poco en el campo de la Detección de Cambios y todos sus elementos claves.

1.3. Estructura del documento

En un primer momento se encontrará una descripción del **Estado del arte**, en qué punto se encuentra actualmente este campo. Junto a él se intentará describir los pasos generales que siguen este tipo de algoritmos, así como las diferentes problemáticas que influyen de forma importante en este campo y por las cuales aún debe de seguirse invirtiendo tiempo de investigación en este.

Justo después se encontrará un apartado dedicado a las **herramientas utilizadas** con las cuales se ha trabajado para este proyecto, ya que considero que es una de las partes más complejas y en la que más horas se invierte.

Tras explicar estas herramientas comunes para preparar el entorno, se muestra un recopilatorio de los **datasets** encontrados con máscaras para aplicar en este campo. Al final en el estudio no se usarán todos los datasets descritos, lo que se pretende es mostrar la cantidad de conjuntos de datos existentes y mostrando de primera mano el principal problema que se encuentra en este campo de investigación. En referencia a esto último también se propondrán unas posibles soluciones, como datasets generados sintéticamente, incluso llegando a aplicar alguna de ellas para la posterior experimentación.

Otro de los elementos importantes a explicar serían las **métricas**. En muchos otros documentos presentan métricas como precisión o Kappa y no todos ellos explican como interpretarlas, cuál es su significado. Se explican las diferentes métricas que se suelen aplicar para medir el rendimiento de los diferentes modelos o algoritmos que se encuentran y que ayudarán a compararlos.

Tras todos estos elementos se introducirán los diferentes **métodos aplicados**, describiendo su funcionamiento detalladamente junto con ejemplos que demuestran su funcionamiento sobre determinados conjuntos de datos. Algunos de estos algoritmos cuentan con una implementación propia que será también comentada.

Finalmente se expondrán y analizarán los diferentes **resultados obtenidos** por cada algoritmo aplicados sobre los datasets seleccionados y comparados gracias a las diferentes métricas estudiadas.

1.4. Planificación

A continuación, en la Figura (1.3) se presenta un esquema de la planificación del desarrollo de este trabajo. La idea general de la planificación fue en un principio documentarse y obtener información de diversos artículos, para posteriormente, recabar los diferentes algoritmos y probar diversas ejecuciones para asegurar el correcto funcionamiento de estos y saber si los algoritmos debían ser adaptados al entorno de trabajo o requerían de alguna modificación.

Tras realizar una documentación base sobre el campo y los diferentes conceptos que abarca el Aprendizaje Profundo, y sobre todo, del campo de la Detección de cambios, se procedió a realizar una preparación del entorno de trabajo en el equipo local para poder trabajar con los diferentes algoritmos.

Durante la documentación sobre la Visión por computador, se procedió a la inves-

tigación sobre los diferentes datasets existentes; tanto en la literatura, como aquellos que podían ser encontrados en la red. El objetivo era encontrar el mayor número posible, para posteriormente realizar una selección y preprocesamiento de los mismos. Aparte de todo esto, también se estuvo experimentando con el desarrollo de datasets sintéticos que serán utilizados en el proceso de experimentación de los diferentes métodos estudiados.

Una vez concluida la instalación y preparación de este entorno de trabajo, se prosigue con la documentación sobre los diferentes métodos tradicionales que se pueden encontrar y una búsqueda y recopilación de diferentes algoritmos, los cuales se deben de adaptar para que estos puedan ser ejecutados correctamente. Una vez finalizado este proceso de adaptación, se procede a realizar la experimentación con los diferentes datasets seleccionados y el posterior análisis de estos.

Para los métodos con Aprendizaje Profundo se realiza un proceso similar, se documenta, se busca implementación sobre los mismos y se procede a experimentar con ellos. Una vez realizados los entrenamientos y ejecuciones de los algoritmos, se procede a realizar un análisis y comparación con los métodos tradicionales.

Como se puede observar en la Figura (1.3), durante todo el proceso de desarrollo de este trabajo, se ha ido realizando documentación en cada una de las etapas. El objetivo era tener una documentación completa que detallara todas las partes del proceso de la Detección de cambios.

Semana												
	1	2	3	4	5	6	7	8	9	10	11	12
Lectura de documentación	Introducción al Deep Learning	Introducción a la Visión por computador	Lectura de artículos relacionados	Documentación sobre métodos tradicionales				Documentación sobre métodos con Aprendizaje Profundo				
Preparación del entorno de ejecución				Búsqueda de herramientas e instalación del entorno								
Búsqueda, creación y pre-procesamiento de Datasets		Búsqueda de datasets	Desarrollo de datasets sintéticos	Selección y pre-procesamiento de los datasets								
Métodos tradicionales				Investigación y búsqueda de implementaciones	Investigación y búsqueda de implementaciones	Experimentación y documentación de los métodos	Análisis					
Métodos con Aprendizaje Profundo								Investigación y búsqueda de implementaciones	Experimentación y documentación de los métodos	Análisis		
Documentación	Desarrollo de la documentación										Aplicación de correcciones	Desarrollo de la presentación

Figura 1.3: Planificación seguida para el desarrollo de este proyecto.

Capítulo 2

ESTADO DEL ARTE

En este capítulo se procede a mostrar una breve revisión sobre la **Ciencia de datos** y la **Visión por computador**, el proceso, las diferentes tareas y técnicas que se han ido aplicando así como problemáticas actuales encontradas.

2.1. Ciencia de datos

Es la encargada de explotar datos para generar conocimientos a partir de estos. El principal objetivo reside en encontrar modelos sean capaces de encontrar patrones entre estos datos para extraer conocimiento, permitiéndoles así ayudar a la hora de tomar decisiones o de hacer predicciones.

En el caso concreto de este trabajo, se pretende aplicar esto sobre pares de imágenes, extraer conocimiento de estas para detectar cuando se han producido cambios. Para esto se debe recurrir a **Knowledge Discovery in Databases** (KDD).

2.1.1. Extracción de conocimiento de bases de datos

Es el proceso de aplicar esta extracción sobre bases de datos, sobre grandes conjuntos de información, para obtener una interpretación de estos y así encontrar patrones. [Frawley et al. \[1992\]](#)

Para desarrollar este proceso, se siguen unas fases que se encuentran bien definidas en [García et al. \[2018\]](#).

- Fase 1: **Comprensión del dominio del estudio**, consolidar los objetivos que se van a seguir. Para ello se necesita identificar diferentes fuentes sobre las que aplicar esta extracción de conocimiento.
- Fase 2: **Preparación de la base de datos**. Una vez descubiertas las diferentes fuentes, se debe de hacer una selección de estas, formatearlas y prepararlas para que todas sean homogéneas y por consiguiente, más fáciles de procesar y analizar.
- Fase 3: **Limpieza y pre-procesamiento de los datos**. El objetivo es disponer de datos de calidad, para ello se deben de eliminar datos para evitar redundancia, la aparición de ruido, de datos anómalos o incompletos que puedan afectar gravemente al proceso de extracción.
- Fase 4: **Minería de datos**. El proceso de búsqueda de patrones sobre los datos, las relaciones existentes entre estos, haciendo uso de diferentes algoritmos.
- Fase 5: **Interpretación de los patrones**. Se analizan los resultados obtenidos, pudiendo incluso tener que volver a realizar alguno de los pasos anteriores si estos no son concluyentes.
- Fase 6: **Consolidación del conocimiento**. Una vez extraído el conocimiento, debe de ser aplicado, siguiendo los objetivos planteados inicialmente.

De las fases anteriores se debe de destacar la Fase 4 sobre minería de datos. Este análisis puede realizarse siguiendo diferentes enfoques. Si se desarrolla un análisis descriptivo, se quiere conocer una visual del estado actual en el que se encuentra el dominio de estudio, para ello utiliza datos históricos; mientras que si se desarrolla el enfoque predictivo, lo que se pretende es conseguir una visión futura sobre el dominio a partir de los datos que se poseen en determinado momento. Para ambos se aplican diferentes algoritmos estadísticos o de Aprendizaje Profundo.

El funcionamiento de estos algoritmos de Aprendizaje Profundo que aplican diferentes técnicas de **Machine Learning** serán explicados más adelante, más concretamente los diferentes tipos de aprendizaje que se pueden aplicar sobre estos datos se encuentra en la Sección [7.1.3](#).

2.1.2. Tareas

En Minería de Datos se distinguen diferentes tareas en función del objetivo para el que se desarrolle el proceso de extracción de conocimiento. Entre ellas destacan es-

pecialmente las siguientes, cuyas descripciones se pueden encontrar en [Álvaro \[2018\]](#):

- **Clasificación:** dentro de un conjunto de datos se pueden localizar diferentes categorías, el objetivo es separar los diferentes datos y asignarles correctamente la categoría a la que pertenecen.
- **Regresión:** son problemas que pretenden predecir un valor Y a partir de unos valores conocidos X . El ejemplo más sencillo sería la regresión lineal, donde dado un conjunto de puntos en el espacio, se pretende encontrar el hiperplano que se ajuste óptimamente a estos.
- **Agrupamiento:** o **Clustering** en inglés, es una técnica que busca conjuntos o subconjuntos de datos. Se pretende que estos grupos sean muy diferentes entre ellos, pero que los elementos de estos sean muy similares.
- **Predicción de series temporales:** un problema de regresión donde los datos además cuentan con cierta dependencia temporal.

2.1.3. Técnicas

Para desarrollar algoritmos enfocados en estas tareas se pueden utilizar diferentes enfoques o técnicas [Maimon and Rokach \[2010\]](#). Entre ellas se destacan, por su amplio uso, las siguientes:

- **Lógica difusa:** es una técnica que permite obtener conclusiones sobre un problema cuando se tienen datos ambiguos, con ruido, incompletos, etc. Pretende imitar la lógica humana cuando, en determinadas circunstancias, cuenta con datos de estas características.
- **Árboles de decisión:** esta técnica permite tomar decisiones secuenciales analizando las características de los datos. Al final, en las diferentes ramas se agrupan aquellos datos que tienen propiedades similares.
- **Máquinas de soporte Vectorial:** explicadas anteriormente en [1 INTRODUCCIÓN](#).
- **Redes Neuronales:** basadas en el funcionamiento del cerebro humano, son conjuntos de neuronas artificiales, organizadas en capas, que les permiten realizar predicciones.

Dentro de todas estas técnicas se va a destacar las Redes Neuronales, debido a que en la última década han conseguido grandes avances en todos las tareas citadas anteriormente. Se podrá encontrar un explicación más extensa de estas más adelante, en la sección [7.1 Fundamentos de las Redes Neuronales](#).

2.2. Visión por computador

La **Visión por computador** ([Rosebrock \[2017\]](#)) pretende conseguir que una máquina visualice y entienda su entorno, que comprenda el contenido de las imágenes para que esta sea capaz de dar una respuesta adecuada a un problema. Para ello debe de “aprender” de su entorno, de otras imágenes y situaciones similares.

El elemento de trabajo en este campo son las **imágenes**, las cuales se pueden definir como matrices que para cada posición almacenan un valor de intensidad, y cada una de estas posiciones se corresponderían con un píxel. Estas imágenes pueden ser clasificadas según el número de bandas que contengan y los colores que alberguen. Normalmente se suele trabajar con imágenes RGB, de solo 3 bandas (Rojo, Azul y Verde). De otra parte, las imágenes satelitales cuentan con numerosas bandas, cada cual aporta diferentes datos al trabajar en diferentes longitudes de onda dentro del espectro de la luz no visible (como por ejemplo en el espectro infrarrojo o ultravioleta), estas son conocidas como imágenes **Hiper-espectrales**. Las anteriores son grandes cubos volumétricos con cientos de bandas, a veces la información de estas es redundante y difícil de procesar, por ello se suele recurrir a las imágenes **Multiespectrales** (de menos de 15 bandas).

Para desempeñar este estudio, se recurren a **métodos tradicionales** basados en operaciones a nivel de píxel, imagen a imagen, realizando diferentes transformaciones y operaciones matemáticas sobre las imágenes para terminar calculando una imagen de diferencia. También surgieron diferentes métodos que aplicaban **Machine Learning**, tales como algoritmos de *clustering* o la ya mencionada SVM. Con los **métodos de aprendizaje automático** se pretende superar a los tradicionales entrenando diferentes modelos, los cuales no solo trabajan a nivel de píxel, sino que tienen en cuenta todo el conjunto de imágenes para aprender de estas.

Con estas imágenes, los modelos son capaces de resolver múltiples tareas relacionados con el tratamiento de estas, tales como **problemas de clasificación**, donde se asignan categorías a las diferentes imágenes. Los problemas de clasificación cuentan con grandes avances por ejemplo en el ámbito de la medicina, para detectar cuerpos

extraños en radiografías. Pero no solo esto, también se han desarrollado numerosos algoritmos que permiten resolver **problemas de segmentación**, en los que el objetivo es marcar los objetos que aparecen en una imagen y en que posición, pudiendo llegar a ser muy útil para personas con discapacidad visual, para su aplicación en coches autónomos o para videovigilancia. Una tarea muy relacionada con la anterior, sería la **detección de objetos en imágenes**, en la que se pretende marcar con precisión la posición de determinados elementos si estos se dan en una imagen. Otro de los usos que tienen estas redes es en la **detección de cambios** en imágenes, en la cual consiste este trabajo y que se centrará en las diferentes soluciones propuestas hasta ahora. Se pueden encontrar muchas más aplicaciones y las definiciones de las mismas en [Szeliski \[2011\]](#).

2.3. Detección de cambios en imágenes

La **Detección de Cambios** es el proceso de identificar cambios de interés observando un mismo escenario en diferentes periodos temporales, es decir, observar un par de imágenes separadas temporalmente para establecer los cambios que han ocurrido entre ellas. Se puede observar un ejemplo de aplicación de este proceso en la Figura 2.1, donde se observan dos pares de imágenes entre las cuales existe una diferencia, el objetivo de los algoritmos para la detección de cambios es dar con la máscara de cambio que se muestra en la parte derecha de dicha figura. Este campo supone un gran reto, ya que a diferencia del ojo humano que es capaz de detectar estos cambios, cualquier computador necesita invertir un mayor esfuerzo para marcar con precisión estas modificaciones. Estas no son siempre a nivel de píxel y fáciles de visualizar, sino que muchas veces también se producen cambios semánticos como ocurre con la vegetación, que dependiendo de la estación pueden presentar diferentes estados, pero que cambien de color no significa que hayan sufrido este cambio de estado. Estos tipos de cambios son más difíciles de señalar por una computadora.

Todas las técnicas siguen en mayor o menor medida un esquema de funcionamiento muy sencillo tal y como se puede leer en la literatura sobre este tema. Todos los autores coinciden en que la detección de cambios se desarrolla siguiendo 3 pasos:

1. **Pre procesamiento de las imágenes.** Este primer paso se debe a que, como se verá en el siguiente apartado de [Problemática asociada a la detección de cambios en imágenes](#), dependiendo del método elegido para extraer las instantáneas del entorno, estas se pueden encontrar en las mismas o diferentes condiciones tanto de iluminación como alineación en su plano o en situación espacial. Lo que

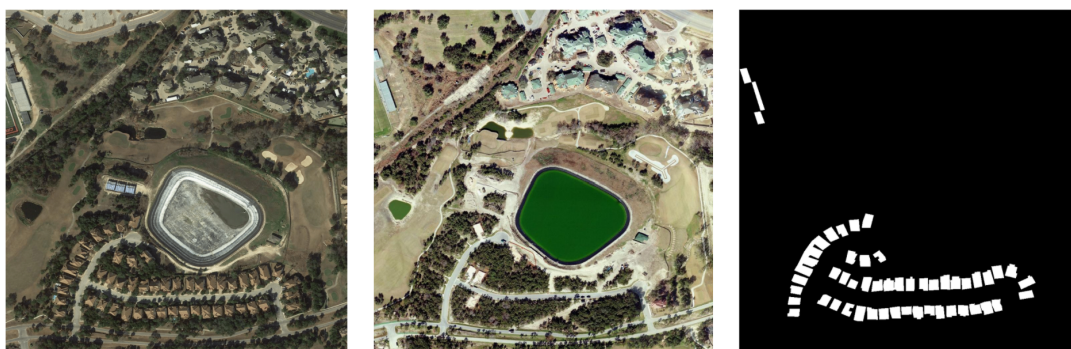


Figura 2.1: Ejemplo de lo que se pretende obtener tras aplicar los algoritmos de detección de cambios.

ocurre es que se obtienen imágenes, que aun perteneciendo al mismo área, se encuentran desplazadas con respecto a la que se tomó en un primer momento o con una incidencia solar diferente. Esto resultará en un grave problema a la hora de realizar el análisis de las mismas.

2. **Obtener un mapa de características.** De cada imagen se procurará extraer las características que destaquen. En los métodos tradicionales puede ser una simple umbralización de la imagen o la aplicación de un algoritmo como podría ser Canny ([Valverde-Rebaza \[2007\]](#)) para la detección de bordes (más adelante se tratará con un método que aplica este mismo algoritmo [6.2](#)). Con los métodos de Aprendizaje Profundo se encontrará un proceso similar, a cada capa en la que se vaya profundizando más detalles a bajo nivel se extraerán. Se empezará por aquello que es más llamativo, como grandes contornos, para ir extrayendo poco a poco detalles más concretos.
3. **Obtener el mapa de cambio.** Se estudiarán numerosos métodos de extracción de máscaras de cambio, algunas funcionarán a nivel de píxel, otras usarán el contexto, la vecindad entre píxeles para obtener si se ha producido un cambio o no. Estos serán marcados en una máscara donde se indicará a nivel de píxel si se ha producido cambio o no. En este proceso también se pueden aplicar diferentes algoritmos de *clustering* o agrupamiento, que como bien indica su nombre agrupará los cambios para marcarlos como tal.

2.4. Métodos para la Detección de cambios

Aunque se van a desarrollar, se procede a realizar una breve descripción de los diferentes métodos estudiados. En primer lugar se encontrarán los métodos tradicionales en el Capítulo [6](#), cuyos resultados serán comentados y comparados con los de

Aprendizaje Profundo, los cuales se encuentran en el Capítulo 7. Estos métodos extraen características a bajo nivel, ya que no cuentan con la información de su entorno para calcular las zonas de cambio ni son capaces de detectar los elementos clave. A lo largo de la literatura sobre la detección de cambios se pueden encontrar numerosos métodos, para algunos se obtuvieron muy buenos resultados, aún así en todos se coincide que han sido más que superados por los métodos basados en Inteligencia Artificial.

En un primer momento se podrá encontrar la descripción y análisis de **Image Differencing** (Minu and Shetty [2015]), un algoritmo bastante sencillo que simplemente resta a píxel por píxel para obtener así la imagen de diferencia. Como se puede deducir, este método no obtiene muy buenos resultados, algo que será comprobado en 6.1 **Image Differencing**.

A continuación se probará con una experimentación propia, utilizando para ello el método de **Canny** (Valverde-Rebaza [2007]) citado anteriormente para la detección de bordes.

Por último se tratará otro método bastante destacable en este campo, **Principal Component analysis method (PCA)** (Jolliffe and Cadima [2016]), cuya explicación se puede leer en el trabajo realizado por Celik [2009]. Más adelante se dará una definición a este método y los diferentes resultados que se obtienen en 6.4 **PCA K-Means**.

Otro de los métodos que mejor resultados ha obtenido en este campo es **Change vector analysis (CVA)** (author et al. [2004]). Como bien indica su nombre para cada píxel se obtiene un vector de cambio. Este vector es el encargado de indicar el tipo de cambio según su dirección y la intensidad del cambio según su magnitud. Para aplicar este método se necesita unas bandas muy concretas para calcular unos índices de reflectancia, por lo que a la hora de desarrollar este trabajo, no se ha podido probar; sin embargo, se pueden observar los resultados de este en el trabajo de Lorena et al..

También se puede destacar **Tasseled cap transformation (TCT)**, el cual obtiene datos sobre diferentes índices tales como Luminosidad, Verdor, Humedad, etc. Con este método se calcula la diferencia entre los índices del par de imágenes, obteniendo así un mapa de cambio. De nuevo se tiene el problema de que las bandas RGB no son suficientes, por lo que este método tampoco ha podido ser comprobado, pero en Huang et al. [2002] se podrá observar las ventajas y desventajas de este método, el cual tiene problemas si no se puede aplicar la corrección atmosférica y aparecen imágenes demasiado iluminadas.

También se encuentran diferentes métodos clásicos que aplican **Machine Lear-**

ning, entre ellos se puede destacar **SVM**, un método que tiene como objetivo clasificar puntos de datos distribuidos sobre un espacio, separando estos mediante un hiperplano.

En lo referente a modelos de Aprendizaje Profundo, se encuentran numerosos estudios que aplican diferentes técnicas y arquitecturas para hallar cambios en imágenes. La mayoría de estas redes están basadas en arquitecturas como las **Siamesas** o las **Encoder-Decoder**, ya que estas han demostrado tener un buen rendimiento en este campo. Un ejemplo basado en estas arquitecturas se puede encontrar en [de Jong and Bosman \[2019\]](#), donde se presenta una Encoder-Decoder para la clasificación semántica y la detección de cambios, en este trabajo también se presentan posibles soluciones para aumentar el tamaño de los datasets y las consecuencias de estos.

En [Wiratama et al. \[2018\]](#) se puede encontrar una red Siamesa, es decir dos arquitecturas idénticas que trabajan en paralelo, y con la que también se obtienen muy buenos resultados frente a los tradicionales.

En este trabajo se desarrollará la experimentación con las redes expuestas en [Chen et al. \[2020\]](#) donde se usan dos redes que combinan Encoder-Decoder con Siamesas y otra basada en la arquitectura Unet. Se harán experimentaciones y se comentarán los resultados de los tres modelos.

2.5. Problemática asociada a la detección de cambios en imágenes

Aunque el campo de la detección de cambios pueda parecer sencillo, como ya se adelantaba en el apartado anterior, se encontrarán numerosos problemas durante el proceso de extracción de conocimiento. Todas estas dificultades serán relativas a los conjuntos de datos, las instantáneas sobre las que aplicar el estudio.

La tendencia actual lleva a almacenar cualquier tipo de información y en grandes cantidades ya que no se encuentra prácticamente ningún límite a la hora de generar esta información y almacenarla. Todo esto se traduce en la existencia de numerosas imágenes extraídas por los satélites y que son fáciles de encontrar en Internet. El problema aquí reside en que no toda esta información es útil.

Uno de los principales problemas se encuentra a la hora de obtener datasets que contengan **máscaras de cambio** sugeridas por los autores. Como se comentará en

DATASETS, existen bastantes conjuntos de datos que incluyen estos mapas de cambio; sin embargo, estos no se encuentran completos ya que solo marcarán cambios de determinado tipo. Estos son muy útiles para aplicar a una aplicación muy concreta, la expansión urbana, ya que la mayoría de los autores marcan modificaciones en las diferentes construcciones que aparecen en estos pares. Pero si se quieren detectar realmente todos los cambios producidos en las instantáneas se debe de considerar también los cambios semánticos en campos de cultivo o en vegetación para poder detectar deforestación o el efecto causado por diferentes desastres naturales. A la hora de trabajar con estos algoritmos se debe poder demostrar que estos funcionan correctamente, o en el caso de los métodos con Aprendizaje Profundo, se debe poder ser capaces de enseñarles los patrones que se buscan, o como deben de detectar los cambios de estas imágenes. Por lo cual si se quiere enfocar el estudio hacia otro ámbito que no aparezca en estas máscaras se encontrarán numerosos problemas, sobre todo si se quiere calcular con diferentes métricas el funcionamiento del algoritmo escogido.

También cabe remarcar las condiciones en las que se toman las imágenes, ya que el método de obtención de estas no es muy exacto. Se debe considerar que estas se toman con satélites a la superficie terrestre, ambos elementos se encuentran en constante movimiento y el tiempo también juega un papel muy importante. Se encuentran entonces instantáneas tomadas sobre la misma región pero ligeramente **desplazadas** hacia un lado, incluso en un **plano diferente** al original; habrá imágenes tomadas en diferentes estaciones o momentos del día, lo cual se traducirá a cambios en las **intensidades de los colores** debidos a la incidencia de la luz y la aparición de sombras. Las **condiciones climatológicas**, como la aparición de niebla o nubes, también supondrán un riesgo para el funcionamiento de estos algoritmos. Más adelante se encuentra un apartado donde se intentará dar solución a todas estas problemáticas comentadas (ver [5.4 Pre-procesamiento](#)).

Capítulo 3

HERRAMIENTAS UTILIZADAS

A la hora de trabajar con las diferentes técnicas aplicadas a la Detección de Cambios se debe de contar con herramientas que permitan realizar experimentaciones con estas, para comprobar de primera mano el funcionamiento de las mismas. Cabe destacar que para el testeo de las mismas, se utilizó un equipo personal en el caso de las técnicas tradicionales; mientras que para las técnicas de Aprendizaje Profundo se utilizó un servidor del grupo de investigación **SIMIDAT** de la Universidad de Jaén. En ambos casos se usaron las mismas herramientas, ya que se escogieron las más usadas en este área actualmente.

3.1. Especificaciones

Para las diferentes ejecuciones se utilizaron equipos, cuyas especificaciones aparecen en la Tabla 3.1.

Equipo 1	
CPU	Intel(R) Core(TM) i7-7700K CPU 4.20 GHz 4 Cores
Almacenamiento	HDD 1TB + SSD 240 GB
GPU	Asus GeForce GTX 1050 Ti 4GB GDDR5
Memoria (RAM)	RAM 2 X 8 GB DDR4 (16 GB TOTAL)
Servidor	
CPU	2 Intel(R) Xeon(R) Silver 4214 2,2GHz 17MB Cache 12 Cores
Almacenamiento	HDD-2 2x HD 4TB, SATA 6 Gb/s 7.200 r.p.m 3,5 128MB Enterprise Storage (RAID0)
GPU	NVIDIA(R) TESLA V100 32GB Cuda Cores : 5120 Tensor
Memoria (RAM)	RAM 4 X 32 GB DDR4-2933 MHz ECC REG (128 GB TOTAL)

Tabla. 3.1: Especificaciones de los diferentes equipos usados

3.2. Python

Para el desarrollo del proyecto se utiliza en todo momento el lenguaje de programación Python, pero no solo por ser de propósito general y sencillo; si no porque a la hora de aplicarlo a la Ciencia de Datos, es de los más usados. Esto se debe a la cantidad de módulos y librerías, sumado a la sencillez del mismo, permitiendo que muchas personas sin previo conocimiento en la programación puedan introducirse en este campo realizando sencillos *scripts* pero que realmente cuentan con un gran potencial.

Si se observan diferentes *rankings* donde se muestran los lenguajes de programación (ver Figura 3.1) más usados, se puede encontrar en la mayoría de ellos a Python en segunda posición, bastante cerca de tomar el primer puesto. Esto se debe a lo citado anteriormente, muchos profesionales, sobretodo científicos se introducen en el mundo de la programación para poder desarrollar diversas experimentaciones, pero no solo en el ámbito de la Ciencia de Datos, sino que también se realizan estudios físicos, químicos, etc. Su sencillez permite centrarse en comprender las bases teóricas de lo que se esté implementando en lugar de encontrar un código complejo y difícil de entender.

Jun 2021	Jun 2020	Change	Programming Language	Ratings	Change
1	1		 C	12.54%	-4.65%
2	3	▲	 Python	11.84%	+3.48%
3	2	▼	 Java	11.54%	-4.56%
4	4		 C++	7.36%	+1.41%

Figura 3.1: Tabla extraída de [Tiobe \[2021\]](#) donde se puede ver el auge de este lenguaje con respecto al último año, situándose en segunda posición.

Más adelante se explicarán las diferentes librerías y módulos utilizados para la ejecución de los diferentes modelos, todas ellas están perfectamente integradas en el entorno de Python.

3.3. Librerías

A la hora de trabajar con Python para Ciencia de datos, se necesita instalar en el equipo diferentes librerías especializadas, que aporten diferentes funcionalidades al código y que lo hagan más breve y conciso. Estas herramientas se describen en las siguientes subsecciones.

3.3.1. OpenCV

Como ya se comentaba anteriormente, uno de los elementos más importantes para este campo de la Detección de cambios son las imágenes. Para poder tratar con ellas se debe recurrir a esta librería, la cual cuenta con cientos de funciones aplicadas a la visión por computador. Como se verá más adelante, se recurrirá a esta librería numerosas veces, ya que automatiza una gran cantidad de procesos tales como umbralizar, aplicar el algoritmo de Canny, mostrar y almacenar imágenes, etc. Además se cuenta con una extensa documentación ([OpenCV \[2021\]](#)) y diferentes tutoriales que pueden ser encontrados por Internet.

Para su instalación se debe de insertar el código que se muestra en el Listado 3.3.1 en la terminal del entorno, pudiendo indicar el número de la versión a continuación.

```
1 pip install opencv-python==version
```

3.3.2. Numpy

Otra de las librerías más usadas en este ámbito es Numpy, un módulo que aporta mejoras a la hora de tratar con vectores y matrices de gran tamaño, volviéndolas estructuras más eficientes. Esta surgió como una incorporación de diferentes competencias matemáticas a Python, ya que este originalmente no estaba pensado para Ciencia de datos. Incorpora numerosas funciones matemáticas especializadas para trabajar con estos conjuntos de datos de una forma bastante eficiente.

Su instalación es bastante sencilla, de nuevo basta con insertar el código mostrado en el Listado 3.3.2 en la terminal del entorno con la versión deseada. Se puede extraer más información de este en [Numpy \[2020\]](#)

```
1 pip install numpy==version
```

3.3.3. Scikit-learn

También conocida como Sklearn, es una biblioteca de aprendizaje automático que aporta diferentes funcionalidades relacionadas con este campo. Su funcionamiento

está basado en otras librerías como Numpy o Matplotlib (otra librería especializada en recrear gráficas), es decir, a partir de estas crea otras funciones como clasificadores, algoritmos de agrupamiento, regresión, etc. Es una manera sencilla para poder introducirse en el análisis de datos, ya que aparte de lo mencionado, también aporta numerosos datasets clásicos para la Visión por computador. Para este proyecto su uso se ha centrado en calcular las métricas, ya que cuenta con un gran repertorio de estas.

Para la instalación de esta solo se debe introducir el comando mostrado en el Listado 3.3.3. Para más información se puede consultar su página oficial [scikit learn \[2021\]](#)

```
1 pip install scikit-learn==version
```

3.3.4. Keras

Es una API (interfaz de uso intuitivo) sencilla que permite implementar de manera sencilla diferentes Redes Neuronales, es capaz de reducir la carga cognitiva del código para poder centrar la atención en su funcionamiento sin tener que conocer la complejidad de los diferentes *frameworks* de Aprendizaje Profundo como **TensorFlow** o **Theano**. Cabe destacar que cuenta con soporte para varias GPU, por lo que los procesos son fácilmente paralelizables. Se pueden observar todas sus características en su extensa documentación [Keras \[2021\]](#)

En el Listado 3.3.4 se encuentra el comando de instalación.

```
1 pip install keras==version
```

3.3.5. TensorFlow

Por último se quiere mencionar esta biblioteca de código abierto para el desarrollo de los diferentes modelos de Visión por computador. Actualmente es una de las más usadas debido al gran potencial que posee. Esta herramienta realiza diferentes operaciones matemáticas aplicando gráficos de flujos de datos, en el que cada nodo

representa una operación matemática, mientras que los tensores serían las conexiones entre estos, los datos.

Keras utiliza esta librería como *backend*, es decir, por debajo de Keras trabaja esta herramienta, mientras que muestra una API sencilla.

A la hora de comprobar los diferentes métodos de Aprendizaje Profundo aplicados a la detección de cambios, se encontrará que la mayoría de estos trabajan con TensorFlow, el cual ya incorpora una dependencia con Keras, por lo que este último no necesita ser instalado.

En el Listado 3.3.5 se encuentra la instalación de TensorFlow, tanto para ejecutar este backend en GPU (Graphics Processing Unit) como en CPU (Central Processing Unit).

```
1 pip install tensorflow-gpu
2 pip install tensorflow
```


Capítulo 4

MÉTRICAS

Antes de comenzar con la descripción y análisis de los diferentes métodos y datasets estudiados, se procede a explicar las diferentes métricas usadas para medir el rendimiento de estos.

Para poder calcular estas métricas se necesita tener, para cada par de imágenes del dataset, la máscara de cambio original y la máscara que se ha obtenido del método elegido, de estas maneras podrán ser comparadas.

A la hora de desarrollar las métricas se encontrarán elementos comunes. Estas, al trabajar a nivel de píxel para comprobar si se ha realizado correctamente una clasificación binaria (para el ámbito de este trabajo se trabaja con las categorías de “cambio” y “no cambio”), deben de aplicar algún tipo de medida sobre estos, encontrando pues valores:

- **Verdaderos Positivos o VP.** Son aquellos píxeles que han sido marcados en una categoría y realmente pertenecen a ella.
- **Verdaderos Negativos o VN.** Al igual que antes, solo que referido a que han sido correctamente señalados en la categoría contraria.
- **Falsos positivos o FP.** Aquellos píxeles que han sido marcados en una categoría a la que realmente no pertenecen.
- **Falsos negativos o FN.** Como antes, es el mismo concepto pero aplicado sobre la otra categoría.

Para aplicar estas medidas se necesita algún tipo de herramienta que permita compararla, para ello en muchas ocasiones se tendrá que recurrir a **matrices de confu-**

sión, donde cada columna indica el número de predicciones de cada clase, mientras que la fila representa las instancias de una clase.

Con esta representación se consigue visualizar con qué nivel una clase es confundida con otra. Se puede encontrar un ejemplo de la construcción de esta en la Figura 4.1.

		Estimado por el modelo	
		Clase 1	Clase 2
Realidad	Clase 1	VP	FN
	Clase 2	FP	VN

Figura 4.1: Construcción matriz de confusión.

4.1. Descripción de las métricas

A continuación se describen diferentes métricas que se puede aplicar a los resultados de los diferentes algoritmos.

4.1.1. Precisión y exhaustividad

Es una de las métricas más sencillas y de las más usadas a la hora de comparar los diferentes resultados de los modelos.

- **Precisión:** indica la proporción de positivos que fueron identificados correctamente, el porcentaje de que una predicción sea correcta. Es recomendable usarlo si se quieren minimizar los falsos positivos.

$$Precision = \frac{VP}{VP + FP} \quad (4.1)$$

Denotando VP y FP los verdaderos y falsos positivos respectivamente

- **Exhaustividad:** o **Recall**, indica la proporción de positivos reales que fueron correctamente identificados. Recomendable si se quieren minimizar los falsos negativos.

$$Recall = \frac{VP}{VP + FN} \quad (4.2)$$

Idem al anterior.

Ambos valores están muy ligados, tanto que si se intenta mejorar la precisión, disminuirá recall y viceversa.

4.1.2. F-Measure

También conocido como **F-score**, utiliza las dos métricas anteriores. Como ya se ha dicho antes, la mejora de una de estas influye directamente en la otra, por lo que F-score ayuda a trabajarlas mediante un único valor. Esta se calcula mediante la media armónica de la precisión y la exhaustividad:

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} = 2 * \frac{precision \cdot recall}{precision + recall} = \frac{VP}{VP + \frac{1}{2}(FP + FN)} \quad (4.3)$$

También existe otra versión de esta en la que se le da mayor importancia a una de las métricas, F_β .

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \quad (4.4)$$

En la función anterior, si se quiere dar un mayor peso al Recall, se usará un valor de 2, mientras que si se le quiere dar una menor importancia, será de 0.5.

Esta métrica surge para resolver el problema de averiguar qué modelo es mejor si los valores de precisión y exhaustividad no son los mismos. No se puede saber exactamente si un modelo con mucha precisión pero con poca exhaustividad es mejor que otro con menos precisión y mayor exhaustividad, por ello esta métrica ayuda a comparar estos modelos, siempre y cuando se quiera tener en cuenta ambas. Pueden darse casos en los que con la Precisión o Recall se tenga más que suficiente.

Si se observa un ejemplo (ver Figura 4.2), se puede ver como si una de las dos métricas anteriores da un mal resultado, al calcular F-measure, no tendrá mucho impacto

sobre esta.

$$\begin{aligned}\text{Precision} &= VP / (VP + FP) \\ \text{Precision} &= 80 / (80 + 30) \\ \text{Precision} &= 0.73\end{aligned}$$

$$\begin{aligned}\text{Recall} &= VP / (VP + FN) \\ \text{Recall} &= 80 / (80 + 5) \\ \text{Recall} &= 0.94\end{aligned}$$



$$\begin{aligned}\text{F-Measure} &= (2 * \text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \\ \text{F-Measure} &= (2 * 0.73 * 0.94) / (0.73 + 0.94) \\ \text{F-Measure} &= 0.82\end{aligned}$$

Figura 4.2: Ejemplo de F-Measure.

4.1.3. Curva ROC y área bajo la curva (AUC)

Esta muestra el rendimiento del modelo de clasificación en todos los umbrales de discriminación, es decir, su porcentaje de acierto. Para representar este rendimiento se usa una gráfica que se construye en base al número de verdaderos y falsos positivos.

- **Tasa de verdaderos positivos:**

$$TPR = \frac{VP}{VP + FN} \quad (4.5)$$

Donde VP denota verdaderos positivos y FN falsos negativos

- **Tasa de falsos positivos**

$$FPR = \frac{FP}{FP + VN} \quad (4.6)$$

Donde FP denota falsos positivos y VN verdaderos negativos

Para representar dicha gráfica (Figura 4.3), se situarán en los ejes las tasas indicadas; si la sensibilidad es máxima (se puede observar en la gráfica como se ajusta a la esquina superior izquierda), quiere decir que el modelo hace todas sus clasificaciones correctamente y será más sensible; mientras que $(1 - \text{especificidad})$ sea máxima, el modelo siempre fallará en la clasificación. Se puede observar en dicha imagen, como

si la gráfica pasa por C' se estaría obteniendo un resultado bastante aceptable, mientras que si pasa por C , el valor de $(1 - \text{especificidad})$ será demasiado elevado, por lo que se estará obteniendo un rendimiento pésimo.

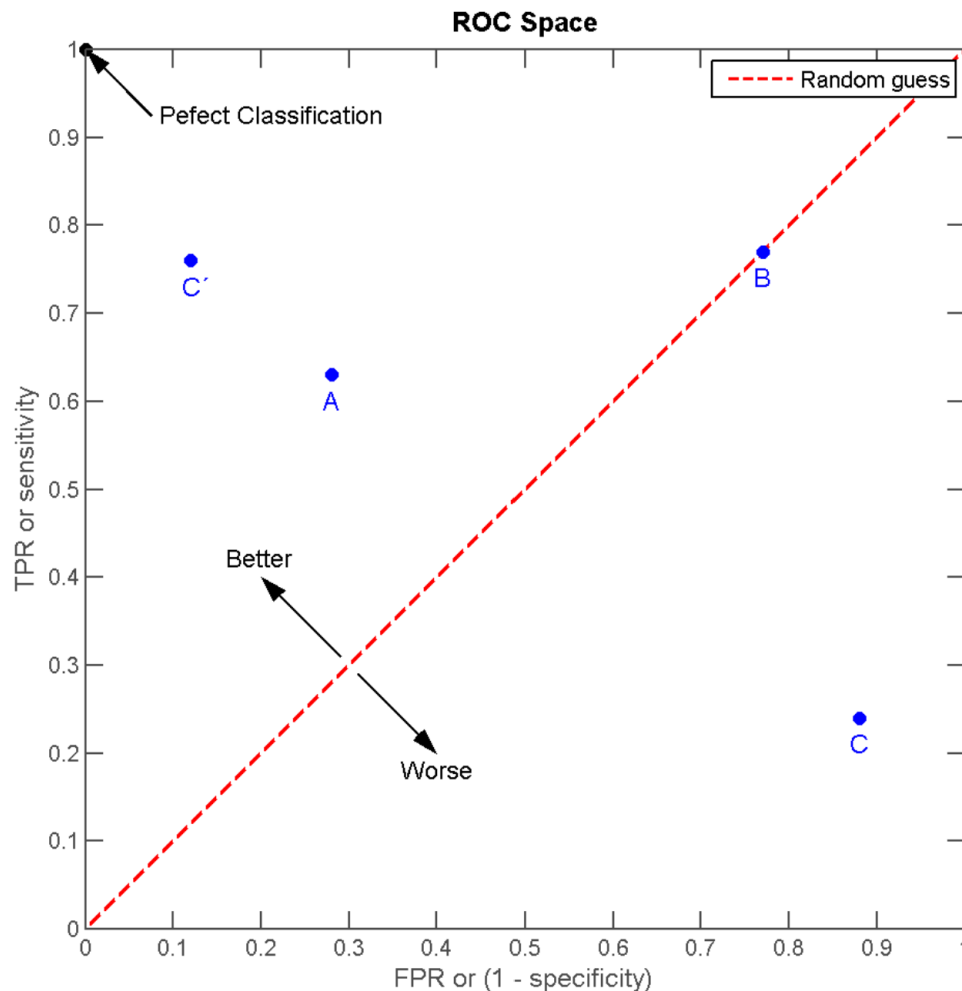


Figura 4.3: Curva ROC. Figura extraída de Wikipedia [wik](#) [2021].

Para desarrollar esta curva se necesitaría ejecutar el modelo variando el valor del umbral numerosas veces. Para ello se usa **AUC (Área bajo la curva)**, que proporciona una medida del rendimiento del algoritmo a través de todos los umbrales de clasificación posibles basándose en la ordenación.

Esta métrica tiene valor entre 0 y 1, siendo 1 cuando el modelo tiene una mayor capacidad de discriminación y 0 cuando menos, es decir, que aquellos valores positivos los marca como negativos y viceversa. Este valor también ayuda a comparar dos curvas ROC fácilmente, siendo la de mayor valor AUC la que mejor resultados presenta. Un ejemplo de esta medida puede ser vista en la Figura 4.4.

El uso de AUC es recomendado por diferentes motivos:

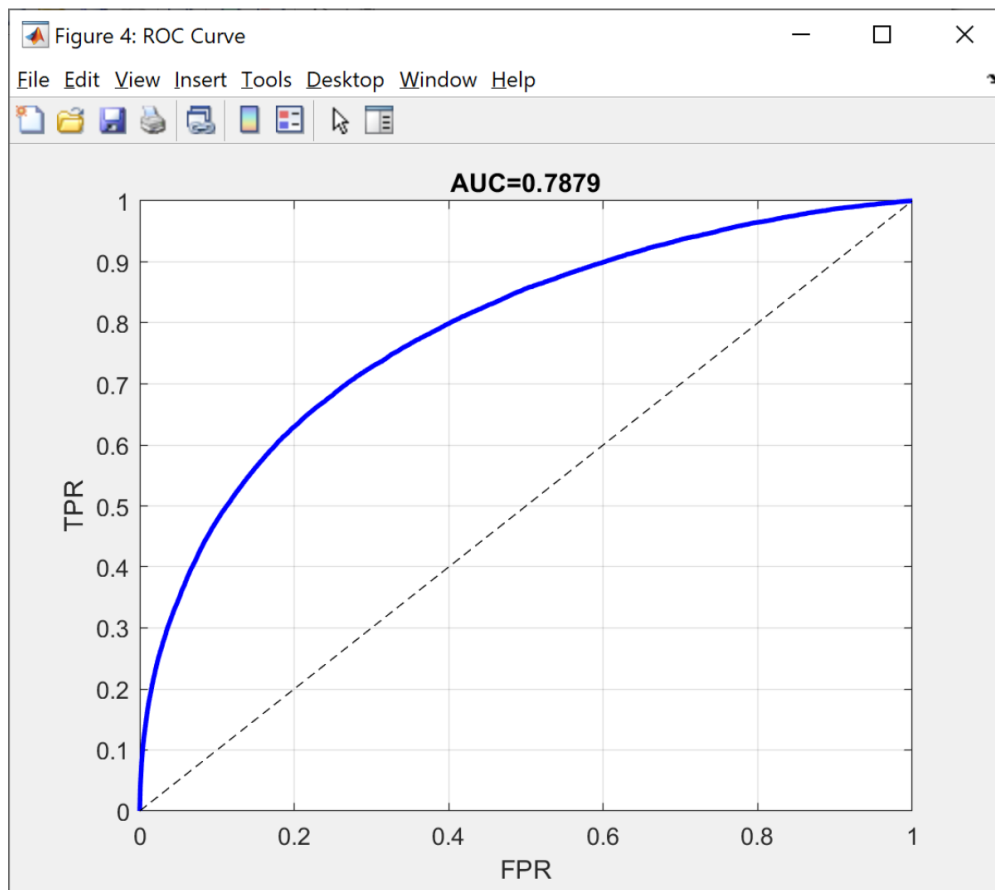


Figura 4.4: Gráfica extraída de una ejecución de **MATLAB**, se puede observar la curva ROC y su correspondiente AUC.

- Es invariante a diferentes escalas, mide cómo de buenas han sido las predicciones, no mira valores absolutos.
- También es invariante ante variaciones en el umbral, hace una medida general de la predicción.

4.1.4. Kappa

Inicialmente se usaba en el campo de la Psicología para medir el “grado de acuerdo” entre dos humanos o calificadores, para después pasar al campo del **Machine Learning** y medir el rendimiento de los modelos.

En este caso no se tienen dos evaluadores, sino un modelo que va a dar predicciones y las máscaras de cambio reales. Se debe medir a qué nivel ambos están de acuerdo en una decisión, en el campo de la clasificación es si ambos modelos coinciden en la predicción de una clase. Se encontrarán casos en los que ambos coincidan,

lo que aumentará este nivel, y casos en los que no. También se considera que esta coincidencia se haya podido dar al azar. Se puede calcular según la ecuación 4.7.

$$k = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \quad (4.7)$$

Donde $Pr(a)$ es el acuerdo observado relativo entre los observadores, $Pr(e)$ es la probabilidad hipotética de acuerdo por azar.

Para poder calcular esta métrica, se debe usar una matriz de confusión, en la que se expresan los diferentes resultados obtenidos. Se encuentra un ejemplo en la Figura 4.5, donde la diagonal indica las veces que el modelo ha hecho una predicción correcta.

		Máscaras		
		Clase 1	Clase 2	Clase 3
Predicciones	Clase 1	4	4	2
	Clase 2	3	5	3
	Clase 3	1	4	1

Figura 4.5: En esta tabla se muestra un ejemplo de matriz de confusión.

Gracias a la tabla anterior se puede calcular las probabilidades de que alguna de estas elecciones se haya dado por casualidad, para ello simplemente hay que calcular el producto de la probabilidad de que ambos “candidatos” estén de acuerdo para cada una de las clases como se puede ver en la Fórmula 4.8.

$$Pr(e)_{clase} = P_{c1}(clase) \cdot P_{c2}(clase) \quad (4.8)$$

$$P_{c1} = \frac{\sum fila_{clase}}{n} \quad (4.9)$$

$$P_{c2} = \frac{\sum columna_{clase}}{n} \quad (4.10)$$

Donde $Pr(e)$ denota la probabilidad de casualidad de que coincidan en una clase determinada, $PC1$ y $PC2$ será la probabilidad del candidato 1 y 2 respectivamente de elegir dicha clase, y n el número total de elementos. De esta manera, al hacer la sumatoria de todas las probabilidades, se obtiene la probabilidad de casualidad total,

por lo que ya se podría calcular Kappa.

4.1.5. Overall Accuracy (OA)

También conocida por sus siglas en inglés como **PCC** (*Percentage of correct classification*) En esta métrica, se mira el conjunto total de píxeles que han sido acertados, es decir, todos los que se han clasificado como cambio y no cambio correctamente. Se puede aplicar la ecuación 4.1.5

$$OA = \frac{VP + VN}{VP + VN + FP + FN} \quad (4.11)$$

Donde VP denota Verdaderos Positivos, VN Verdaderos Negativos, FP Falsos Positivos y FN Falsos Negativos.

4.1.6. User Accuracy (UA)

Ayuda a ver desde el punto de vista del modelo cómo de precisa ha sido la clasificación. Normalmente se aplica a problemas de segmentación, pero puede ser aplicada a una clasificación binaria, por lo que el resultado sería el mismo a aplicar la métrica de **Precisión**, tal y como se puede ver en la ecuación 4.1.6

$$UA_{clase} = \frac{VP}{VP + FP} \quad (4.12)$$

Se puede observar cómo esta fórmula se aplicará para cada clase, por lo que para una máscara de cambio sería aplicarlo a las clases de “cambiado” y “no cambiado”, obteniendo la proporción de píxeles que se han acertado con respecto al total en una clase determinada.

4.2. Interpretación de las métricas

- **AUC**: indica en qué porcentaje el modelo va a proporcionar una clasificación correcta.

- **Precisión:** el porcentaje indica el nivel de acierto. Por ejemplo, si resulta en un 0.5, quiere decir que acierta un 50 % de las veces en sus predicciones.
- **Recall:** este indica el porcentaje de aciertos con respecto a todos los valores que realmente son positivos. Por ejemplo, si se obtiene un 0.10 y hay 100 elementos que son positivos, el modelo sólo marcará el 10 % como positivos.
- **F-measure:** ayuda a comparar modelos en el caso de que se quieran usar las dos métricas anteriores, ya que estas por sí solas pueden resultar bastante confusas.
- **Kappa:** en este caso las puntuaciones van de 1 a -1, siendo 1 un nivel perfecto de acuerdo entre ambos candidatos, por lo que el modelo habría predicho a la perfección.
- **OA:** se obtiene la probabilidad que tiene un individuo, en este caso el píxel, de ser clasificado correctamente.
- **UA:** se muestra el porcentaje de áreas que fueron identificadas correctamente como una clase en concreto. Como ya se ha mencionado antes, en este caso sería igual que aplicar Precisión.

4.3. Implementación de las métricas

Para aplicar estas métricas sobre el conjunto de datos del que se dispone, se usarán las ya implementadas por la librería `sklearn.metrics`, donde se puede encontrar la mayoría de ellas. Estas serán almacenadas en un fichero donde se indica la fecha en la que se calculó la métrica, así como el método y dataset al que ha sido aplicado. Gracias a este fichero, se podrá acceder a estos datos de una forma más sencilla.

Capítulo 5

DATASETS

Para este trabajo se necesita hacer diferentes experimentos sobre datos, para poder así comparar los resultados obtenidos. Como ya se ha visto en apartados anteriores, hay una gran escasez en cuanto a datasets para la detección de cambios, aunque esto no es realmente así. En realidad hay escasez de datasets que incluyan máscaras de cambio, o por lo menos que incluyan etiquetados todos los cambios que se producen entre ambos pares de imágenes. El etiquetado de datasets es uno de los retos que tiene la **Minería de datos** en este momento y en especial los relacionados con imágenes.

En este apartado se muestra una recopilación de las características de diferentes datasets encontrados para analizar respecto al problema de detección de cambios.

Como se puede observar en la tabla anterior [5.1](#), no todos los datasets tienen una gran cantidad de imágenes, algunos de ellos solo tienen un par. Otro aspecto que se puede comprobar es que la resolución de la mayoría de ellos tampoco es muy grande y que incluyen únicamente bandas RGB.

5.1. Datasets con máscaras de cambio

Tras revisar todos los datasets de la tabla anterior, se han extraído todos aquellos que contenían máscaras de cambio.

Dataset	#Cat.	#Image pairs.	Resolution (m)	#Bands	Image size	GL/IT/SP	Year
SZTAKI AirChange	2	13	1.5	RGB	952×640	X✓X	2009
AICD	2	1,000	0.5	115 bands	800×600	XXX	2011
Taizhou Data	4	1	30	6 bands	400×400	✓✓✓	2014
Kunshan Data	3	1	30	6 bands	800×800	✓✓✓	2014
Cross-sensor Bastrop	2	4	30, 120	7,9	444×300;1534×808	✓✓✓	2015
Mts-WH	9	1	1	NIR, RGB	7,200×6,000	✓✓✓	2017
Yancheng	4	2	30	242 bands	400×145	✓✓✓	2018
GETNET dataset	2	1	30	198	463×241	X✓✓	2018
Urban-rural boundary of Wuhan	20	1	4/30	4/9 bands	960×960	✓✓✓	2018
Hermiston City area, Oregon	5	1	30	242 bands	390×200	✓✓✓	2018
OSCD	2	24	10	13 bands	600×600	✓✓✓	2018
WHU building dataset	2	1	0.2	RGB	32,207×15,354	✓✓✓	2018
Season-varying Dataset	2	16,000	0.03 to 0.1	RGB	256×256	XXX	2018
ABCD	2	16,950	0.4	RGB	128×128;160×160	X✓X	2018
California flood dataset	2	1	5,30	RGB,11	1534×808	✓✓✓	2019
Lopez-Fandino et al.	5	2	20	224	984×740;600×500	X✓✓	2019
HRSCD	6	291	up to 0.8	RGB	10,000×10,000	✓✓✓	2019
xBD	6	11034	0.5	RGB	1024×1,024	✓✓✓	2019
LEVIR-CD	2	637	0.5	RGB	1,024×1,024	XXX	2020
SECOND	30	4,214	0.5 to 3	RGB	512×512	XXX	2020
Google Dataset	2	1067	0.55	RGB	256×256	✓✓X	2020
Zhang et al.	2	4	2,2,4,5,8	NIR,RGB	1,431×1,431;458×559;1,154×740	✓✓✓	2020
Hi-UCD	9	1,293	0.1	RGB	1,024×1,024	--/--/✓	2020
SpaceNet7	--	24	4	RGB	--	✓✓✓	2020
S2MTC	2	1,520	up to 10	13	600×600	✓✓✓	2021

Figura 5.1: Tabla extraída de Long [2021] donde se puede ver algunos de los diferentes datasets existentes para el campo de la Detección de Cambios

5.1.1. SZTAKI

Este conjunto de datos es bastante escaso, cuenta con solo 13 pares de imágenes cuyas máscaras están pintadas a mano. Este dataset está pensado para hacer un benchmark de los diferentes métodos. Se puede encontrar la tabla resumen en 5.1

nº Pares	Resolución (m)	Bandas	Tamaño
13	1,5	3	952x640

Tabla. 5.1: Tabla resumen de las características de las imágenes encontradas en el dataset de SZTAKI.

Los autores informan de que los cambios que se tuvieron en cuenta a la hora de desarrollar las máscaras fueron:

- Nuevas construcciones
- Cambios en edificios
- Plantación de grandes grupos de árboles
- Tierra arada
- Preparación de terreno para construir

En este se pueden encontrar máscaras que muestren cambios del terreno y edificaciones, lo cual resulta bastante útil, aún así la cantidad de pares es bastante escasa.



Figura 5.2: Ejemplo de los diferentes pares con sus máscaras correspondientes que se puede encontrar en este dataset. Extraído de [Benedek \[2013\]](#)

5.1.2. AICD

Contiene 1000 pares de imágenes generadas por ordenador, simulando diferentes construcciones y terrenos como si de una escena de un videojuego se tratase, más concretamente "Virtual Battle Station 2", del cual gracias a su renderizador se obtienen estas imágenes. Contiene imágenes de 800x600 desde 5 puntos de vista para una misma escena, además de una versión con y sin sombra. Se puede observar la tabla resumen [5.2](#).

nº Pares	Resolución (m)	Bandas	Tamaño
1000	0,5	3	800x600

Tabla. 5.2: Tabla resumen de las características de las imágenes encontradas en el dataset de AICD

Para cada par de imágenes se encuentra una máscara con las diferentes edificaciones que aparecen, así como el cambio en la vegetación, centrándose más bien en los árboles, ya que el terreno no cambia.



Figura 5.3: Ejemplo de uno de los pares que se pueden encontrar, extraído de Bourdis [2020]

Aunque en un principio este dataset puede parecer útil para comprobar el funcionamiento de los algoritmos desde diferentes ángulos (y no solo vistos desde vista de satélite), también para observar su comportamiento frente a sombras; sin embargo, no es muy útil ya que no se producen cambios en el terreno, solo se obtienen máscaras para los cambios en las edificaciones.

5.1.3. OSCD

Este dataset contiene 24 pares de imágenes extraídas de los satélites **Sentinel-2**, por lo que se encuentran imágenes de diferentes partes del mundo tales como Europa, Estados Unidos, Brasil, Asia, etc. Las imágenes tienen diferentes resoluciones espaciales y tamaño, así como más bandas, tal y como se puede observar en la tabla 5.3.

nº Pares	Resolución (m)	Bandas	Tamaño
24	10	13	variado

Tabla. 5.3: Resumen de las especificaciones de este dataset.

De nuevo las máscaras que ofrecen los autores se centran más en los cambios urbanos, los cambios producidos por el hombre como construcciones o nuevas carreteras como se puede observar en la Figura 5.4. Se puede ver cómo, a pesar de ser imágenes de gran resolución espacial, se marcan multitud de cambios, siendo algunos de ellos diminutos y difíciles de ver a simple vista

Aunque los detalles que proporcionan los autores son numerosos, se pueden encontrar fallos tales como zonas construidas, cambios en el terreno o en la vegetación y que no se encuentran marcados. En la Figura 5.5 se puede ver un ejemplo de la problemática comentada, hay ciertos cambios evidentes que no son marcados, por ejemplo en la parte superior derecha.



Figura 5.4: Uno de los pares incluidos en este dataset. Extraído de [Daudt \[2018\]](#)



Figura 5.5: Problemática encontrada en las máscaras.

A pesar de estos problemas, este dataset será utilizado para este estudio debido a la variabilidad de los diferentes escenarios y la precisión de las máscaras.

5.1.4. HRSCD

Este dataset está compuesto por 291 imágenes tomadas en 2006 y 2012 procedentes del programa **Copernicus**, como se podrá observar en la tabla 5.4. Cuenta con máscaras de cambio y máscaras de segmentación, las cuales vienen marcadas píxel por píxel con una etiqueta numérica.

nº Pares	Resolución (m)	Bandas	Tamaño
291	hasta 0,8	3	10000x10000

Tabla. 5.4: Resumen de las especificaciones de este dataset.

En el caso de las máscaras de cambio se debe destacar que de nuevo, lo que se vuelve a indicar son modificaciones en edificaciones. Si se observa la Figura 5.6 se puede observar cómo los cambios marcados en la mayoría de ellas son muy escasos

a pesar de que se pueden ver numerosas variaciones en la vegetación, diferentes cultivos, y que no son marcados en la máscara.



Figura 5.6: Ejemplo de los pares en alta resolución. [Daudt \[2021\]](#)

Una ventaja que se encuentra con este dataset, es que incluye máscaras de segmentación y que vienen señaladas mediante un código numérico:

- 0: No hay información
- 1: Superficies artificiales
- 2: Áreas agrícolas
- 3: Bosques
- 4: Humedales
- 5: Agua

Si se observan estas máscaras de segmentación (ver [5.7](#)) se puede comprobar que no están del todo completas y que se encuentran numerosas zonas sin segmentar.

Además se encuentra otro problema al descargar esta base de datos, las máscaras vienen con estos valores asignados a cada píxel (es decir, del 0 al 5), por lo que se debe de procesar para poder visualizarlas correctamente. El ejemplo más sencillo es con la máscara de cambio, si el valor que hay en el píxel es 1, significa que hay un cambio, por lo que se debe modificar el valor de dicho píxel para que se marque como blanco.

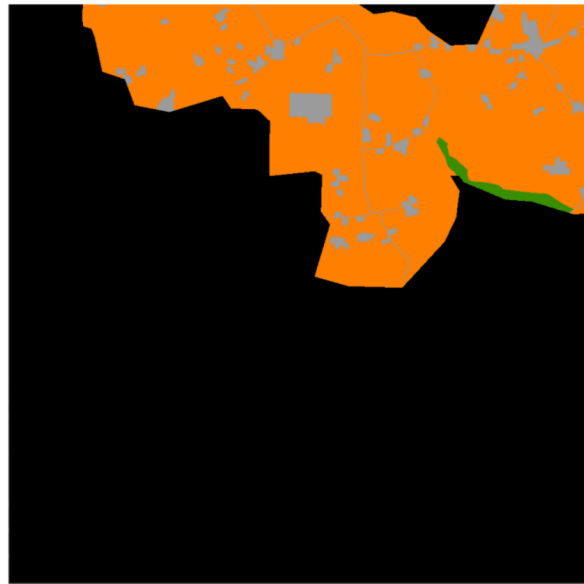


Figura 5.7: Ejemplo máscara de segmentación.

5.1.5. xBD

Conformado por pares de imágenes con el antes y el después de múltiples desastres naturales. Se pueden encontrar instantáneas tras la erupción de un volcán, inundaciones, incendios, etc. Cada imagen está nombrada con el desastre natural que la produjo. En la Figura 5.8 se puede ver la fuente, los diferentes desastres de donde se extrajeron estos pares:

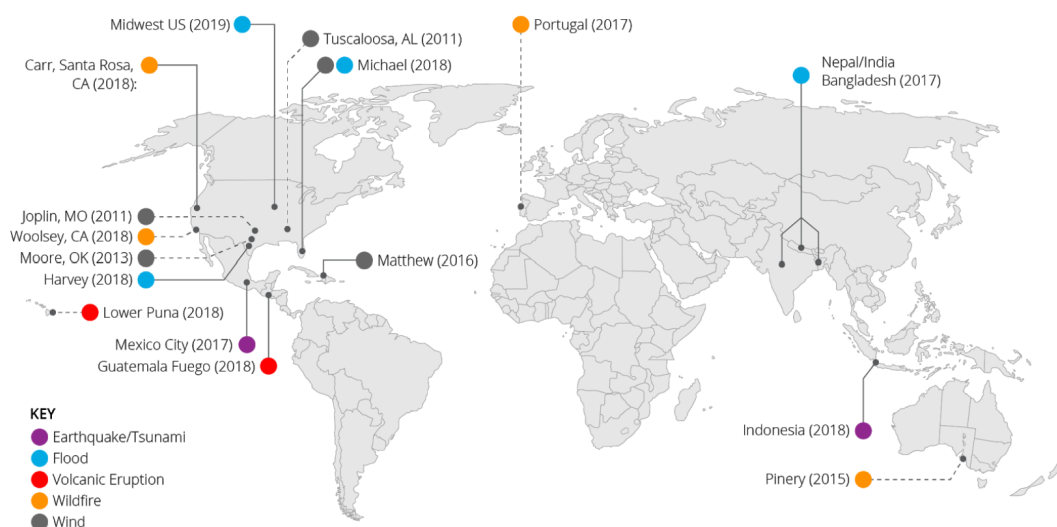


Figura 5.8: Imagen extraída de xVi [2021] donde se pueden apreciar las localizaciones y el tipo de desastre recogido por xBD.

Se encuentran 11034 pares de imágenes RGB proporcionadas por los satélites **Maxar**, tal y como se puede observar en la tabla 5.5.

nº Pares	Resolución (m)	Bandas	Tamaño
11034	0,5	3	1024x1024

Tabla. 5.5: Resumen de las características de las imágenes encontradas en el dataset.

Las máscaras de cambio muestran el nivel de daño producido a las diferentes estructuras:

- **Verde:** daño mínimo, los edificios están parcialmente quemados, levemente inundados, techo hundido o estructura levemente dañada.
- **Naranja:** daño intermedio, techo o paredes parcialmente destruidos, rodeados de agua o barro.
- **Rojo:** daño crítico, completamente destruido.
- **Sin daño.**

A pesar de que el dataset es perfecto para detectar cambios en el terreno, los autores sólo proporcionan información sobre el nivel de daño producido en las diferentes edificaciones, como se puede ver en la Figura 5.9, donde se puede apreciar la desaparición de vegetación en la parte izquierda debido a una erupción volcánica; sin embargo, solo se muestra cómo las diferentes construcciones no se han visto alteradas.



Figura 5.9: Ejemplo de un par extraído del conjunto de xBD. Imágenes tomadas de [xVi \[2021\]](#)

5.1.6. LEVIR-CD

Un extenso dataset usado para realizar *Benchmark* sobre los diferentes métodos aplicados a la Detección de cambios. Los pares de imágenes están extraídos de **Google Earth** con una separación temporal de entre 5 y 14 años. Ver la tabla 5.6 para más detalles.

nº Pares	Resolución (m)	Bandas	Tamaño
637	0,5	3	1024x1024

Tabla. 5.6: Características de las imágenes del dataset de LEVIR.

Las máscaras de cambio están realizadas a mano por expertos, los cuales indican en la máscara binaria los cambios en las construcciones, tanto nuevas como destruidas.

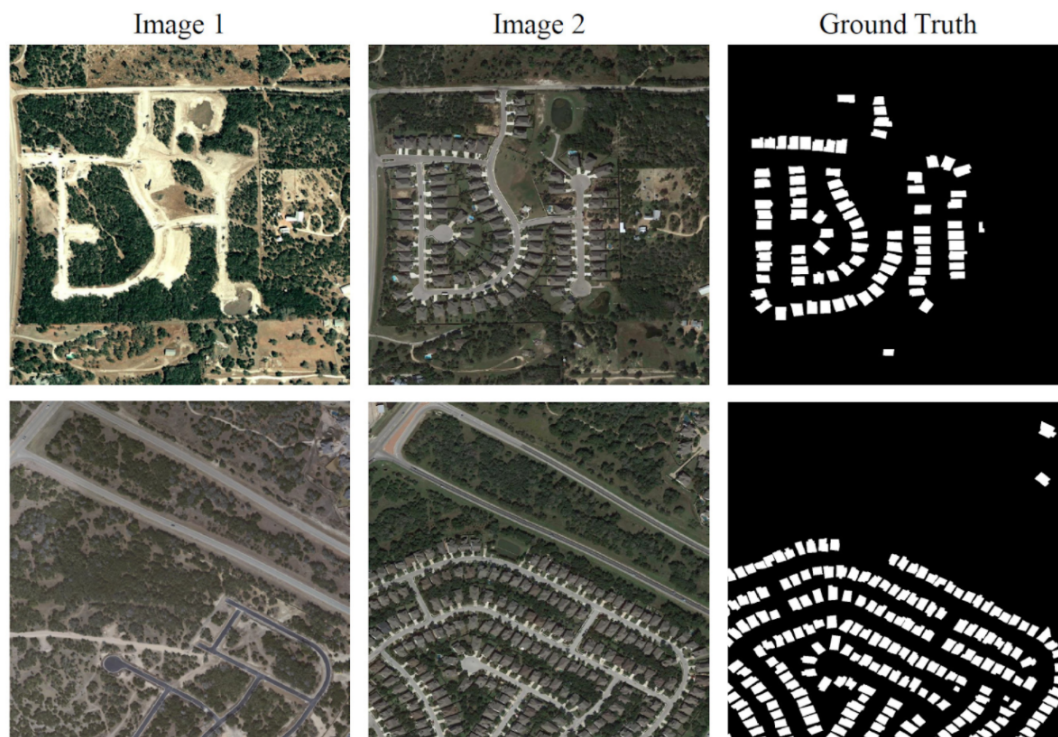


Figura 5.10: Extraída de [Chen and Shi \[2020\]](#), se pueden encontrar ejemplos de los pares con sus correspondientes máscaras.

Como se ha mencionado anteriormente, este dataset suele ser utilizado para estudiar la precisión de diferentes métodos de detección de cambios, por lo que será utilizado en el estudio de los diferentes algoritmos y redes seleccionados. La elección de este también se debe a su gran resolución espacial, ya que se encuentran imágenes de gran calidad, además de la gran cantidad de cambios que se encuentran tras la construcción y/o destrucción de edificaciones.

5.1.7. Hi-UCD

De este dataset solo se cuenta con la información descrita en la tabla 5.7. Como se puede ver, se encuentran imágenes de una gran resolución.

nº Pares	Resolución (m)	Bandas	Tamaño
1293	0,1	3	1024x1024

Tabla. 5.7: Tabla resumen con las especificaciones de Hi-UCD.

De este solo se ha encontrado un artículo que describe con más detalles este dataset, por lo que puede que no se haya publicado aún. En este mismo documento se encuentra un ejemplo de lo que contiene (ver Figura 5.11).

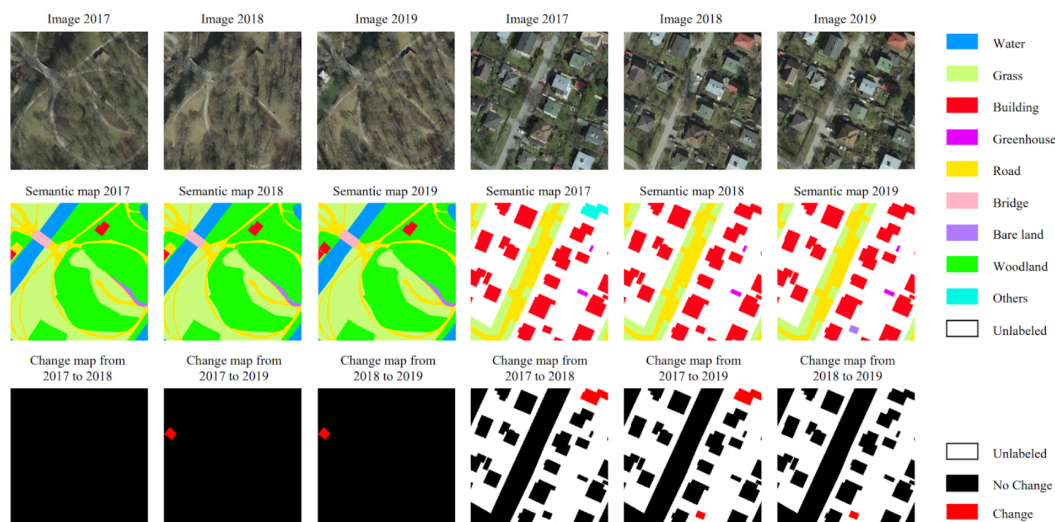


Figura 5.11: Extraída de Tian [2020], muestra las máscaras de cambio y de segmentación junto con el código de color para interpretarlas.

De nuevo las máscaras no muestran cambios sobre el terreno o la vegetación para volver a centrarse en las modificaciones urbanas.

5.1.8. SpaceNet7

Este dataset es realmente un *challenge* que consiste en desarrollar un modelo capaz de analizar imágenes de satélite para predecir el crecimiento de las zonas urbanas tales como grandes ciudades. En la tabla 5.8 se encuentra un resumen de las especificaciones de estas imágenes.

nº Pares	Resolución (m)	Bandas	Tamaño
24x12	4	3	variado

Tabla. 5.8: Resumen de las características de las instantáneas de SpaceNet.

Los autores proporcionan imágenes mensuales a lo largo de 24 zonas diferentes durante 2 años, así como una “máscara” para cada imagen mensual donde se pintan

con formas geométricas la localización de los diferentes edificios. Aprovechando entonces que se cuenta con 24 “máscaras” con la localización de los edificios de una zona, se puede calcular la máscara de cambio entre dos meses restando las propias máscaras de estas. Es decir, se pueden calcular el cambio producido entre el mes 1 y el 2, o el 2-4, 6-7... etc. Produciendo entonces numerosas máscaras de cambio.

Se puede encontrar un ejemplo de estos pares y sus correspondientes máscaras en 5.12, donde también se puede apreciar la máscara obtenida tras restar las propuestas por el autor para cada imagen.

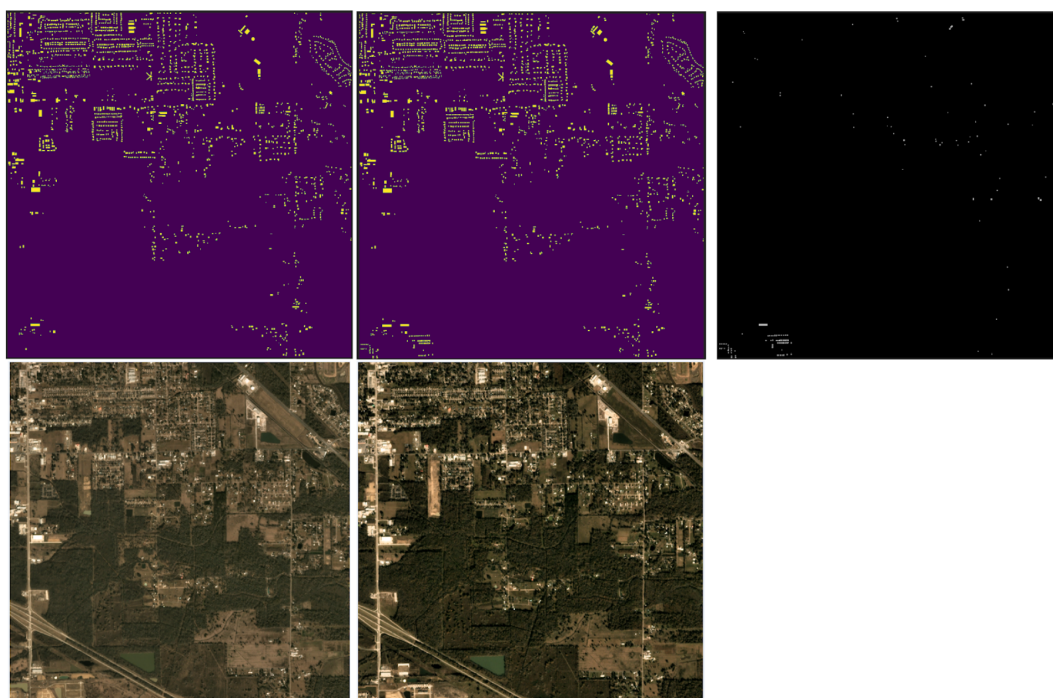


Figura 5.12: Ejemplo de un par de imágenes de meses diferentes, en la parte superior se encuentran las máscaras con las edificaciones correspondientes a las imágenes inferiores. A la derecha se puede ver la máscara generada al restar las anteriores. Imágenes extraídas de Spa [2021]

El principal problema a la hora de tratar estas máscaras que proporcionan los autores, es que están en formato .geojson, por lo que deben de ser procesadas y almacenadas antes de poder trabajar con ellas, o al menos visualizarlas. También se debe mencionar que algunas imágenes están “corregidas” mediante un borrón negro para evitar que la aparición de nubes transmita una información equivocada al modelo o algoritmo.

Se vuelve a la problemática que se tenía con estos datasets, solo se pueden obtener cambios producidos en construcciones.

5.2. Problemas con las máscaras

Tras analizar uno por uno estos datasets y otros encontrados por Internet, la mayoría de ellos no tienen máscaras asociadas a los diferentes pares o si las tienen no muestran realmente todos los cambios producidos en la imagen, sino que suelen mostrar las modificaciones realizadas por el hombre tales como edificaciones y carreteras, como se ha podido comprobar en la Sección 5.1. Si se produce algún cambio en el terreno, bien por cultivos o por el crecimiento de la vegetación (por ejemplo de un bosque), no se ven marcados en ninguno de estos. Esto supone un problema a la hora de realizar las diferentes experimentaciones, ya que el algoritmo puede detectar cambios, que aunque no parecen marcados en las máscaras, si es verdad que se producen. A la hora de calcular las diferentes métricas para comprobar su funcionamiento, no se estaría mostrando realmente cuál es la precisión de estos métodos, sino la que más se ajusta a las máscaras proporcionadas por los propios autores.

5.3. Posibles soluciones

Ante estas dificultades encontradas, se proponen diferentes soluciones descritas a continuación:

5.3.1. Máscara de segmentación

Algunos de los datasets descritos anteriormente cuenta con máscaras de segmentación que marcan a nivel de píxel a qué categoría puede pertenecer. A partir de estas máscaras se puede obtener una de cambio. Para ello se procede a comparar píxel por píxel para comprobar si se ha realizado un cambio en la clase de este, en caso de ser así se marcaría como cambio.

El problema que se encuentra es que las categorías son limitadas, pudiendo encontrar categoría de vegetación en general cuando en realidad lo deseable sería que estuviese dividido en campos de cultivo, bosques, etc.

Este método podría intentar aplicarse sobre los datasets de **HRSCD** (ver apartado 5.1.4) y **Hi-UCD** (ver apartado 5.1.7). Pero como se ha mencionado antes, las categorías son escasas y tras realizar diferentes pruebas no se ha obtenido ninguna máscara de cambio diferente a la que ya se tenía originalmente ya que las categorías de los

terrenos no han variado. Por este motivo se ha decidido descartar esta solución.

5.3.2. Dividir las imágenes

Como se puede ver en la tabla 5.1 la mayoría de los datasets son escasos o solo cuentan con un par de imágenes. Si estas imágenes tienen una buena resolución se pueden dividir en otras más pequeñas, pudiendo aumentar el tamaño del dataset considerablemente.

Este caso en particular se puede aplicar al dataset de HRSCD (apartado 5.1.4) tal y como se muestra en siguiente figura 5.13. Si se amplía en una de las zonas donde se produce un cambio, se puede ver como dicha zona es candidata para transformarla en un nuevo par de imágenes.

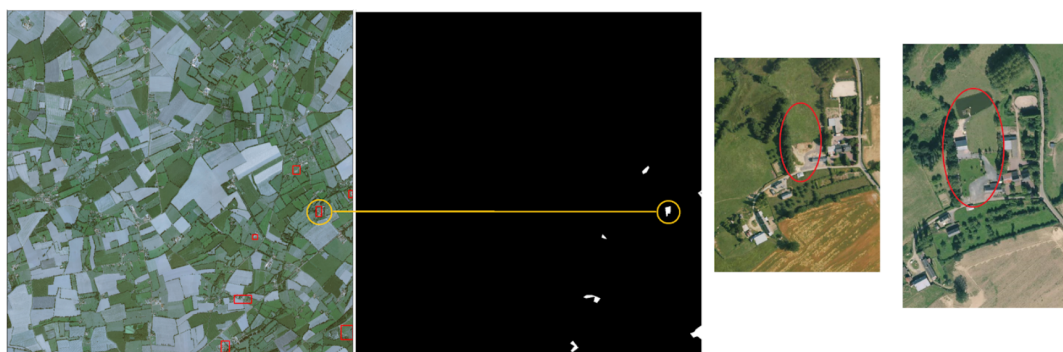


Figura 5.13: Ejemplo de recortado de imágenes.

Como se puede observar, las imágenes son de gran resolución (10000x10000), por lo que puede ser una buena idea recortar las partes donde se encuentran unos cuantos cambios y así obtener aún más pares de imágenes.

Para el caso concreto de este trabajo no se ha aplicado esta solución ya que los datasets escogidos no contaban con esta resolución espacial.

5.3.3. Data augmentation

Esta técnica es una de las más utilizadas actualmente para aumentar la cantidad de datos en un dataset para **Deep Learning**. Consiste en alterar los datos disponibles, generando más imágenes para el dataset. Estas transformaciones son muy variadas pudiendo ser rotaciones, invertir las imágenes, añadir ruido, distorsionar, etc. Se puede

encontrar una extensa explicación para la aplicación de estas sobre un dataset en la documentación de **TensorFlow** ([ten](#) [2021]).

Se puede observar en la Figura 5.14 unos ejemplos tras aplicar esta herramienta ofrecida por **Keras**. De esta manera se consiguen nuevas imágenes sintéticas a partir de otras originales, pero manteniendo la esencia de las mismas.

Con estos nuevos datos sintéticos se puede enseñar a la red neuronal a como comportarse en caso de encontrar imágenes rotadas, con ruido, o distorsionadas. Por lo que se consiguen modelos con una mayor resistencia y que siga funcionando aunque los datos cambien, evitando así el sobreaprendizaje de la misma.

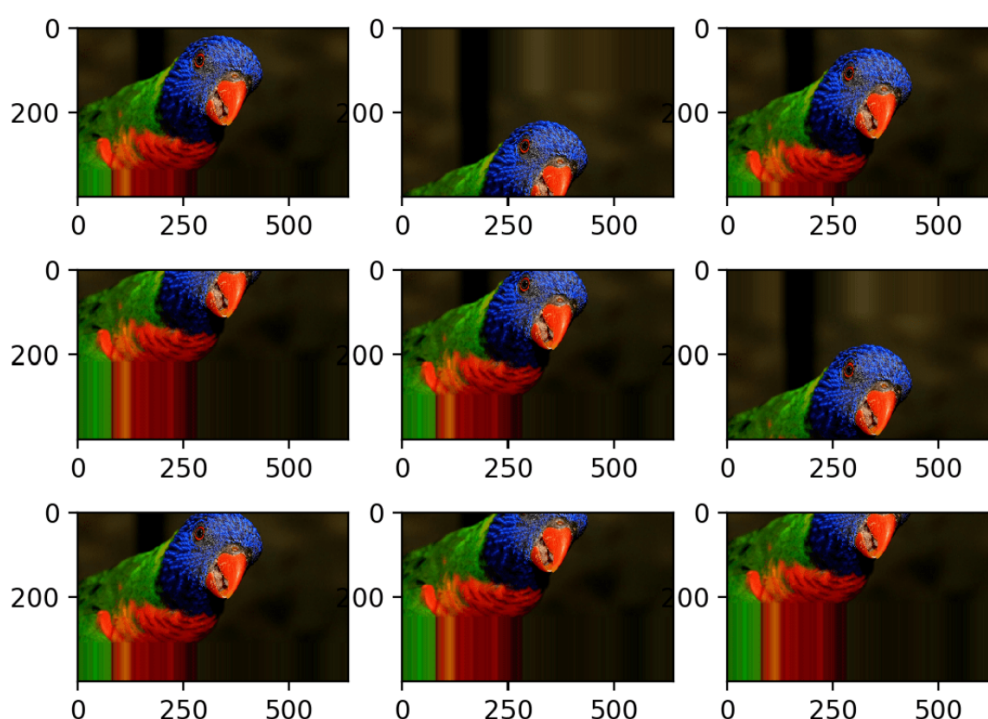


Figura 5.14: Resultado de aplicar una utilidad de Keras para *Data Augmentation*. Figura extraída de [Brownlee](#) [2019].

5.3.4. Crear un dataset sintético

Una de las soluciones más eficaces podría ser desarrollar un dataset propio. Se puede proceder de diferentes formas, pero el objetivo siempre debe ser el mismo, tener un dataset con máscaras de cambio completas, las cuales muestran todos los cambios que se producen realmente entre dos imágenes y que estos no sean solo edificios o modificaciones realizadas por el ser humano.

Como ya se ha mencionado anteriormente, para realizar este trabajo es necesario tener un dataset que permita ver realmente cual es la precisión de los diferentes métodos empleados, pero que con los conjuntos de datos existentes, es un poco difícil ya que estos marcan falsos positivos sobre las máscaras, pero que si se mira detenidamente el par de instantáneas se puede observar como si se ha producido algún tipo de cambio. Para intentar solucionar este problema, se desarrolló un dataset sintético.

Para el desarrollo de este dataset se partió de otro existente, en este caso en concreto de [Mukherjee \[2020\]](#), un dataset diseñado para un *challenge*. Este consiste en desarrollar un algoritmo para la clasificación de diferentes edificaciones, campos e instalaciones en imágenes satelitales. Contiene numerosas imágenes de diferentes zonas, con varias instantáneas de la misma en diferentes periodos. Los autores proporcionan estas imágenes separadas en 4 u 8 bandas o bien se puede adquirir la versión RGB, así como separadas por diferentes categorías. Para el desarrollo de este trabajo se trabajará con la categoría de **CROP_FIELD**, donde se muestran imágenes de diferentes entornos agrícolas, donde también aparecen edificaciones, pero sobretodo aparecen cambios en las cosechas y campos de cultivo.

Como se mencionará unos apartados más adelante (ver apartado [5.4](#) sobre pre-procesamiento), este dataset ha sido procesado con anterioridad ya que las imágenes no estaban alineadas correctamente o tienen problemas de iluminación. Tras una revisión manual de las diferentes instantáneas, se aplica uno de los algoritmos conocidos, de los que se verán posteriormente (más concretamente [6.4 PCA K-Means](#)), partimos del resultado de este para generar los cambios de la imagen sintética.

En la figura [5.15](#) se puede revisar visualmente el proceso de creación de estas imágenes. De esta forma se obtiene un par totalmente nuevo con una de las imágenes anteriores “sin retocar” y la siguiente con píxeles añadidos correspondientes a la máscara generada por el algoritmo. A esta nueva imagen generada se le debe aplicar también otra transformación, ya que al fin y al cabo, se estaría trabajando con la primera y un “parche” de la segunda, lo cual terminan detectando los algoritmos (sobre todo los que trabajan a nivel de píxel).

Para esta segunda transformación se aplicará la corrección Gamma mencionada en el siguiente apartado (ver [5.4.2 Corrección de la iluminación](#)). Para cada una de ellas se usará un valor de gamma aleatorio, con lo que se conseguirá una variación en la iluminación de la segunda escena, evitando así en lo posible el problema mencionado, aunque como se verá, esto no es suficiente para “engañar” a algunos de estos algoritmos. En la Figura [5.16](#) se puede ver el resultado final y la aplicación de la corrección Gamma en la segunda instantánea.

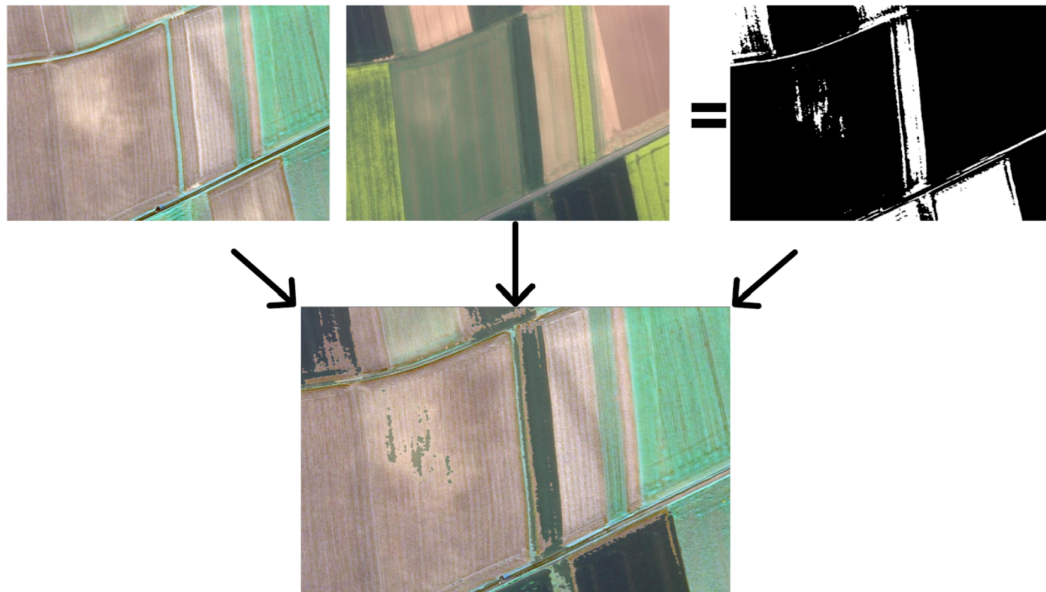


Figura 5.15: En la parte superior se puede ver el par de imágenes original y el resultado obtenido tras aplicar PCA K-means. En la parte inferior se ve el resultado de combinar la primera y la segunda imagen de tal manera que los cambios se correspondan a la máscara generada.

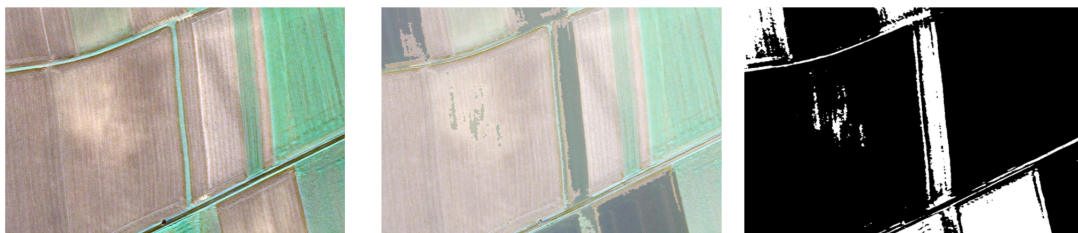


Figura 5.16: Ejemplo del nuevo par generado junto con la mascara de cambio correspondiente.

Un problema que tiene este método es que necesita de una revisión manual ya que algunos resultados del algoritmo pueden no ser deseables, mostrando pocos cambios o demasiado ruido.

5.4. Pre-procesamiento

Una vez se obtienen estos conjuntos de datos se debe de asegurar que ambas imágenes se encuentran en las mismas condiciones, se deben comprobar aspectos importantes tales como la alineación, la iluminación y las condiciones atmosféricas. A simple vista para un humano dos imágenes pueden parecer muy similares aunque se produzca alguno de estos factores, pero para un algoritmo esta tarea resulta más

compleja llegando a confundirlo y producir resultados totalmente erróneos. Por ello es preferible realizar un paso anterior a la aplicación de dichos algoritmos sobre los pares. Existen diferentes correcciones que se pueden aplicar a estas para obtener un resultado deseable y conseguir “igualar” en mayor o menor medida estas instantáneas.

5.4.1. Registro de imágenes

Esta corrección se aplica sobre imágenes que se encuentran desplazadas con respecto a la tomada en el primer instante. Se debe asegurar que ambas se encuentran en el mismo plano y perfectamente alineadas. Si no se da el caso, cualquier algoritmo podría detectar este desplazamiento como un cambio y producir una gran cantidad de ruido en la máscara resultante.

Para realizar este proceso se puede recurrir a una implementación en Python muy sencilla (implementación extraída de [Uberoi \[2019\]](#)). Su funcionamiento consiste en marcar unos puntos clave en la primera imagen para posteriormente buscar estos mismos en la segunda. Una vez marcados y relacionados estos puntos claves en ambas, borra aquellos que considera ruido y los elimina. Finalmente aplica una transformación para alinear la imagen en el mismo plano.

Para esta corrección también se puede usar las coordenadas exactas en la que fue tomada la primera imagen para poder compararlas y realizar las correcciones pertinentes con respecto al desplazamiento de la misma. Aún así es muy recomendable aplicar el alineamiento estudiado en el párrafo anterior tras la corrección sobre las coordenadas.

5.4.2. Corrección de la iluminación

La luz del sol puede incidir sobre la superficie terrestre en diferentes ángulos según al hora del día, pero no solo esto, sino que las condiciones meteorológicas o el material en el que se refleje también influirá en las imágenes. Se debe tener en cuenta que superficies como el suelo descubierto, sin ningún tipo de vegetación, el agua y más elementos pueden producir una saturación en los colores. Se deben conseguir algoritmos que sean capaces de diferenciar estos cambios de escaso interés para centrarse en aquellos relacionados con el entorno, y no con las condiciones en la que estos se encuentran.

También se puede aplicar correcciones en la iluminación de estas escenas para

paliar los problemas que puedan causar. Un ejemplo se puede encontrar en la **Corrección Gamma**, un algoritmo aplicado a los diferentes dispositivos de reproducción de imágenes para adaptar su exposición a lo que vería el ojo humano. Es decir, el ojo humano y un dispositivo de captura (como una cámara) no captan la luz de la misma forma, por lo que se debe de adaptar la salida de estos dispositivos a como lo haría en el ojo. Para ello aplican una fórmula bastante sencilla, parametrizada por el valor de gamma, lo que indica el nivel de exposición que debe de presentar una imagen (ver 5.4.2). Se puede encontrar más información en [Wright \[2010\]](#).

$$g(x, y) = cf(x, y)^\gamma \quad (5.1)$$

$$c = \frac{255}{\max(f(x, y)^\gamma)} \quad (5.2)$$

Donde g denota la imagen de entrada, c una constante para marcar el rango en el que se van a obtener los resultados (normalmente de 0 a 255) y f será el valor del píxel elevado a γ . A menor valor de gamma, más claro será el resultado obtenido.

5.4.3. Corrección atmosférica

Las condiciones atmosféricas no solo afectan a la iluminación de la escena, también pueden aparecer nubes, polvo o niebla. Sin embargo, la atmósfera también produce problemas como la dispersión y absorción de la luz. Sobre las imágenes afectadas se deben aplicar diferentes correcciones radiométricas. Sobre algunos datasets se ha podido comprobar que muchos autores ponen parches sobre las nubes, el cual también se colocaría sobre la otra imagen del par con el fin de evitar calcular cambios en dichas zonas.

Con respecto a la dispersión y absorción existen métodos más sofisticados que se pueden aplicar a imágenes multiespectrales, es decir, que contienen más de 3 bandas (estas se corresponderían con las bandas RGB). Se puede encontrar un ejemplo en el trabajo de [Wiratama et al. \[2018\]](#), donde convierten las intensidades de los píxeles en valores de reflectancia de la superficie y posteriormente a radiación de la atmósfera. Gracias a estos valores se puede realizar una corrección atmosférica sobre dicho par.

5.4.4. Aplicación sobre el dataset sintético

Para el desarrollo del dataset sintético mencionado (ver [5.3.4 Crear un dataset sintético](#)) se realizó una selección manual de diferentes imágenes evitando que se produjeran grandes variaciones entre los diferentes pares; sin embargo, este dataset necesitó ser preprocesado ya que la mayoría de las imágenes aún no se encontraban alineadas o presentaban problemas en la iluminación. Sobre la selección se aplicó el método descrito de *image registration* a diferente nivel de precisión. El problema es que se obtenían imágenes totalmente deformadas, ya que debido al funcionamiento de este algoritmo, se encontraban demasiados puntos de interés o estos no se emparejaban correctamente, lo que resultaba en un cambio de plano incomprensible (ver [5.17](#)). Por lo que finalmente se debía de volver a realizar un proceso manual de clasificación para obtener de esta manera todas las imágenes que se encontraban perfectamente alineadas.

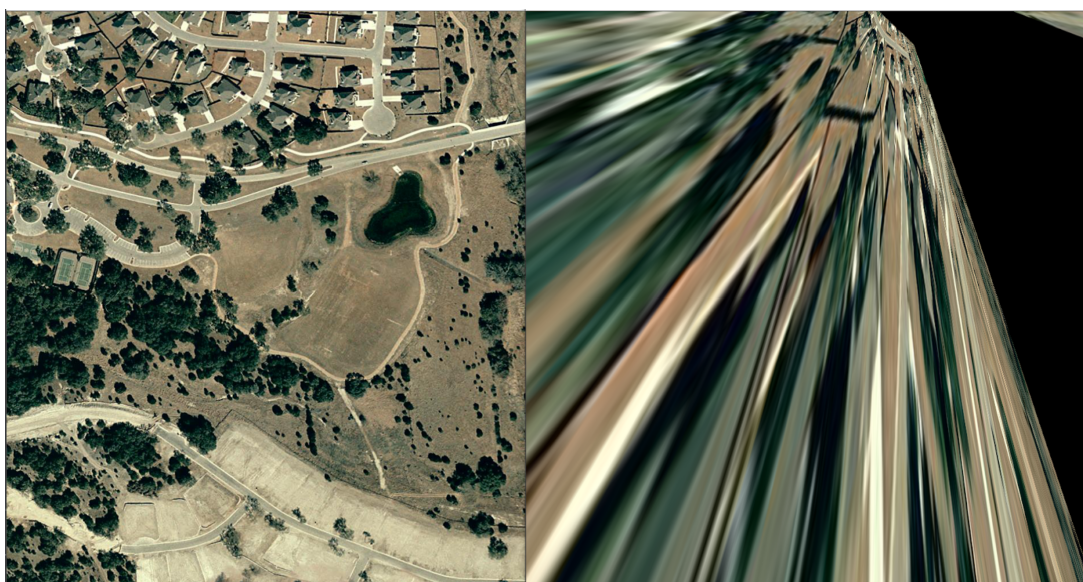


Figura 5.17: Ejemplo de mal funcionamiento de *image registration*.

En los pares en los que se podía observar una gran diferencia con respecto a la iluminación se aplicaba una corrección Gamma, pero de nuevo, los resultados obtenidos dependen del parámetro de gamma y dependiendo de la instantánea se debía aplicar diferentes valores, por lo que también sería necesario un proceso manual de revisión de las diferentes posibilidades. Se puede ver un resultado de estas correcciones en la Figura [5.18](#).



Figura 5.18: Un ejemplo de corrección gamma aplicado al dataset de Crop-Field con un valor de gamma de 0.5.

Capítulo 6

MÉTODOS TRADICIONALES DESARROLLADOS

En este apartado se describirán los diferentes métodos tradicionales seleccionados en este TFG para la aplicación al problema de detección de cambios en imágenes.

En la bibliografía especializada existen múltiples propuestas para la detección de cambios con enfoques tradicionales. De algunos de ellos se puede encontrar implementación con una simple búsqueda en Internet, pero de otros solo se encuentran artículos que describen su funcionamiento y de los cuales se pueden desarrollar una implementación. Se debe tener en cuenta que todos ellos trabajan a nivel de píxel, y a diferencia de los métodos con *Deep Learning*, estos no aprenden y siempre funcionan igual, por lo que se obtienen resultados no muy destacables que contendrán en su mayoría ruido.

De entre todas las propuestas existentes, en este trabajo se han seleccionado las que se describen en las siguientes subsecciones.

6.1. Image Differencing

Este es el método más sencillo que se puede encontrar, ya que consiste en restar píxel por píxel las imágenes en escala de gris de los diferentes instantes ([Minu and Shetty \[2015\]](#)).

Para describir su funcionamiento se recurre a una serie de pasos

6.1.1. Funcionamiento

Paso 1: Obtener las imágenes en escala de grises y la imagen de diferencia

Se debe obtener para cada instantánea la versión en escala de grises. Una vez obtenido esto se debe restar ambas imágenes, píxel por píxel para dar con la imagen de diferencia.

Paso 2: Umbralizar

Una vez calculada esta imagen de diferencia se obtiene la máscara de cambio tras umbralizar la imagen, de esta manera se quedarían los píxeles con mayor intensidad.

La umbralización consiste en transformar una imagen en escala de grises a una nueva en la que solo se tienen dos valores, de esta manera se obtiene una imagen binaria. Para hacer uso de esta técnica se debe aplicar un valor de umbral, a partir del cual los píxeles serán clasificados en una categoría o en otra. El proceso de selección de este valor es bastante delicado, ya que un valor demasiado alto puede descartar cambios muy importantes, mientras que si este valor es bajo se podría obtener demasiado ruido. Para un valor bien escogido, se pueden enfatizar ciertos objetos con respecto al fondo, o en este caso, resaltar los elementos más importantes de la imagen para los cuales se va a estudiar si han sufrido algún tipo de modificación.

Para aplicar esta transformación se recurre a un método proporcionado por **OpenCV**. Para el que se puede encontrar la descripción de su sintaxis a continuación:

```
1 imagenUmbralizada = cv2.threshold(imagenAUmbralizar, umbral, maximoValor,  
2     tipoUmbralizacion)  
3 imagenUmbralizada : imagen resultado tras umbralizar.  
4 imagenAUmbralizar : imagen que va a ser umbralizada.  
5 umbral : valor de umbral a aplicar sobre la imagen.  
6 maximoValor : en caso de la umbralización binaria, valor máximo que se pondrá sobre  
    aquellos valores que superen el umbral  
7 tipoUmbralizacion : existen de diferentes tipos, incluso se pueden concatenar mediante el  
    operador OR
```

Listado 6.1: Método para la aplicación del umbral.

En el caso concreto de este proyecto, el valor de umbral es trivial, ya que el tipo de umbralización utilizado será una combinación entre **binario** y **Otsu**. Mediante la umbralización binaria lo que se indica es que en la imagen resultante únicamente se encontrarán dos valores de intensidad, 0 y el máximo valor que se haya indicado con

el parámetro `maximoValor` (en este caso concreto será 255 para que así los cambios sean marcados como blanco). Otsu a su vez ayuda a calcular el mejor umbral para la imagen.

El funcionamiento del algoritmo de umbralización de **Otsu** puede encontrarse explicado detalladamente en [Murzova and Seth \[2020\]](#), pero se puede resumir en unos pasos. Primero extrae el histograma de la imagen, donde se encuentra la distribución de las intensidades de los diferentes píxeles en una escala del 0 al 255 en caso de que la imagen esté en escala de grises. La idea principal consistirá en dividir este histograma en dos grupos, uno en el que las intensidades de los píxeles será menor a un umbral calculado y otro en el que estas intensidades superen a dicho valor. Para calcular el umbral se puede aplicar la Ecuación 6.1.1, donde se quiere minimizar la suma ponderada de la variación entre dos clases.

$$\sigma_w^2(t) = w_1(t)\sigma_1^2(t) + w_2(t)\sigma_2^2(t) \quad (6.1)$$

Donde w_1, w_2 son las probabilidades de pertenecer a cada clase separadas por el valor t , el umbral. $\sigma_1^2(t), \sigma_2^2(t)$ serían la dispersión de cada clase. Lo que se pretende es minimizar la dispersión entre ambas, con ello lo que consigue es calcular el valor del umbral para obtener la dispersión mínima entre componentes de la misma, pero que esta sea alta para elementos de grupos diferentes.

Por lo que finalmente, la sentencia utilizada a la hora de umbralizar en los diferentes métodos, es la que se presenta a continuación:

```
1 img = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

Listado 6.2: Aplicación del umbral.

6.1.2. Pseudocódigo

```
1 function ImgDiff(imagen1, imagen2){
2     imagen1 = pasarEscalaGrises()
3     imagen2 = pasarEscalaGrises()
4
5     X = absoluto(imagen1 - imagen2)
```



```

6 |
7 |     Y = umbralizar(X)
8 |
9 |     return Y
10| }
11|
12| imagen1 : Imagen en el instante 1
13| imagen2 : Imagen en el instante 2
14| X : Imagen diferencia
15| Y : Mapa de cambio

```

Listado 6.3: Pseudocódigo de Image Differencing.

6.1.3. Implementación

Para hacer pruebas con este método tradicional se recurre a una implementación propia ya que, como se ha podido observar en el pseudocódigo, está es bastante sencilla de desarrollar. A la hora de implementarlo se encontraron dos posibilidades:

- **Img_diff**: restar píxel a píxel y obtener el valor absoluto de dicho resultado.
- **Compare_ssim**: utilizar el método `compare_ssim()` del módulo **Skimage**. Este método aplica la métrica de **SSIM** (*Structural Similarity Index*), que fue creada por [Wang et al. \[2004\]](#) como alternativa al **Error Cuadrático Medio** usado hasta entonces para comparar dos imágenes. Este método se basa exclusivamente en los valores de intensidad de los píxeles, con el se calcula la iluminación, el contraste y la estructura de la imagen para comprobar si realmente dos imágenes son iguales o muy similares. Se puede observar la Ecuación para calcularlo en [6.2](#)

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \quad (6.2)$$

Donde μ , σ y $\sigma(x, y)$ son la media, la desviación estándar y la covarianza cruzada respectivamente del correspondiente píxel x , y .

Esta función ayuda a calcular la similitud estructural de dos imágenes, es decir, ayuda a calcular las diferencias entre dos imágenes similares. Normalmente es usada para comprobar si hay una gran diferencia entre la imagen original y una que ha sido comprimida o a la que se le ha aplicado algún filtro, pero en este caso también ayudará a comprobar las diferencias a nivel de píxel entre dos instantáneas. Como resultado se obtiene una imagen de diferencia así como una puntuación, el nivel de similitud entre estas dos imágenes.

En la Figura 6.1 se puede observar como ambos algoritmos varían en los resultados para el mismo par.

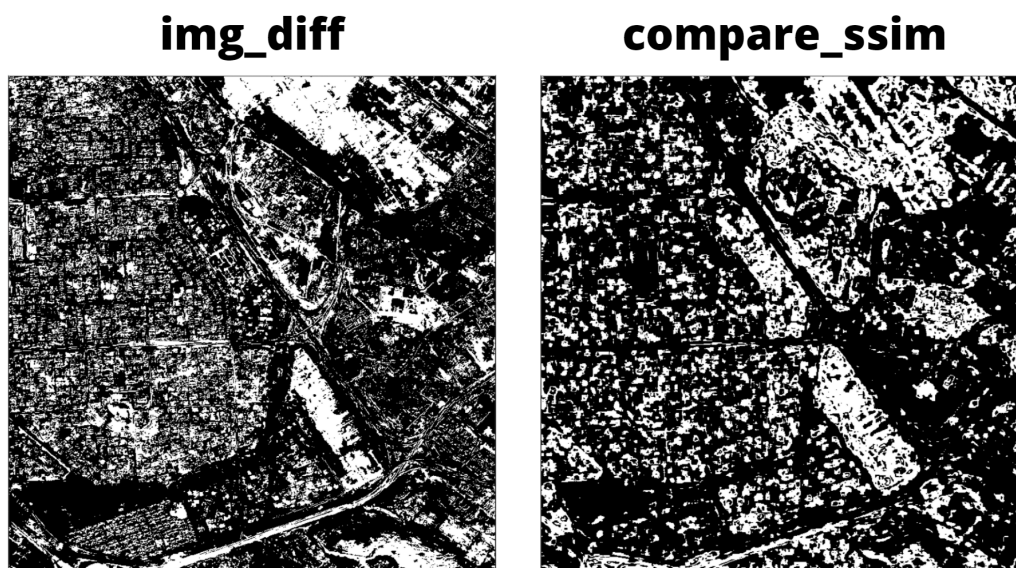


Figura 6.1: Resultado de la imagen de diferencia calculada por ambos algoritmos.

Como se puede deducir, para estas implementaciones se variará ligeramente el pseudocódigo anterior.

```
1 function ImgDiff(imagen1, imagen2){
2     imagen1 = pasarEscalaGrises()
3     imagen2 = pasarEscalaGrises()
4
5     X = calculaImagenDiferencia(imagen1, imagen2)
6
7     Y = calculaContornos(X)
8
9     return Y
10 }
11
12 imagen1 : Imagen en el instante 1
13 imagen2 : Imagen en el instante 2
14 X : Imagen diferencia
15 Y : Mapa de cambio
```

Listado 6.4: Pseudocódigo de *Image Differencing* modificado.

La primera opción se desarrollaría tal y como se puede ver en el pseudocódigo anterior, mientras que en la segunda se sustituiría por la función de **Skimage**. Una vez calculada la imagen de diferencia, se puede ver como se genera mucho ruido (esto se puede apreciar en la Figura 6.1), muchos conjuntos de píxeles muy pequeños y que

realmente no tienen ningún significado ni representan ningún cambio. Para solucionar este problema, se usará el método `calculaContornos()`, otra utilidad que se encuentra en la librería OpenCV que busca y rellena los diferentes contornos, cuya sintaxis se describe en los siguientes Listados 6.5 y 6.6

```
1 cv2.findContours(imagen, contornos, jerarquía, modo, método, offset)
2
3 imagen: en la cual se van a buscar los contornos, en este caso es binaria.
4 contornos: vector donde se van a almacenar los resultados, para cada uno almacena un
   vector de puntos.
5 jerarquía: vector que incluye información sobre la topología de la imagen, en este caso
   no es necesario.
6 modo: como se va a construir la jerarquía.
7 método: de aproximación de los contornos.
8 offset: desplazamiento que se aplica al contorno encontrado, es decir, lo que se desplaza
   el contorno del punto original donde se ha encontrado para marcarlo así como un
   borde alrededor de este punto.
```

Listado 6.5: Métodos para encontrar los contornos.

```
1 cv2.drawContours(imagen, contornos, IndiceContorno, color, grosor, tipoLinea, jerarquía,
   nivelMáximo, offset)
2
3 imagen: sobre la que se van a pintar los contornos.
4 contornos: que se quieren pintar, se obtiene de la función anterior.
5 IndiceContorno: índice del contorno a pintar, si se quieren pintar todos se debe
   establecer un valor negativo.
6 color, grosor y tipo de línea: parámetros para indicar el formato de la línea.
7 jerarquía: también extraída de la función anterior.
8 nivelMáximo: pinta el nivel correspondiente de la jerarquía el cual se indique.
9 offset: de nuevo, se puede añadir un nuevo desplazamiento.
```

Listado 6.6: Método para rellenar los contornos.

Para la implementación de todos estos métodos tradicionales se han usado los parámetros definidos en el Listado 6.7. El modo escogido para encontrar los contornos es **cv2.RETR_EXTERNAL**, que devolverá los bordes externos a los contornos encontrados, por esta razón no se necesita un offset; mientras que como método de aproximación, será utilizado **cv2.CHAIN_APPROX_SIMPLE** que almacena solo los puntos cuando se produce un cambio en la orientación de los mismos, y no un listado con todos ellos, para pintarlo posteriormente solo debe unir los puntos consecutivos, por lo que también supone un ahorro de memoria.

```
1 contours, h = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```
2  
3 img = cv2.drawContours(img, contours, -1, (255, 255, 255), -1)
```

Listado 6.7: Aplicación de los métodos de buscar y rellenar.

Al aplicarlas devolvería estos conjuntos de píxeles totalmente rellenos, pero se obtienen demasiados. Una vez encontrados estos contornos, solo se marcarán 20, los cuales han sido previamente ordenados de mayor a menor atendiendo a que estos tengan una mayor presencia de píxeles. De esta manera se consigue dar más importancia a los cambios más destacables, los más “grandes” y teniendo así una mayor probabilidad de descartar los conjuntos que en realidad son ruido o más “pequeños”, los cuales no marcan ningún tipo de cambio.

Al hacer las pruebas con estos algoritmos no se obtuvieron muy buenos resultados, como se verá más adelante en el apartado [8.1.1 Experimentaciones con Image Differencing](#), así que se consideró aplicar una posible mejora, realizar una umbralización adaptativa por bloques.

Esta umbralización lo que permite es mejorar el método usado por la mayoría de estos algoritmos, que aplican un umbral fijo sin considerar variaciones en la iluminación, la aparición de sombras en determinadas zonas, variaciones de color, etc. Para aplicar este método se recurre de nuevo a una función de **OpenCV**, cuya sintaxis se encuentra en el Listado [6.8](#).

```
1 cv2.adaptativeThreshold(imagenAUmbralizar, imagenDeSalida, maximoValor, método,  
2     tipoUmbralización, tamañoDeBloque, c )  
3 imagenAUmbralizar: imagen que se quiere umbralizar.  
4 imagenSalida: imagen resultante.  
5 maximoValor: valor que se le dará a los píxeles que sobrepasen el umbral.  
6 método: se indica el tipo de algoritmo adaptativo que se quiere utilizar.  
7 tipoUmbralización: al igual que con la umbralización simple.  
8 tamañoDeBloque: de píxeles que serán procesados para calcular el umbral adecuado para el  
9     píxel central.  
c: constante, se aplicará en las diferentes operaciones realizadas por el algoritmo.
```

Listado 6.8: Aplicación del método de umbralización adaptativa.

Como se puede observar, para aplicar este algoritmo, se necesita un tamaño de bloque, ya que se calcula un valor de umbral adecuado para cada píxel en función de su vecindad, es decir, del bloque de píxeles que se ha definido. Por ello este valor debe de ser impar, ya que debe de quedar un píxel central al que asociar el valor calculado. Para obtener este umbral se aplica el método deseado, en este caso se aplicará **cv.ADAPTIVE_THRESH_GAUSSIAN_C**. Este método realiza una suma ponderada

de toda la vecindad, usando un filtro Gaussiano, menos la constante c .

El **filtro Gaussiano** se caracteriza por ser **isotrópico**, si se rota, las posiciones de los coeficientes se mantienen. Además de esto, el centro de este filtro es el que tiene mayor peso, disminuyendo gradualmente a medida que se acerca a los bordes, con esto se consigue darle una mayor ponderación, una mayor importancia a los píxeles centrales. Se puede ver un ejemplo en la Figura 6.2. El filtro utilizado por este algoritmo es calculado por defecto por el programa dependiendo del tamaño de bloque aplicado.

Matriz de filtro 5×5

$$\begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 3 & 5 & 3 & 1 \\ 2 & 5 & 9 & 5 & 2 \\ 1 & 3 & 5 & 3 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix}$$

Figura 6.2: Ejemplo de filtro Gaussiano extraído de [Raziel \[2017\]](#).

El objetivo de aplicar este algoritmo reside en obtener el valor apropiado para cada zona, es decir, si se encuentra una zona muy iluminada y otra con mucha sombra, cada una contará con un umbral adecuado, el cual remarcará los diferentes detalles. Con la umbralización normal utilizada hasta ahora, estas zonas de intensa luminosidad u oscuridad generarían un gran contraste como se puede observar en la Figura 6.3, donde se puede ver como en la zona con sombra se pierde toda la información.

También se puede ver un ejemplo sobre una imagen del dataset de OSCD (ver 6.4), se puede observar el mismo efecto. En la parte superior derecha se puede ver como con la umbralización general se habría perdido mucha información, pero con la adaptativa se recupera, se ve como todos los detalles, las diferentes carreteras quedan marcadas, así como los edificios.

A la hora de aplicar el método se ha utilizado la llamada mostrada en el Listado 6.9.

```
1 img1 = cv2.adaptiveThreshold(img1, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.
    THRESH_BINARY, tamañoBloque, 2)
```

Listado 6.9: Aplicación del método de umbralización adaptativa.

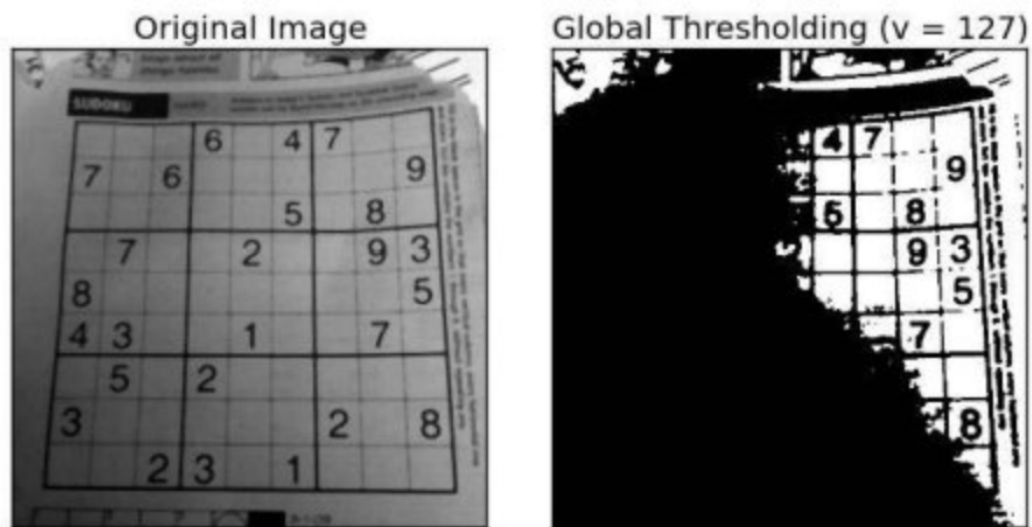


Figura 6.3: Ejemplo de umbralización adaptativa extraída de la documentación de [OpenCV](#) [OpenCV](#) [2021].

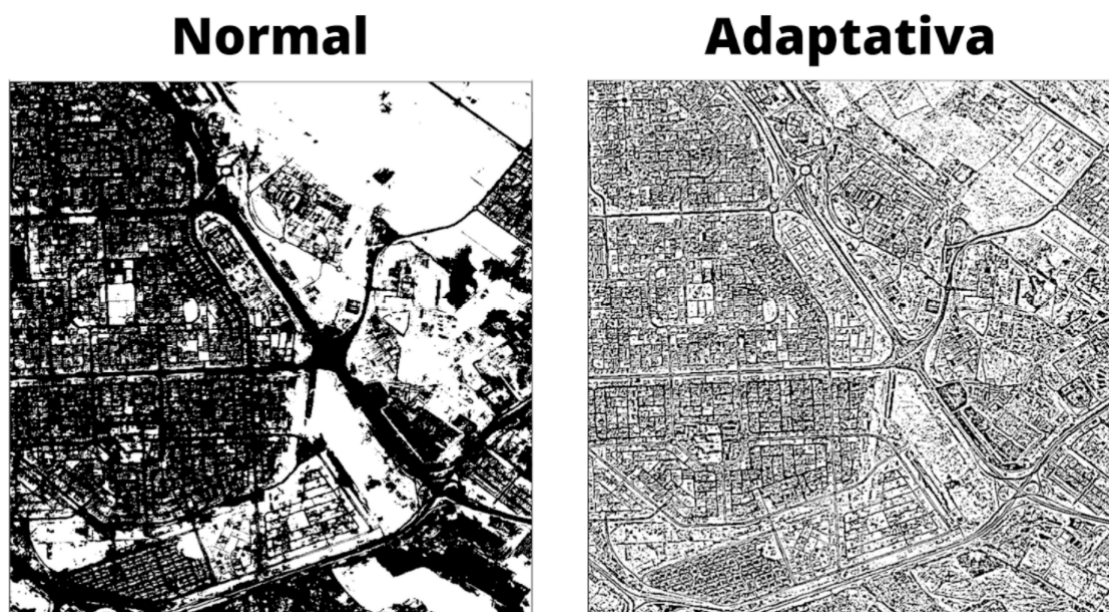


Figura 6.4: Se puede observar las claras diferencias entre ambos métodos de umbralización.

Se ejecutaron diferentes experimentos con bloques de tamaño 11, 31 y 101 para poder comparar los posibles resultados. En la Figura 6.5 se puede ver como la ejecución con diferente tamaño de bloque puede variar los resultados obtenidos por el algoritmo, a mayor tamaño de bloque menos detalles se obtienen.

Para aplicar esta posible mejora, se debe modificar de nuevo el pseudocódigo:



Figura 6.5: Dependiendo del tamaño del bloque, se obtienen diferentes resultados.

```

1 function ImgDiff(imagen1, imagen2){
2     imagen1 = pasarEscalaGrises()
3     imagen2 = pasarEscalaGrises()
4     ____
5     ____imagen1 = umbralizar(imagen1)
6     ____imagen2 = umbralizar(imagen2)
7
8     X = calculaImagenDiferencia(imagen1, imagen2)
9     ____
10    ____Y = calculaContornos(X)
11
12    return Y
13 }
14
15 imagen1 : Imagen en el instante 1
16 imagen2 : Imagen en el instante 2
17 X : Imagen diferencia
18 Y : Mapa de cambio

```

Listado 6.10: Pseudocódigo de *Image Differencing* con Umbralización adaptativa.

En esta ocasión se han añadido las líneas 5 y 6 en el pseudocódigo 6.10 ya que se debe umbralizar las imágenes antes de calcular la imagen de diferencia mediante los métodos descritos.

6.2. Canny

Esta técnica se basa en aplicar Canny ([Valverde-Rebaza \[2007\]](#)) a los pares antes de calcular los cambios. Tras aplicar Canny, se procede al igual que con *Image Differencing*, se resta píxel por píxel para calcular la imagen de cambio. El objetivo de usar

Canny reside en destacar todos los detalles de la imagen, en extraer los bordes de los diferentes elementos para posteriormente poder restarlos. De esta manera si ha aparecido o desaparecido alguno de estos, en teoría también se verá reflejado en la aparición o desaparición de bordes tras aplicar Canny.

6.2.1. Funcionamiento

Paso 1: Aplicar Canny

Esta técnica de pre procesamiento de imágenes consiste en aplicar diferentes transformaciones a la imagen para obtener el contorno de los diferentes elementos localizados en la imagen. Se encuentra una explicación del funcionamiento de Canny en [Valverde-Rebaza \[2007\]](#), de donde se extrae un resumen de los pasos seguidos por el algoritmo:

- Primer paso: **Obtención del gradiente**, es decir, se debe calcular la magnitud y orientación del gradiente. Antes de ello se debe aplicar un filtro gaussiano para suavizar la imagen y así impedir que aparezca ruido.
- Segundo paso: **Supresión no máxima**. En este paso se logra el adelgazamiento del ancho de los bordes, obtenidos en el paso anterior, hasta que estos tengan un grosor de un píxel.
- Tercer paso: **Histéresis de umbral**, basada en dos umbrales, con este proceso se pretende reducir la posibilidad de aparición de contornos falsos, eliminando aquellos que no entren entre estos umbrales.

A la hora de desarrollar este método se ha utilizado la implementación de Canny ofrecida por **OpenCV**, cuya descripción del método se puede hallar en el Listado [6.11](#), aunque para este algoritmo solo se han definido los dos umbrales.

```
1 cv2.Canny(imagen, vectorSalida, umbral1, umbral2, aperturaSobel, gradienteL2)
2
3 imagen: a la que aplicar el método.
4 vectorSalida: mapa donde se muestran los contornos encontrados.
5 umbral1 y umbral2: aplicados en el proceso de histéresis.
6 aperturaSobel: en este caso no se aplica Sobel.
7 gradienteL2: un booleano que indica si para calcular el gradiente se aplica L2 en lugar de L1.
```

Listado 6.11: Sintaxis del método de Canny de OpenCV.

Se aplica este método a ambas imágenes tras pasarlas a escala de grises, obteniendo como resultado lo observado en la Figura 6.6, donde se marca en rojo los cambios producidos entre ambos pares y que se pueden ver a simple vista.

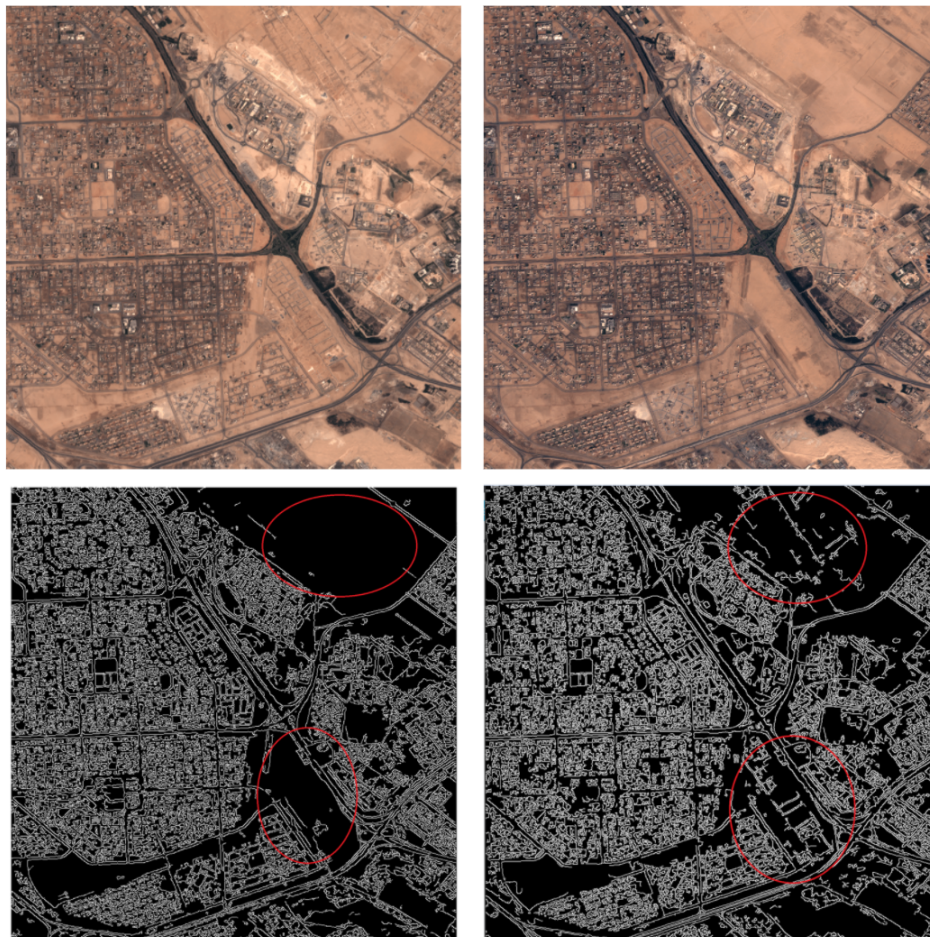


Figura 6.6: Resultado tras aplicar Canny a un par de imágenes de OSCD.

Paso 2: Aplicar un filtro gaussiano y rellenar los contornos

El objetivo de este paso es aplicar un emborronamiento a la imagen, ya que, tras obtener los bordes de los diferentes elementos, algunos de estos quedan abiertos y, como se verá, esto podría suponer un problema para el siguiente paso de rellenar los contornos. Esto último se realiza para evitar generar grandes cantidades de ruido en la imagen diferencia calculada, tal y como se muestra en la Figura 6.7. En ella se puede observar en la parte superior como si no se aplica dicho emborronamiento, muchos de

los contornos detectados por Canny quedan abiertos, por lo que se genera una gran cantidad de ruido. Este se traducirá en aún más ruido a la hora de calcular la imagen de diferencia.

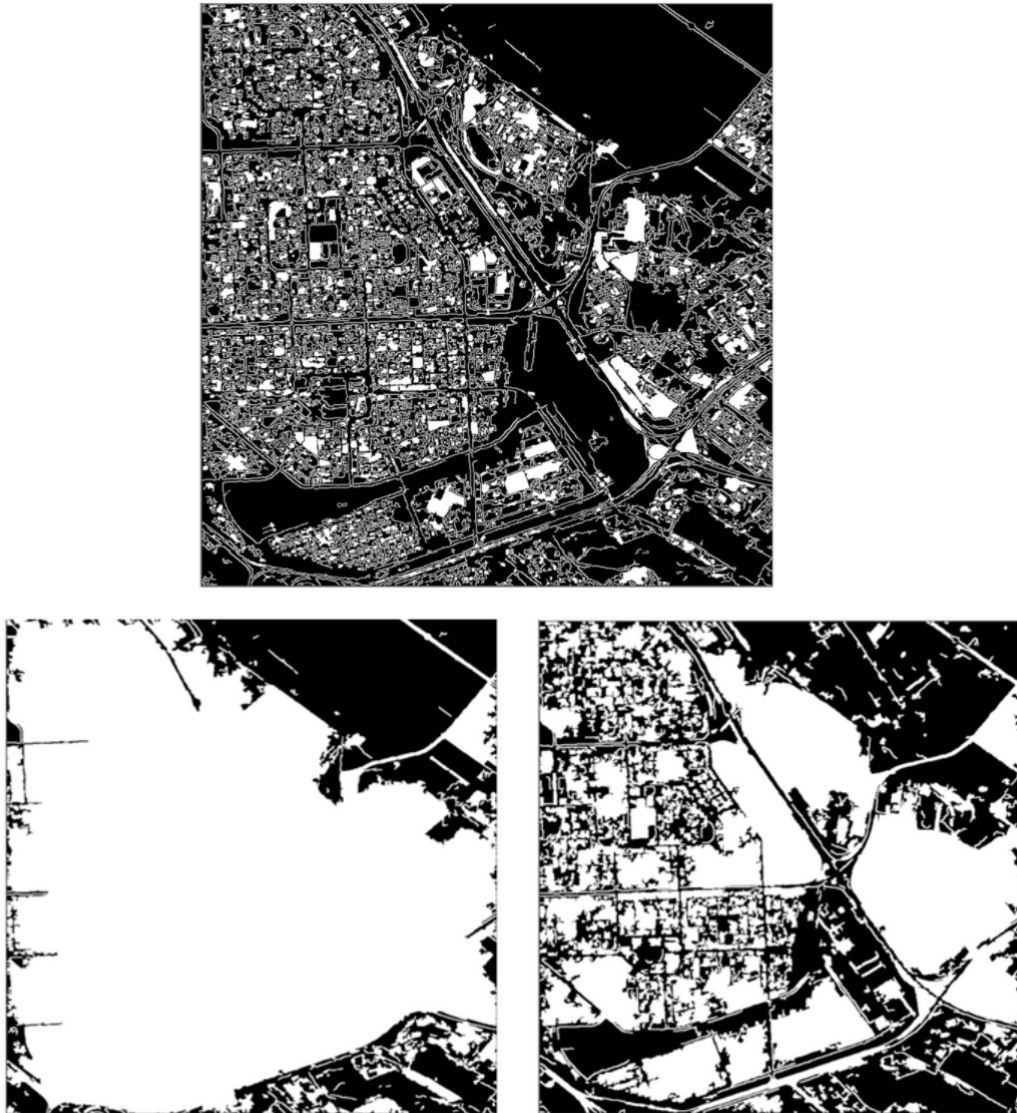


Figura 6.7: En la primera imagen se puede observar el resultado de rellenar los contornos si no se cuenta con el emborronamiento; mientras que en la parte inferior sí se ha producido dicho emborronamiento.

Para aplicar este emborronamiento se aplicará un filtro, de nuevo gracias a otro método de **OpenCV**, cuya sintaxis se encuentra en [6.12](#). Para este algoritmo solo se definirá la imagen de entrada y el tamaño del filtro a aplicar.

```
1 cv2.blur(imagen, vectorSalida, tamañoKernel, ancla, tipoDeBorde)
2
```

```

3 imagen: que se procede a emborronar.
4 vectorSalida: imagen salida del algoritmo.
5 tamañoKernel: tamaño del filtro que se va a aplicar.
6 ancla: punto que se va a emborronar tras aplicar el filtro, lo más común es situarlo
  centralizado.
7 tipoDeBorde: tratamiento del filtro cuando se sale de la imagen. Tipo de padding aplicado
  .

```

Listado 6.12: Sintaxis del método blur de OpenCV.

El filtro aplicado viene definido por la Figura 6.8, este se calcula en base al tamaño de la imagen y el tamaño de kernel elegido en el método.

$$K = \frac{1}{\text{ksize.width} * \text{ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \\ \dots & & & & & \\ 1 & 1 & 1 & \dots & 1 & 1 \end{bmatrix}$$

Figura 6.8: Filtro aplicado por blur.

Paso 3: calcular la imagen de diferencia

Tras obtener el resultado del paso anterior, se procede restando ambas imágenes, tal y como se había visto en *Image Differencing* para obtener la máscara de cambio de este algoritmo, solo que en este caso no hará falta aplicar ningún tipo de umbralización ya que ambas instantáneas solo contienen valores de 0 o 255.

6.2.2. Pseudocódigo

```

1 function PCA(imagen1, imagen2){
2     imagen1 = Canny()
3     imagen2 = Canny()
4
5     imagen1 = blur()
6     imagen2 = blur()
7     imagen1 = rellenarContornos()
8     imagen2 = rellenarContornos()
9
10    Y = absoluto(imagen1 - imagen2)
11
12    return Y

```

```
13 }  
14  
15 imagen1 : Imagen en el instante 1  
16 imagen2 : Imagen en el instante 2  
17 Y : Mapa de cambio
```

Listado 6.13: Pseudocódigo de *Canny*

6.2.3. Implementación

En este caso se utilizó una implementación propia para desarrollar los diferentes experimentos, utilizando el metodo de Canny explicado anteriormente y siguiendo el pseudocódigo.

6.3. Difference PCA

Es una técnica que ayuda a enfatizar la variación, extraer patrones de un conjunto de datos, lo que permite reducir el tamaño de estos conjuntos e incrementar su interpretabilidad. Este método consigue comprimir las imágenes manteniendo la mayoría de la información relevante. Se puede encontrar una definición más extensa en [Jolliffe and Cadima \[2016\]](#).

Esta transformación se aplica a variables correlacionadas creando nuevas variables no correlacionadas que maximizan la varianza, estas nuevas variables son los denominados componentes principales. Si se encuentran dos variables correlacionadas, hay redundancia en la información, por lo que este método se encargará de expresarlas como una combinación lineal de ambas.

Mediante el cálculo de valores propios y vectores propios asociados a estos componentes, se determina la magnitud y dirección de la variación, por lo que es fácil saber que componentes se debe descartar para así obtener los principales.

6.3.1. Funcionamiento

Paso 1: Se estandarizan los datos y se obtiene la imagen de diferencia

Se debe tener en cuenta que no todos los datos han sido tomados con las mismas unidades de medida, con la misma escala. Para poder trabajar con estos datos se deben estandarizar. En el caso de las imágenes no siempre es necesaria la estandarización, si las diferentes bandas o canales de la imagen no se encuentran en el mismo rango sí sería necesaria.

Una vez estandarizados los datos de ambas imágenes, se debe obtener la imagen de diferencia, para ello se resta píxel por píxel el par de imágenes en escala de grises.

Paso 2: Obtener los vectores y valores propios

Los vectores indican la dirección que tomarán en el nuevo espacio de características, los nuevos ejes. El valor indica la magnitud de estos, lo que permite ver la variación de los datos producida en dicho espacio.

Para obtener estos valores se debe de aplicar una factorización de la matriz de covarianzas producida a partir de la imagen de diferencia calculada anteriormente.

Paso 3: Ordenar los valores propios y escoger los k mayores

Para saber de qué vector propio se puede prescindir se pueden observar los valores propios, aquellos que tengan menor valor indican que no contienen mucha información. Estos van a ser los primeros en ser descartados.

Como ya se había mencionado anteriormente, se ordenan estos valores de mayor a menor, y posteriormente, solo se quedarían los k mayores, esta k es el tamaño al que se quiere reducir el conjunto de datos.

Paso 4: Construir la matriz de proyección

A continuación se debe construir la matriz de proyección, esta es simplemente una concatenación de aquellos vectores propios seleccionados en el proceso anterior. Se

obtiene así una matriz W cuyas dimensiones son dxk , siendo d el tamaño original del conjunto de datos.

Paso 5: Transformar la imagen original con respecto a la matriz de proyección

Finalmente solo quedaría aplicar la transformación hacia el nuevo espacio de características.

$$Y = XxW \quad (6.3)$$

donde X sería el conjunto original de datos (la imagen), W la matriz de proyección e Y el conjunto de datos en el nuevo espacio de características.

En este caso se obtendría una salida con tamaño k , el tamaño al que se quiere reducir la entrada. En este caso lo que importa de este método es la obtención de los componentes principales, por lo que el tamaño se mantiene, es decir, no hace falta aplicar ninguna k .

6.3.2. Pseudocódigo

```

1 function PCA(imagen1, imagen2){
2     imagen1 = estandarizarImagen()
3     imagen2 = estandarizarImagen()
4
5     imagen1 = pasarEscalaGris()
6     imagen2 = pasarEscalaGris()
7
8     X = absoluto(imagen1 - imagen2)
9
10    M = matrizCovarianza(X)
11    VP = factorizacion(M)
12
13    W = ordenar(VP)[: , 0:k]
14
15    Y = X x W
16
17    return Y
18 }
19
20 imagen1 : Imagen en el instante 1
21 imagen2 : Imagen en el instante 2
22 X : Imagen diferencia
23 M : Matriz de covarianza
24 VP : valores propios
25 W : Matriz de proyección
26 Y : Mapa de cambio

```

Listado 6.14: Pseudocódigo de *Difference PCA*

6.4. PCA K-Means

Para realizar el estudio se usará directamente una versión mejorada del anterior. Este algoritmo se basa en la obtención de un mapa de cambios agrupando los diferentes píxeles como “Cambio” o “No cambio” a partir de un vector de características. Este vector de características estará construido a partir de la extracción de un vector propio para cada píxel de la imagen diferencia mediante PCA. El **vector propio** es aquel que al multiplicarlo por cualquier valor, resulta un múltiplo escalar de este y que tiene la misma dirección.

6.4.1. Funcionamiento

Paso 1: Calcular la imagen de diferencia y el vector propio

El primer paso sería transformar las dimensiones de las imágenes a un número múltiplo de 5 ya que como se verá posteriormente, este algoritmo aplica bloques de 5x5 a la imagen. Este proceso de transformación de las dimensiones es reflejado brevemente en el pseudocódigo del siguiente apartado (ver 6.15) en `function cambioDimensiones()`, donde el autor divide las dimensiones de una de las imágenes entre 5 y posteriormente trunca la multiplicación del resultado anterior por 5.

La imagen de diferencia, como ya se ha mencionado anteriormente, se calcula mediante la resta píxel por píxel de ambas imágenes que se quieren comparar en escala de grises.

Antes de calcular los valores y vectores propios, este algoritmo transforma bloques de 5x5 de la imagen de diferencia en vectores, se obtiene un conjunto de vectores cuyas dimensiones serían múltiplo de 5 y se obtienen como sigue:

$$nFilas = \frac{m \times n}{5 \times 5} \quad (6.4)$$

$$nColumnas = 5 \times 5 = 25 \quad (6.5)$$

Donde m y n son las dimensiones de la imagen de diferencia.

En `buscarConjuntosVectores()` se observa como se toman estos bloques (los cuales no se deben de solapar) y se “aplanan”, se transforman en un vector. Posteriormente se ha de aplicar PCA tal y como se ha visto anteriormente, obteniendo los **componentes principales** o **EVS** (Eigenvectors).

Paso 2: Construir el espacio de características

A continuación, se vuelve a recortar la imagen en bloques de 5x5, solo que esta vez sí se solapan y se almacenarán en un vector. Es decir, para cada píxel se obtiene entonces un vector que contendrá su vecindad (5x5), obteniendo así el espacio de vectores (**Vector Space** o **VS**).

Un pequeño fallo en este algoritmo es que no considera los cambios que se puedan producir en los bordes de las imágenes, ya que en el proceso anterior de crear vectores con la vecindad de cada píxel, omite los píxeles que se encuentran en el borde y no tienen una vecindad de 5x5.

Para proyectar los vectores propios dentro del espacio de características obtenido, simplemente se debe aplicar una multiplicación.

$$VS \cdot EVS = FVS \quad (6.6)$$

Donde *FVS* sería la proyección de estos últimos en el espacio de vectores. Si se vuelve al pseudocódigo, este paso se ve reflejado en `function calcularFVS()`, donde se puede observar que los bloques se solapan.

Paso 3: Agrupar en el espacio de características

Para realizar la agrupación, se utilizará el método de **K-Means**. Se asignará a cada píxel una etiqueta que indica al grupo que pertenece. Para ello se debe observar la diferencia entre la media del vector correspondiente a cada píxel y la media del grupo, para asignarle la etiqueta de aquel al que se acerque más, es decir, se etiquetará un píxel en un grupo cuya media esté más próxima a la media del vector de características de dicho píxel.

6.4.2. Pseudocódigo

```
1 function PCAKmeans(imagen1, imagen2){
2     imagen1 = estandarizarImagen()
3     imagen2 = estandarizarImagen()
4
5     imagen1 = pasarEscalaGrisen()
6     imagen2 = pasarEscalaGrisen()
7
8     imagen1, imagen2 = cambioDimensiones(imagen1, imagen2)
9     X = absoluto(imagen1 - imagen2)
10    X = buscarConjuntosVectores(X)
11    EVS = PCA()
12
13    FVS = calcularFVS(EVS, X)
14
15    Y = KmeansClustering()
16
17    return Y
18 }
19
20 function cambioDimensiones(imagen1, imagen2){
21     tamaño = obtenerTamaño(imagen1)
22     nuevoTamaño = tamaño/5
23     nuevoTamaño = int(nuevoTamaño*5)
24
25     imagen1 = resize(nuevoTamaño)
26     imagen2 = resize(nuevoTamaño)
27 }
28
29 function buscarConjuntosVectores(imagenDiferencia){
30     for i in imagenDiferencia:
31         for j in imagenDiferencia:
32             bloque = imagenDiferencia[j: j+5, i:i +5]
33             característica = añadir(transformaEnVector(bloque))
34             j += 5
35             i += 5
36     return característica
37 }
38
39 function PCA(){
40     M = matrizCovarianza(X)
41     VP = factorizacion(M)
42
43     return VP
44 }
45
46 function calcularFVS(EVS, imagenDiferencia){
47     for i in imagenDiferencia:
48         for j in imagenDiferencia:
49             bloque = imagenDiferencia[j: j+1, i:i +1]
50             VS = añadir(transformaEnVector(bloque))
51     FVS = VS dot EVS
52     return FVS
53 }
54
```



```
55 imagen1 : Imagen en el instante 1
56 imagen2 : Imagen en el instante 2
57 X, imagenDiferencia : Imagen diferencia
58 VP, EVS : valores propios
59 FVS : Espacio de características
60 Y : Mapa de cambio
61 bloque : bloques de la imagen diferencia de tamaño 5x5
62 caracteristica : vector que contiene la vecindad 5x5 de un píxel
63 M : Matriz de covarianza
64 VS : Conjunto de vectores
```

Listado 6.15: Pseudocódigo de PCA K-Means

6.4.3. Implementación

Para este caso específico, se ha tomado el código desarrollado por **Abhijeet Kumar** en [Jolliffe and Cadima \[2016\]](#), el cual se basó en un *paper* sobre este mismo método ([Celik \[2009\]](#)). Este sigue el pseudocódigo anterior, pero hubo que hacer ciertas modificaciones para que funcionara con imágenes de cualquier tamaño y con tres bandas.

Esta implementación se encuentra en su repositorio de GitHub ([Celik \[2019\]](#)), donde además incluye diferentes imágenes de prueba.

Capítulo 7

MÉTODOS DE APRENDIZAJE PROFUNDO DESARROLLADOS

A lo largo de este capítulo se desarrollarán y adaptarán diferentes métodos de Aprendizaje Profundo, ya que estos son una evolución de las Redes Neuronales clásicas, se procede a hacer una descripción sobre el funcionamiento de las mismas. De esta manera, al describir las usadas para este proyecto, se podrá también analizar la estructura de estas de una forma más sencilla.

7.1. Fundamentos de las Redes Neuronales

La **Inteligencia Artificial** ([Russell \[2006\]](#)), como ya se ha mencionado en secciones anteriores, consiste en generar diferentes técnicas para que las máquinas sean capaces de resolver problemas que para el ser humano tendrían una solución muy intuitiva. Dentro de este campo se pueden encontrar múltiples técnicas, pero la que mejor resultados ha demostrado en áreas como la Detección de cambios en imágenes, en el que se centra este proyecto, son las **Redes Neuronales** ([Rosebrock \[2017\]](#)). Como se estudiará más adelante, se obtiene un buen rendimiento a la hora de etiquetar cambios entre pares de imágenes.

Las Redes Neuronales son un tipo de algoritmos inspirados en la estructura y funcionamiento del cerebro humano, donde se puede encontrar más de 10 billones de neuronas interconectadas entre ellas. Se pueden aplicar estas redes a multitud de problemas, tanto lineales como no lineales. El ejemplo más sencillo de estos problemas puede encontrarse en los operadores binarios, siendo **AND** y **OR** problemas que

se pueden resolver linealmente, con una línea que separe ambos conjuntos de datos que se quieren clasificar; mientras que con **XOR** se encuentra un problema no lineal, es decir, sus conjuntos no pueden ser separados linealmente. Las Redes son capaces de estudiar este entorno y de hallar esta solución no lineal si se aplica, como se verá más adelante (ver [Función de activación](#)), una función de activación a la salida de las diferentes neuronas. Se puede ver un ejemplo representado gráficamente en la Figura 7.1. Con AND se representan los resultados sobre un espacio siendo los puntos 0 y los cuadrados 1, se puede observar como este problema se resuelve linealmente; a la derecha se puede ver como XOR, con la misma notación, no puede separar los conjuntos linealmente.

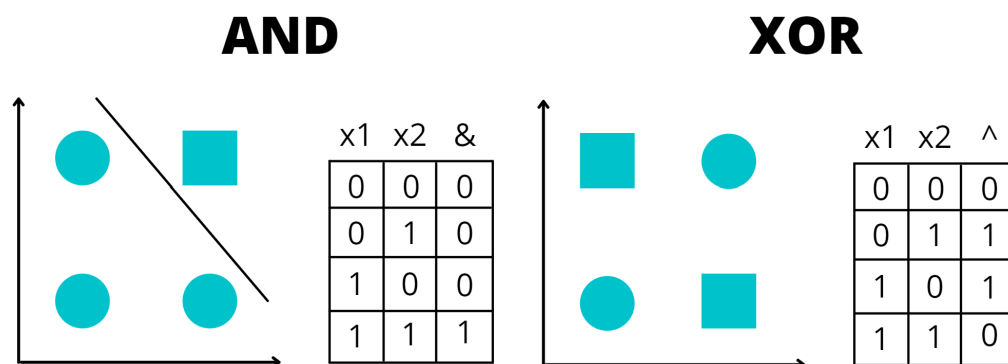


Figura 7.1: A la izquierda se puede encontrar un ejemplo de los diferentes tipos de problemas lineales (AND) a la izquierda y no lineales (XOR) a la derecha. Esta figura ha sido recreada a partir del trabajo de [Rosebrock \[2017\]](#).

Dentro de las Redes Neuronales se debe destacar el área del **Deep Learning** ([Witten and Frank \[2005\]](#)), que aprovecha múltiples capas de procesamiento para realizar una gran cantidad de operaciones, estas capas estarían compuestas por neuronas, las cuáles realizan a su vez pequeñas operaciones. Esta profundidad, es lo que le da la capacidad a los modelos para detectar patrones complejos en los diferentes conjuntos de datos, pudiendo así resolver problemas lineales o no lineales.

7.1.1. Elementos de una red neuronal

Ya se ha adelantado anteriormente que uno de los principales elementos, la base de las Redes Neuronales, son las **neuronas**. Pero no solo se cuenta con estas, sino que también se deben considerar las **capas** o arquitectura del modelo, la **función de pérdida** y el **método de optimización** así como los **datos de entrada**.

Neuronas

Estas billones de neuronas biológicas que se encuentran conectadas en el cuerpo tienen unas **partes** muy diferenciadas, no muy distintas a las que hay en una neurona de un modelo. Al igual que estas últimas, funcionan mediante impulsos que reciben por sus entradas o dendritas las cuales activan o no la neurona según un criterio o función de activación, para terminar propagando dicha información por el axón o salidas hacia otras neuronas. El funcionamiento de estas es muy simple y a la vez complejo si se tiene en cuenta el conjunto de estas.

- **Input:** el dato de entrada a la neurona sobre el que se va a aplicar la operación, la multiplicación del peso por el valor de la entrada. En este caso concreto, el valor de entrada se corresponderá al valor de la intensidad de un píxel, y gracias a las conexiones con otras neuronas, se podrá obtener información de vecindad de este píxel, por lo que en realidad se tendrán múltiples entradas.
- **Peso:** el valor por el cual se multiplica la entrada, se encuentra en la conexión entre la entrada y la neurona. Modificando este parámetro se consigue que la red “aprenda”.
- **Bias:** permite a la neurona modificar la salida independientemente de la entrada.
- **Output:** la salida de la neurona, la clasificación de un elemento dentro de un conjunto de clases predeterminadas. En el campo de la Detección de cambios se encuentran las dos clases con las que ya se ha trabajado anteriormente, si se ha producido un cambio o no, para cada píxel un conjunto de neuronas realizará una predicción. Se puede calcular siguiendo la fórmula:

$$Y = WX + b \quad (7.1)$$

Denotando la Y la salida, X la entrada, W el peso y b el Bias.

Gracias a la Figura 7.2 se puede visualizar la estructura de estas neuronas artificiales con las biológicas, para ver que tienen una gran similitud. Por ejemplo se encuentran las Dendritas y el Axón, que serían las correspondientes entradas y salidas de la neurona artificial que se encuentra en la parte inferior.

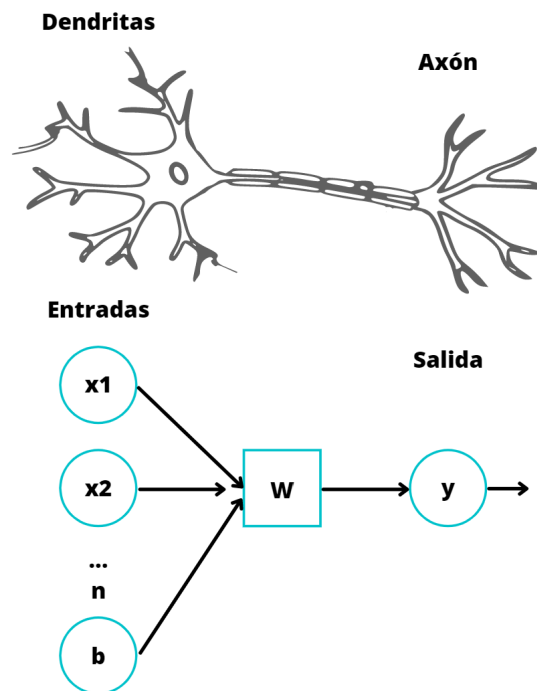


Figura 7.2: Comparación entre una neurona humana (superior) y una artificial (inferior).

Función de activación

Tener múltiples neuronas conectadas, sin nada entre medias que modifique el resultado obtenido en la anterior, es como si se tuviera solo una gran neurona. Con esto lo único que se obtendría serían siempre soluciones lineales, el problema es que no todos los problemas que encontrados son lineales, por lo que se necesitan las funciones de activación.

Gracias a estas las Redes Neuronales pueden aprender relaciones no lineales. Estas se aplican a las salidas de las neuronas y decide si esta se activa o no, es decir, si su salida servirá para realimentar la siguiente capa o no.

Se encuentran numerosas funciones de activación, cada una cuenta con unas características concretas que se pueden aplicar en diferentes situaciones o necesidades:

- **Step**: la más sencilla de todas, si la salida de la neurona es positiva, esta devuelve 1, por el contrario al ser negativa devolvería 0.
- **Sigmoid**: esta es de las más usadas en las Redes Neuronales debido a que:
 - Es continua y diferenciable.
 - Es simétrica para el eje Y.

- Se aproxima asintóticamente hacia sus valores de saturación.

Sin embargo, se pueden encontrar problemas al hacer uso de esta ya que, aunque esté centrada en el eje Y, no está centrada en 0. Esto produce una “muerte” en el gradiente, es decir, que para valores muy bajos de este, las neuronas no se activaran nunca.

- **Tanh**: sería la versión mejorada de la anterior, ya que la forma es similar y en este caso sí se encuentra centrada en 0. Aún así se sigue encontrando el mismo problema.
- **ReLU**: está demostrado que su funcionamiento es lo más similar a las neuronas biológicas. Estas sencillamente devolverán como salida 0 para valores negativos, pero si son positivos su valor crece linealmente. Resuelven el problema anterior, pero surge uno nuevo, algunas neuronas pueden “morir”, permanecer inactivas ya que por ellas no fluye ningún tipo de gradiente.
- **Leaky ReLU**: esta vez se utiliza un nuevo parámetro, α , que varía la pendiente de la parte izquierda del eje x, lo que evita que estas neuronas se desactiven.

La práctica más común es aplicar una ReLU para obtener una precisión base y a partir de aquí aplicar otros tipos de ReLU como Leaky para mejorar el modelo y obtener así una mayor precisión aún.

Métodos de optimización

Cuando se inicializa la red, los pesos pueden ser inicializados de múltiples formas, se puede aplicar una inicialización aleatoria que siga una Uniforme o una Normal, o bien inicializar todos los valores con una constante predefinida. Se debe tener en cuenta que en este punto la red aún no sabe nada, no ha aprendido. Para que esta aprenda se deben ajustar los pesos de tal manera que sea capaz de transformar los inputs en un objetivo, la salida.

El objetivo será entonces conseguir una mayor precisión y minimizar las pérdidas. Para lograrlo se debe utilizar las **funciones de pérdida**, encargadas de calcular la disparidad entre el valor que ha predicho el modelo y el valor real. Es lógico pensar, que a menor pérdida, mejor es el modelo y por lo cual mayor es su precisión. Lo ideal es que esta pérdida se vaya reduciendo a medida que itera para entrenar el modelo.

Existen numerosas funciones de pérdida, pero la más usada en este campo es la denominada **Cross-Entropy Loss**, la cual devuelve una probabilidad para cada clase,

lo que permite observar el margen de error para cada una de ellas y finalmente calcular la pérdida sufrida.

$$L_i = -\log(e^{S_y} / \sum_j e^{S_j}) \quad (7.2)$$

Con esta fórmula se calcula la pérdida de una imagen en concreto, siendo la total de todo el dataset la media de todas las clases posibles. Se encuentra entonces en S , el resultado de la función de puntuación o la salida de la neurona sobre y , el resultado real, y j la clase que está evaluando.

Pero únicamente con el cálculo de este error, aún no se puede entrenar la red correctamente, aún no se ha modificado el peso de las neuronas para que estas empiecen a reconocer patrones. Para ello se necesita un método de optimización, un algoritmo iterativo que trabaja en un entorno de pérdida, el cual se desconoce (en la Figura 7.3 se puede ver un ejemplo de entorno, el cual puede variar según el modelo utilizado). El objetivo va a ser obtener un mínimo tras probar diferentes valores de W y b para evaluar su pérdida y así dar un paso hacia este valor óptimo. Se debe tener en cuenta que al comenzar la ejecución de la red, esta se sitúa en una posición aleatoria de este entorno y se irá moviendo por este, como ya se ha comentado antes, preferiblemente hacia los mínimos.

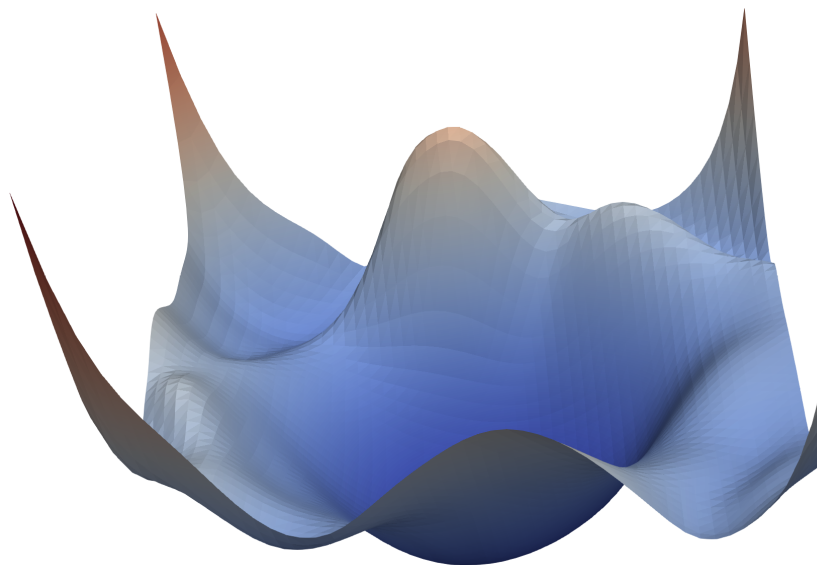


Figura 7.3: Ejemplo de espacio de pérdida generado por una Red Neuronal y sus múltiples parámetros. Imagen extraída de [Li et al. \[2018\]](#).

Se puede resumir su funcionamiento en unos pasos:

1. Se coge un conjunto de datos y se pasa por la red para que esta devuelva sus

predicciones. Estos subconjuntos de datos son denominados *batches*, no tienen porque pasar por todos los datos del dataset para actualizar la matriz de pesos. Estos **batches** suelen ser de un tamaño múltiplo de 2.

2. Se calcula la pérdida de dicho subconjunto.
3. Se ajustan los pesos para minimizar esta pérdida sufrida

Este proceso se hace durante un determinado número de epochs con varios *batches*, estos epoch se refieren al número de veces que la red va a recorrer cada dato del dataset. Cada vez que visite todos los puntos, habrá concluido un epoch.

Uno de los algoritmos de optimización más usados es el **Gradiente Descendente (SGD)**, que como bien indica su nombre utiliza el gradiente para “guiar” a estos parámetros hacia un mínimo. Este es un vector que indica hacia qué dirección se deben actualizar los pesos para minimizar la pérdida. Se llama descendiente, ya que como se ha comentado, se usa el gradiente para poder moverse en el espacio de pérdida hacia la zona con menor pérdida, donde la curva esté muy próxima al mínimo. Es estocástico porque se usan pequeños conjuntos aleatorios del dataset, los *batches*.

Para actualizar los pesos se aplica la siguiente función:

$$W = W - \alpha * W_{gradient} \quad (7.3)$$

Donde W es la matriz de pesos, α la tasa de aprendizaje (ver 7.1.1) y $W_{gradient}$ el gradiente de los pesos para dicha zona del espacio de pérdida en el que se encuentra localizado actualmente.

Sin embargo, no se puede estar constantemente actualizando esta matriz. Por ello se debe parar cuando se lleve un determinado número de epochs, la pérdida sea lo suficientemente baja y por consecuencia, aceptable, la precisión es lo suficientemente alta o bien cuando la pérdida no ha mejorado en un número determinado de iteraciones.

Aún con todos estos detalles el modelo no será capaz de generalizar, se debe regularizar. Ayudará a controlar la capacidad del modelo de clasificar aquellos datos para los cuales no ha sido entrenado, pero se debe tener cuidado, ya que demasiada regularización puede suponer problemas de *underfitting*. Existen diversos métodos para ello:

- **L2**: también conocida como *weight decay*, es de las más usadas ya que con ella se consigue dar una mayor importancia a los pesos más pequeños, siendo así

capaces de generalizar más que si siempre se le da importancia a los pesos más grandes. Sería calculada tal y como se puede ver en la ecuación 7.4

$$R(W) = \sum_i \sum_j W_{i,j}^2 \quad (7.4)$$

Siendo el sumatorio de los cuadrados de todos los pesos del modelo

- **L1**: Es exactamente igual que L2, solo que se toma el valor absoluto en lugar del cuadrado como se puede comprobar en 7.5.

$$R(W) = \sum_i \sum_j |W_{i,j}| \quad (7.5)$$

Estas técnicas de regularización serán añadidas a la hora de actualizar los pesos en la matriz del modelo en el momento de calcular la pérdida (ver 7.6).

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W) \quad (7.6)$$

Donde L es la pérdida total de ese batch calculada como el sumatorio de las pérdidas de cada uno de los datos que componen dicho subconjunto por λ , la penalización de la regularización (con esto se controla la potencia de la regularización que se está aplicando) y finalmente por el método de regularización escogido.

Con todos estos elementos, la red debe de ser capaz de aprender a buscar patrones en las imágenes y a modificar los diferentes parámetros para que aprenda a generalizar y a actuar ante situaciones tanto conocidas como desconocidas.

Tasa de aprendizaje

O en inglés **Learning Rate** es uno de los parámetros más importantes que se deben ajustar en el modelo. Este lo que hace es controlar el tamaño del paso que se da en dirección al gradiente. Si es muy grande estaría constantemente dando vueltas por el entorno de pérdida; sin embargo, si es pequeño se avanza poco a poco, por lo que se traduce en la necesidad de más iteraciones. Por ello es importante regularlo correctamente, para que no ocurra ninguna de las problemáticas anteriores, pero es bastante complejo.

Una buena práctica para controlar lo anterior, es empezar con un valor y poco a poco ir reduciéndolo en cada epoch. Con esto se consigue encontrar en este espacio

una zona donde la pérdida sea mínima, sin necesidad de que esta sea un mínimo global, ya que encontrarlo sería muy costoso computacionalmente, a veces hay que conformarse con que sea “lo suficientemente bueno”. Se encuentran dos maneras de proceder:

- **Standard decay Schedule:** se actualiza la tasa de aprendizaje dependiendo del número de epoch en el que se encuentre durante el entrenamiento. Se establece un nuevo parámetro o decay que será multiplicado por el número de la iteración, esto servirá para ir disminuyendo poco a poco hasta encontrar un valor estable para la tasa.
- **Step-based Decay:** En este caso se decrementa el valor en determinados epoch durante el entrenamiento. El *Learning Rate* se mantiene constante durante este determinado número de epochs.

7.1.2. Tipos de capas

Estas Redes tienen que seguir una arquitectura, es decir, se deben conectar las neuronas siguiendo un esquema de conexión entre estas. Para hacer estas conexiones se deben considerar diferentes tipos de capas, las cuales se conocen como capas ocultas o *Hidden Layers*, siendo “visibles” solo las entradas y la salida de la red. Cada una de estas tienen funciones diferentes que se deben de contemplar.

Capas de convolución

Compuestas por un conjunto K de matrices o filtros que son capaces de aprender. Cada uno de estos filtros se aplica a un input y a su vecindad, generando como salida una pila con el resultado de la convolución de cada kernel con este conjunto de inputs, el **mapa de activación** de igual tamaño que la matriz de entradas. Para realizar la convolución sobre una entrada, simplemente se debe usar dos matrices de igual dimensión, multiplicar elemento por elemento y finalmente sumarlo. En el caso del modelo se tienen numerosos inputs en forma de matriz, para aplicarle la convolución se usarán una o más matrices pequeñas, la cual se irán moviendo por todas estas entradas para estudiar los píxeles vecinos.

Para su construcción, se deben tener en cuenta ciertos aspectos para controlar el tamaño del output, ya que como se ha podido comprobar, el tamaño puede llegar a ser excesivo dependiendo de determinados parámetros:

- **Profundidad (Depth):** indica el número de filtros que se quieren aplicar al input.
- **Paso (Stride):** como se ha visto, una convolución normal daría como resultado un output con las mismas dimensiones que el input, pero con más profundidad. Una manera de solucionarlo es hacer que el paso para calcular la siguiente convolución se “salte” unos cuantos píxeles más. Lo más normal es tener un salto igual a 1 ó 2.
- **Relleno (Padding):** se deben considerar los límites del input a la hora de aplicar la convolución, ya que si se quiere aplicar sobre todos los píxeles de la imagen, en estos límites el kernel sobresaldría y no sería capaz de realizar dicha operación. Para solucionarlo se puede añadir píxeles cuyo valor sea 0 o repliquen los valores de los píxeles de bordes opuestos. De esta manera se conseguiría calcular la convolución sobre todos los píxeles y el tamaño del output será igual al del input original, sin el *padding*.

Una vez escogidos todos estos parámetros se puede comprobar si se ha realizado correctamente la construcción de esta capa si se aplica la fórmula 7.1.2

$$\frac{(W - F + 2P)}{S} + 1 \quad (7.7)$$

Donde W indica el ancho del input, F el tamaño del kernel, P indicaría el *padding* aplicado así como S el *stride*. Esta operación debe resultar un numero entero para comprobar que realmente se han aplicado correctamente los valores de estos parámetros.

Estas capas tendrán una función muy importante en el modelo, los primeros niveles de estas se encargarán de extraer características simples como líneas o contornos, mientras que los últimos niveles extraerán las características más complejas como patrones o elementos de una forma más detallada.

Capas de activación

Como ya se ha visto en el apartado de 7.1.1 [Función de activación](#), de debe aplicar una función de activación a las neuronas, de esta manera estas son capaces de “desactivar” ciertas partes del modelo y poder de esta manera resolver problemas no lineales. Aunque realmente no tendría que considerarse como una capa ya que

no contiene ningún parámetro que deba de ajustarse, esta se encarga de aplicar dicha operación a la salida de otras capas.

Este proceso siempre debe de ser realizado tras finalizar la capa de convolución, por lo que debe de aparecer implícito en la construcción de la red.

Capas de agrupación

La principal función de estas es la de reducir (tanto en anchura como en altura) el tamaño de los inputs; reducir el número de parámetros que la red debe de calcular. Aunque esta también ayudará a la hora de evitar el overfitting y a favorecer aquellas señales de activación que tengan una mayor importancia.

Lo que hacen simplemente es recorrer el input en bloques de tamaño fijo (normalmente 2x2) y se queda con un determinado valor. En este caso, se volverá a contar como parámetro el paso o *stride*, que indica cuánto se tiene que mover el bloque para la siguiente iteración. Se pueden encontrar dos tipos dependiendo del valor que se decida escoger:

- **Max pooling:** usada en posiciones intermedias de la red. Como indica su nombre, al recorrer el input, se queda con el píxel cuyo valor sea mayor.
- **Average pooling:** usada al final de la red. Y de nuevo, como indica su nombre, hace la media entre los valores del bloque.

Como se ha visto antes, las capas de convolución también sirven para reducir el tamaño del output, esto explicaría porque la tendencia actual es evitar el uso del max pooling.

Capas de normalización de lote

O por su nombre en inglés *Batch Normalization*, son usadas para normalizar las activaciones de las entradas antes de que estas pasen a la siguiente capa. Con esto se consigue reducir drásticamente el número de epochs, así como tener una menor pérdida y la prevención del overfitting.

Esta capa debe de ir situada pues, tras la de activación, aunque existen casos (escasos) en los que funciona mejor si se pone antes de esta.

Capas completamente conectadas

Situadas en la parte final del modelo, estas se construyen al convertir la matriz de inputs en único vector donde todas las neuronas se encuentran conectadas con todas las de la capa siguiente.

Dropout

Esta en sí no es una capa, pero ayudan al modelo a obtener una mayor precisión y le enseña a generalizar. Actúa como las capas completamente conectadas, solo que de manera aleatoria y siguiendo una probabilidad, algunas de estas conexiones se “desactivan”. De esta manera consiguen el objetivo anterior de mejorar la clasificación de la red, ya que demuestra que algunos patrones pueden ser detectados por diferentes neuronas, no siempre por las mismas.

Para poder usarlo se debe definir la probabilidad, P , que indica el número de conexiones que van a estar activas y se deben situar entre las completamente-conectadas.

7.1.3. Aspectos a considerar

Antes de finalizar con los fundamentos de las Redes Neuronales, cabe destacar el proceso de creación y desarrollo de las mismas, como deben de ser usadas. Para ello se debe tener en cuenta la existencia de tres modalidades de aprendizaje automático:

- **Supervisado:** hay pocos datos, pero todos estos están totalmente etiquetados.
- **No supervisado:** hay una mayor cantidad de datos, pero estos no están estructurados, es decir, no se encuentran etiquetados. Al final los resultados deberán de ser comprobados mediante un proceso manual por parte de un experto.
- **Semi-supervisado:** una mezcla de lo anterior, se tienen muchos datos, pero solo una parte de ellos están etiquetados.

A la hora de afrontar la detección de cambios, y como bien se explicó en el apartado [DATASETS](#), se encuentran numerosos datasets etiquetados, pero estas máscaras no se encuentran completas, ya que solo muestran cambios de tipos muy determinados tales como modificaciones en construcciones. Por ello se debe tener en cuenta que en el proceso de aprendizaje, las Redes aprenderán de forma semi-supervisada y

supervisada, ya que se puede considerar que se quieren encontrar todas las diferencias entre dos pares o que solo se quieren detectar estos cambios en las diferentes edificaciones.

El proceso de desarrollo de una red neuronal tiene unos pasos específicos:

1. **Preparar el dataset de entrenamiento:** debiendo establecer el tipo de aprendizaje, el cual siempre es preferible que sea supervisado. También se debe de comprobar que la cantidad de etiquetas de cada tipo sean equitativas, ya que si no se consigue, se puede enseñar a la red más sobre una clase, llegando a suponer un problema de *overfitting*.
2. **Separar estos datos:** para preparar la red se debe pasar por un proceso de entrenamiento en el cual se van ajustando todos los parámetros comentados anteriormente tales como la matriz de pesos, el bias, etc. Para ello primeramente se necesita un **conjunto de entrenamiento** del cual también se extraerá un **conjunto de validación**, para ajustar los hiper parámetros (la cantidad se corresponde entre un 10 y un 20 %). Una vez entrenada la red se debe probar todo lo que ha aprendido con un **conjunto de testeo** cuyo tamaño debe ser menor.
3. **Entrenar la red:** una vez separados estos datos se debe recorrer todos los datos del dataset en determinados **epoch** y aplicar técnicas de aprendizaje como la **propagación hacia atrás** para modificar los parámetros.
4. **Evaluar:** Una vez terminado el entrenamiento, se debe comprobar que la red funciona correctamente, para ello se probará el conjunto de testeo y se comprobarán diferentes gráficas para ver si se ha producido algún error al ejecutar la red tales como *overfitting* o *underfitting*.

Como se verá a continuación, al probar los diferentes modelos, se debe seguir todo este proceso. De hecho el primer paso se ve reflejado en el apartado [5 DATASETS](#), donde se recopilan los diferentes datasets y se especifica cuales de ellos van a ser usados. También se explica porque se considera semi-supervisada. La mayoría de estos datasets ya cuenta con carpetas que separan los que serían los datos de entrenamiento de los de testeo así como sus máscaras.

A la hora de evaluar la red se tendrá en cuenta las métricas estudiadas en el apartado [4 MÉTRICAS](#) así como revisar las gráficas generadas por el modelo durante el proceso de entrenamiento y validación para descubrir si se ha producido algún tipo de sobreajuste o subajuste.

Con el concepto de sobreajuste, se quiere indicar que el modelo se ajusta demasiado a los datos de entrenamiento, por lo que no es capaz de generalizar en los datos de validación. Un modelo que sufre *overfitting* tiene un buen rendimiento durante el entrenamiento, pero a la hora de probar los datos de testeo se ve como no es capaz de realizar una clasificación con el porcentaje de precisión que se esperaba y que habría mostrado el proceso de entrenamiento. El subajuste hace referencia a que el modelo tiene una gran pérdida, es decir, comete muchos errores ya que falla a la hora de entrenar. Con este problema se encontraría un modelo que no ofrece buen rendimiento, ni en la etapa de entrenamiento, ni en la de testeo.

Se puede comprobar si se ha producido alguno de estos sucesos si se observa la gráfica generada por el modelo. Se debe procurar que la brecha entre entrenamiento y prueba sea mínima, si se observa la Figura 7.4 se puede ver como se obtienen dos líneas que marcan el error en el entreno y en la validación. A medida que se va incrementando la capacidad del modelo, se llega a una zona óptima a partir de la cual, si se sigue avanzando, estas se van separando. Es en ese punto en el cual se debe de dejar de entrenar. En la primera parte de la gráfica se puede observar como el modelo en un principio no “sabe” nada, por lo que se produce *underfitting*. Está empezando el proceso de aprendizaje, intentando alejarse de las inicializaciones aleatorias de las matrices de peso hacia valores “aprendidos”. En la Figura mencionada anteriormente se puede ver como en la primera gráfica, la separación entre la pérdida de validación y entrenamiento es cada vez mayor, lo que indica que se está produciendo *overfitting*; mientras que en la parte derecha se puede ver como el otro modelo mantiene el tamaño de esta brecha constante

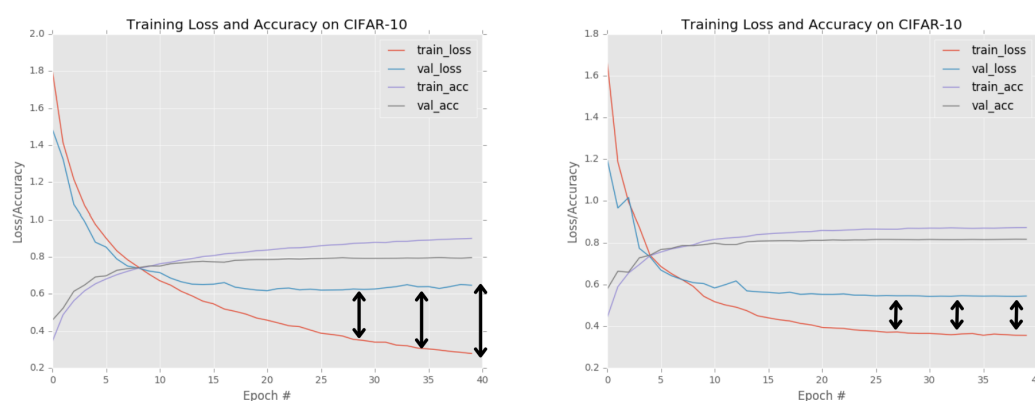


Figura 7.4: En estas gráficas se muestra una representación del resultado de dos Redes diferentes sobre el dataset de CIFAR-10.

En el caso concreto de este trabajo no se desarrolla ninguna red neuronal, pero si se pueden proponer ciertas mejoras si se observa cualquier tipo de anomalía en las

salidas. Para poder evitar el *overfitting* existen diferentes estrategias tales como:

- Usar capas de agrupación, ya que en estas no se tienen tantos parámetros que ajustar.
- Usar capas de normalización por lote, ya que normaliza las entradas a las neuronas. Gracias a estas se necesitan menos epoch, lo que previene la aparición de sobre aprendizaje.
- Reducir la complejidad del modelo, de esta manera también se evita tener que ajustar multitud de parámetros.

Una práctica muy recomendable a la hora de realizar este proceso es almacenar los pesos a medida que se va avanzando por los diferentes epochs, de esta manera si se detecta que en algún momento se produce esta separación en la brecha, se detenga el proceso de entrenamiento.

7.2. Red Siamesa y Encoder-Decoder

La mayoría de las Redes que se han revisado para este trabajo estaban basadas en otras arquitecturas comunes que se pueden encontrar a lo largo de toda la literatura sobre Aprendizaje Profundo, más concretamente las que dan nombre a este apartado, las **Redes Siamesas** y los **Encoder-Decoder**. Existen otras arquitecturas, basadas todas ellas en el uso de Redes **Convolucionales**, es decir, que aplican numerosas capas de convolución con diferentes números de *kernels* para poder extraer con estas diferentes características. Como ya se ha mencionado, el objetivo es extraer características a alto nivel con las primeras capas de convolución, para posteriormente, extraer otras a bajo nivel a medida que se acerca a la salida de la red.

Los mencionados anteriormente también funcionan con multitud de capas convolucionales, solo que se sitúan siguiendo unos criterios que varían con respecto a las Redes Convolucionales por defecto.

- **Siamesas**: esta primera aproximación consiste en disponer dos Redes de iguales características y arquitectura funcionando en paralelo y en la mayoría de los casos compartiendo parámetros tales como los pesos. A cada una de las “ramas” se le pasará como entrada una de las imágenes pertenecientes al par para

obtener al final el mapa de cambio. Cada rama se encarga de extraer las principales características, mediante capas de convolución, de estas imágenes para basar en estas sus comparaciones.

- **Codificadoras-Decodificadoras:** o como han sido llamadas hasta ahora *Encoder-Decoder*. En este caso no se cuenta con estructuras paralelas, pero el objetivo es similar, al comprimir y descomprimir la imagen se extraen las características más importantes de estas para posteriormente usarlas en la clasificación. Es sobretodo de ayuda en la clasificación semántica ya que con la compresión se obtienen las características más importantes, y tras aplicar decompresores se descubre que píxeles pertenecen a determinadas categorías.

Un claro ejemplo de *Encoder-Decoder* sería **UNet**, desarrollada por [Ronneberger et al. \[2015\]](#), cuya explicación se puede encontrar en el trabajo de [de Jong and Bosman \[2019\]](#), una de las arquitecturas más destacadas y la que se toma como base para muchas de las experimentaciones que se comentarán a continuación. Recibe este nombre, ya que como se puede observar en la Figura 7.5, tiene forma de U; siendo la primera parte el codificador y la siguiente parte el decodificador. Esta estructura se suele conseguir aplicando capas de Convolución y *MaxPooling*, para dar paso a continuación a más capas de De-Convolución que se diferencian de las de convolución en el uso de características extraídas de capas anteriores del codificador. Si se observa la figura, se puede ver como la salida del codificador se copiaría en el decodificador del mismo nivel, con esto lo que se pretende es obtener una mayor relación con el contexto, ser capaces de encontrar aquellos componentes claves más fácilmente además de ayudar a decodificar la imagen correctamente.

El problema de estas Redes es que únicamente cuentan con una entrada, y para la Detección de Cambios se dispone de un par de imágenes que se deben de analizar. Una de las soluciones aplicadas, y como se verá más adelante, es concatenar ambas imágenes y procesarlas a la vez, y tras, esto calcular la imagen de diferencia. Pero ante esta problemática también se encuentra otra posible solución, las Redes Siamesas. Como ya se adelantaba antes, se cuenta con dos estructuras idénticas que se procesan en paralelo y comparten sus parámetros. Por lo que se recrearán estas dos ramificaciones y a cada una de ellas se le introducirá una imagen del par.

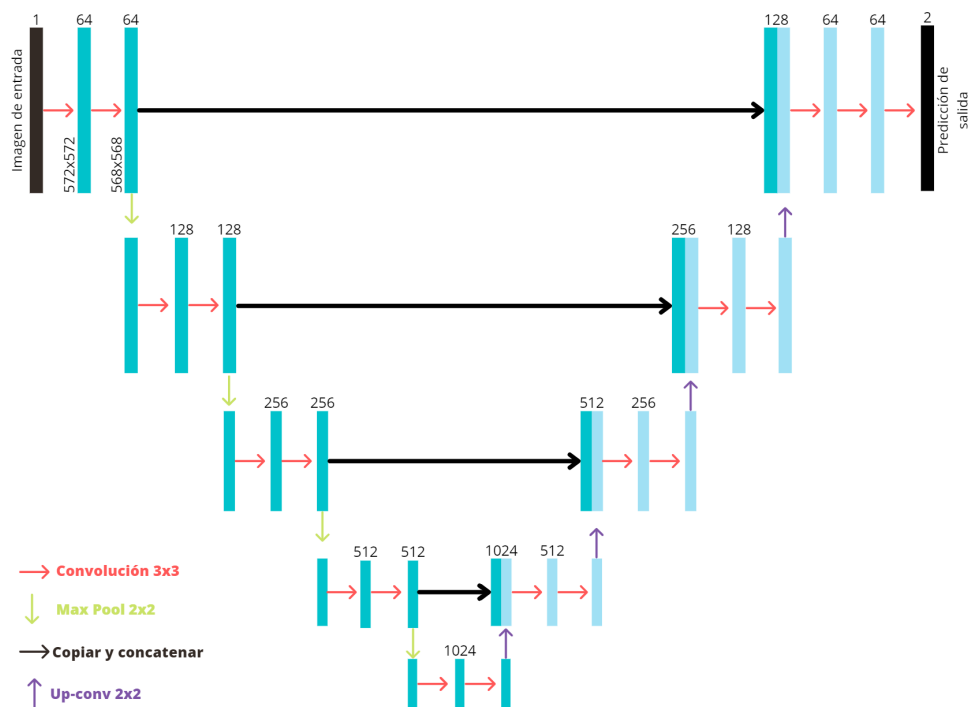


Figura 7.5: Estructura original de *U-net*, imagen recreada a partir de la mostrada en [Tomar \[2021\]](#).

7.3. Fully Convolutional siamese networks for change detection

Uno de los primeros modelos a tratar consiste en una combinación de *Encoder-Decoder* y Siamesa presentado en [Daudt et al. \[2018\]](#). En dicho trabajo se presentan realmente 3 modelos, dos siamesas y una red *UNet* modificada. Se puede observar la estructura de estas Redes en la Figura 7.6, siendo (a) la red Fully Convolutional Early Fusion (FC-EF), (b) Fully Convolutional Siamese-Concatenation (FC-SC) y (c) Fully Convolutional Siamese-Difference (FC-SD). Todas ellas son variantes de *UNet*, en las cuales se disminuye el número de las capas de Convolución y *MaxPooling*, en la primera versión de esta se contaba con 5 capas de cada tipo, en este artículo se reducen a 4; además, las profundidades de las capas de convolución también se ven reducidas drásticamente.

Para la primera de estas versiones, **FC-EF**, se cuenta con un simple *Encoder-Decoder*, cada salida del codificador servirá para alimentar la capa del decodificador paralela, tal y como se puede observar en la imagen anterior (las flechas negras). Concatenando ambas entradas a esa capa de la red lo que se pretende es conse-

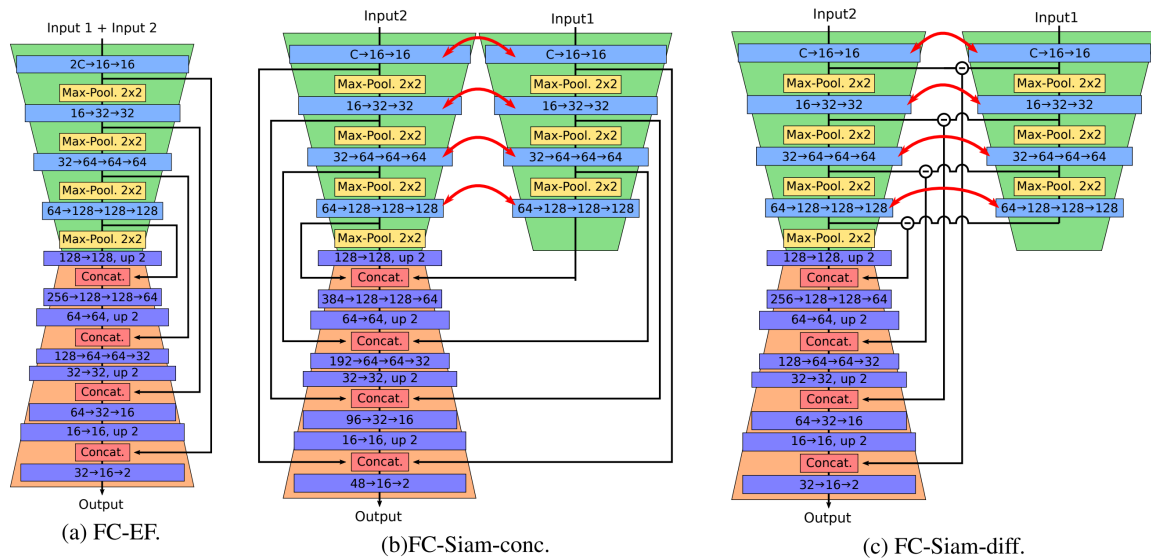


Figura 7.6: Estructuras de las diferentes variantes, extraídas de [Daudt et al. \[2018\]](#). Las flechas negras indican las diferentes conexiones entre entradas y salidas, las rojas son los parámetros compartidos.

guir complementar la información que se obtiene al comprimir la imagen, ya que esta puede resultar abstracta y con escasa información espacial, por lo que con estas conexiones se mantiene esta información, obteniendo resultados más precisos a la hora de decodificar.

Como en este caso solo se cuenta con una entrada, las imágenes deben de ser concatenadas, de tal manera que se procesarán como si de diferentes bandas se trataran, por lo que el input tendría una profundidad de 6 bandas si se trata con imágenes RGB.

En cuanto a las Redes de **FC-SC** y **FC-SD**, ambas son Redes cuya primera porción, el codificador, se encuentra ramificado en dos, por ello pueden ser consideradas como Redes siamesas. Por lo que ya no se encuentra el problema anterior, ahora cada imagen será introducida en su propia entrada y procesada individualmente, para posteriormente concatenarse a la entrada del decodificador. Sin embargo, se encuentra una diferencia entre ellas; mientras que la primera concatena las salidas de ambas capas de codificación paralelas en la correspondiente de decodificación; la segunda concatena el valor absoluto de la resta, píxel por píxel, de las salidas de las capas de ambas ramas. Este comportamiento se puede observar más claro en la Figura 7.6, donde las flechas negras indican las diferentes conexiones realizadas, en (b) se puede ver como la salida de cada capa se concatena, mientras que en (c) se puede ver la concatenación de la resta.

Como se puede observar, estas Redes tiene una estructura muy sencilla, y como se comprobará en el apartado de [8.1.4 Experimentaciones con Fully Convolutional siamese networks for change detection](#), se obtendrán muy buenos resultados, mejorando en gran medida con respecto a cualquiera de los algoritmos clásicos probados con antelación.

7.4. Deep Siamese Multi-scale Fully Convolutional Network

Este modelo, cuyo artículo se encuentra en [Chen et al. \[2020\]](#), toma como referencia varios trabajos existentes, entre ellos los explicados anteriormente. La arquitectura de este modelo (**Deep Siamese multi-scale Fully Convolutional Network (DSMS-FCN)**) es muy similar a las anteriores, solo que estos autores proponen una nuevo **Módulo Convolutivo de Características a Multi-Escala (MFCU)**. Este surge con el objetivo de mejorar la extracción de características espacio-espectrales en imágenes de muy alta resolución (VHR).

Los autores mencionan en su trabajo los problemas que pueden sufrir los módulos convolucionales sobre estas imágenes de gran resolución:

- Solo pueden extraer características espacio-espectrales del mismo tamaño para un único módulo. Es decir, que si se produce un mismo cambio a diferentes escalas, estas Redes no son capaces de detectar este cambio si se produce a gran escala.
- La operación de convolución puede tener efectos de emborronamiento sobre la imagen, lo que se traduce en pérdidas de información cuando ya se ha pasado por varios de estos módulos.

Para solucionar estos problemas, desarrollaron el mencionado módulo MFCU, el cual está compuesto por las 4 operaciones que se visualizan en la Figura [7.7](#), las cuales le permiten extraer estas características espacio-espectrales a diferentes escalas. Tras extraer estas diferentes características, se fusionan en un último paso. Estas operaciones consisten en:

- **Convolución con kernel 1x1**: capaz de extraer características a nivel de píxel.

- **Convolución con kernel 3x3:** que es capaz de extraer las características espacio-espectrales.
- **Convolución con kernel 5x5:** extrae estas características espacio-espectrales a mayor escala, lo que permite hacer un mejor seguimiento de objetos grandes en imágenes VHR.
- **MaxPooling de 3x3:** que consigue extraer aquellas características más destacadas de la imagen evitando el efecto de emborronamiento producido por las capas de convolución.

A parte de estas, se puede ver como se aplican convoluciones de 1x1, esta práctica se realiza para minimizar así el tamaño de la entrada y obtener un algoritmo más eficiente, ya que hay menos entradas y parámetros que considerar.

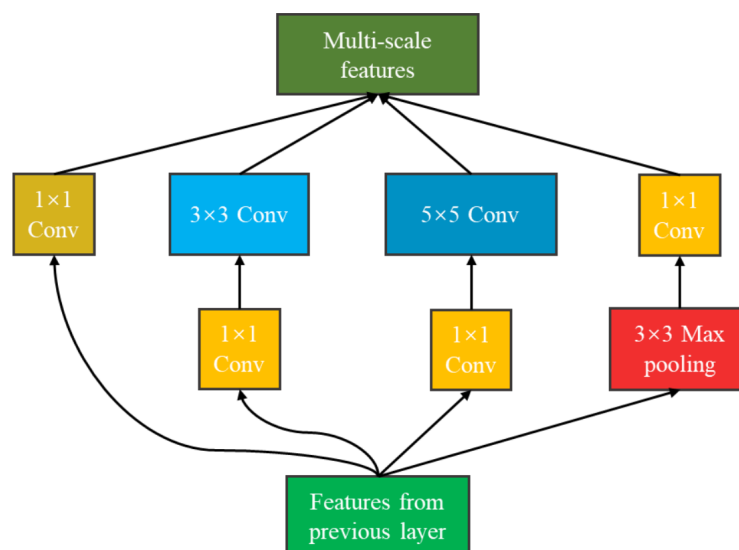


Figura 7.7: Estructura de MFCU, capaz de extraer características espacio-espectrales a múltiples escalas. Extraído de [Chen et al. \[2020\]](#).

Usando este módulo como extractor de características, se construye una red similar a las anteriores, donde se encuentra un *Encoder-Decoder* cuyo primer tramo se encuentra ramificado, es decir, se encuentra una red siamesa convolucional. Al igual que los modelos anteriores, las ramificaciones son usadas para extraer las características de cada una de las imágenes del par, pero una de las principales diferencias es que se pasan las imágenes recortadas en parches de igual tamaño.

En la Figura 7.8 se puede visualizar la arquitectura de esta red, donde se puede ver que las diferentes salidas de las capas del codificador están conectadas con las entradas a las capas paralelas del decodificador, al igual que ocurría con FC-SD, lo

que se concatena es la resta absoluta de la salida de ambas ramas. Como ya se mencionó, estas conexiones sirven para que a la hora de decodificar, se pueda recuperar información sobre la localización de los diferentes elementos, lo que permite que se generen mapas de cambio con bordes más precisos.

Los recuadros verdes marcan las capas convolucionales de 3×3 , los rojos son las capas de MaxPooling de 2×2 , en azul el módulo MFCU y en morado se encuentran las capas convolucionales traspuestas. Las conexiones con líneas negras serían las restas entre las salidas y la concatenación entre las diferentes capas del *Encoder-Decoder*.

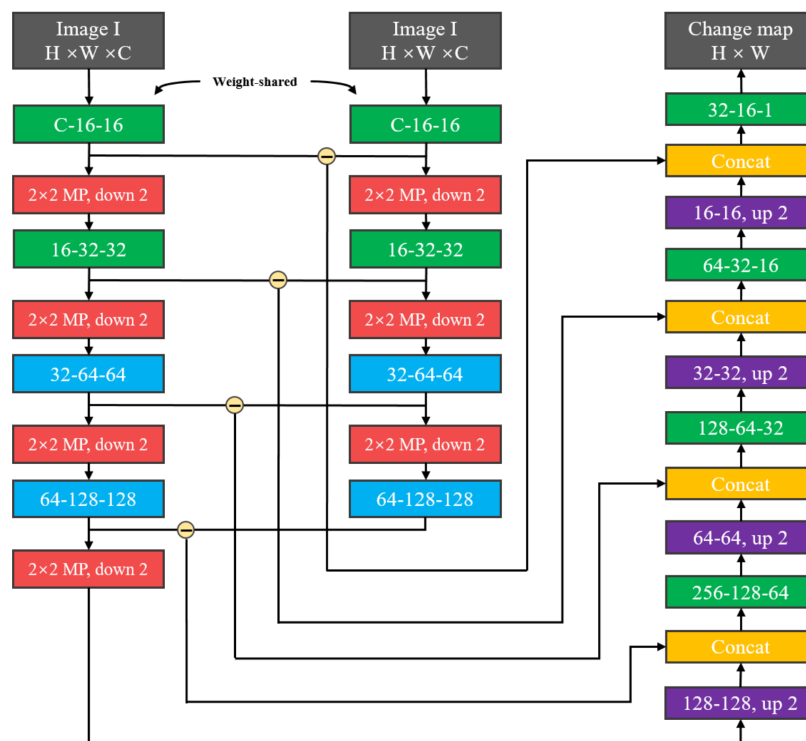


Figura 7.8: Arquitectura de la red DSMS-FCN.

Para la activación de las diferentes capas de convolución y convolución traspuesta se aplica una **ReLU**; sin embargo, al final de la red se aplica una **Sigmoid** para poder calcular la probabilidad de cambio sobre toda la imagen. Como algoritmo de optimización se usará **Adam**, ya que este funciona mejor que el gradiente descendente en algunas circunstancias, ya que este converge mejor. La principal diferencia entre estos es que el SGD aplica una tasa de aprendizaje fija, mientras que Adam la cambia para cada variable utilizando para ello el gradiente y el *momentum*, se puede extraer más información sobre este método en [Kingma and Ba \[2017\]](#).

7.5. Improved UNet++

En el artículo [Peng et al. \[2019\]](#) se muestra de nuevo una arquitectura *Encoder-Decoder*, una mejora de la propuesta UNet++ presentada inicialmente en [Zhou et al. \[2018\]](#) para realizar segmentación sobre imágenes médicas, por lo que este modelo lo que busca es adaptarse al campo de la Detección de Cambios.

La principal diferencia entre *UNet* y *UNet++* son las conexiones simétricas que se encuentran entre las capas de codificador y decodificador. Con el modelo anterior simplemente se concatenaban las salidas de unas con las entradas de otras; pero en esta ocasión, se encuentran módulos convolucionales entre estas conexiones. Se puede ver la estructura en la Figura 7.9, donde se pueden apreciar las diferentes conexiones entre estos nodos, encontrando conexiones desde las salidas del encoder a todos los nodos intermedios pertenecientes al mismo nivel (flechas moradas). Para cada nodo convolucional entonces se procesan todas las salidas de los nodos anteriores, permitiéndole extraer características multi-escalares desde múltiples nodos.

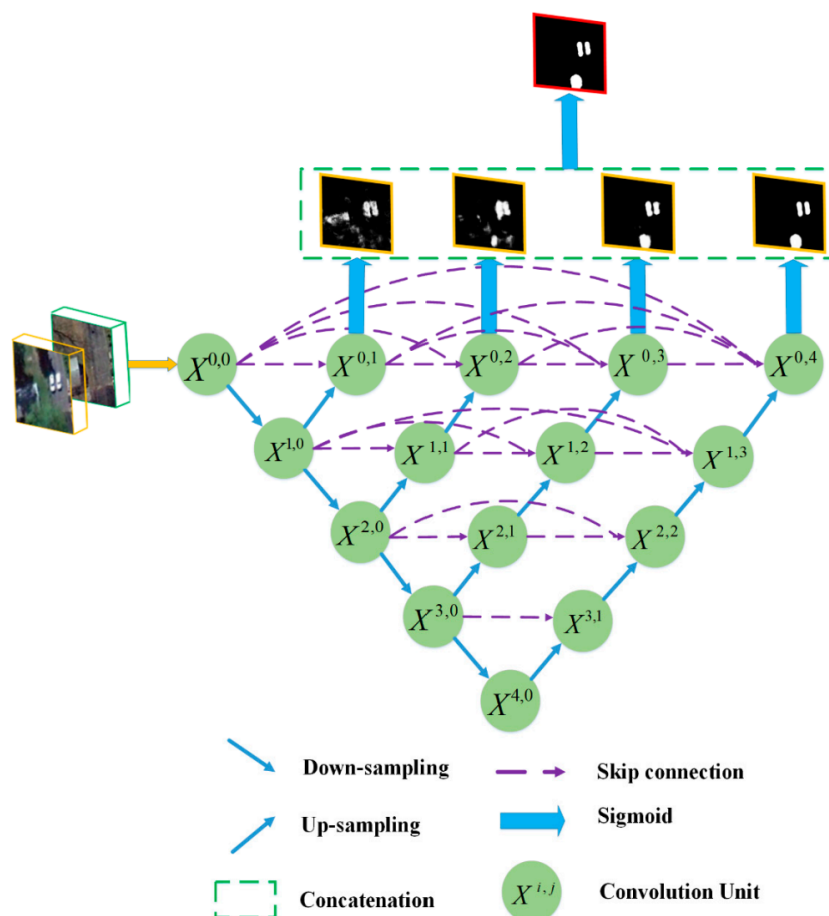


Figura 7.9: Arquitectura de la red UNet++ mejorada.

Estos nodos están compuestos por las operaciones presentes en la Figura 7.10, se encuentran dos bloques formados por una operación de convolución y otro de *Batch Normalization*. Para la salida de este nodo se realiza la suma de la salida de la primera operación de convolución y la salida de la última operación de *Batch Normalization*. Para la activación de estos se aplica una SeLU en lugar de ReLU ya que se obtienen un mayor nivel de regularización, lo que hace el proceso de aprendizaje más robusto.

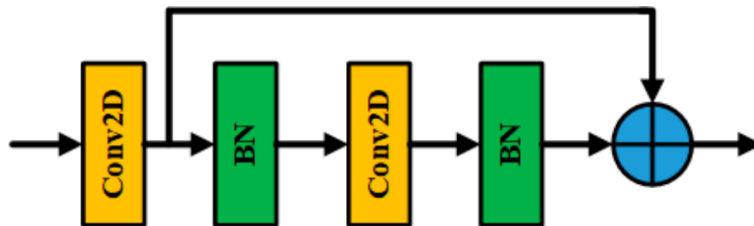


Figura 7.10: Módulo convolucional aplicado en *UNet++*.

Para la Figura 7.9 también se puede observar como se producen 5 salidas para este modelo, una para cada camino posible. De esta manera se obtiene una salida para los caminos:

$$X_{0,0} \rightarrow X_{1,0} \rightarrow X_{2,0} \rightarrow X_{3,0} \rightarrow X_{4,0} \rightarrow X_{3,1} \rightarrow X_{2,2} \rightarrow X_{1,3} \rightarrow X_{0,4} \quad (7.8)$$

$$X_{0,0} \rightarrow X_{1,0} \rightarrow X_{2,0} \rightarrow X_{3,0} \rightarrow X_{2,1} \rightarrow X_{1,2} \rightarrow X_{0,3} \quad (7.9)$$

$$X_{0,0} \rightarrow X_{1,0} \rightarrow X_{2,0} \rightarrow X_{1,1} \rightarrow X_{0,2} \quad (7.10)$$

$$X_{0,0} \rightarrow X_{1,0} \rightarrow X_{0,1} \quad (7.11)$$

Una vez obtenidos estos cuatro mapas de cambios se concatenan y se le aplica una Sigmod para obtener el mapa de cambio final.

Para calcular la pérdida en este modelo se deben de considerar entonces todas las salidas que se producen, por ello se introduce una función de pérdida, que realiza el sumatorio de las pérdidas producidas por cada una de estas. Esta función de pérdida es una combinación de **Binary Cross-Entropy Loss** y **Dice Coefficient Loss**.

$$L_i = L_{bce_i} + \lambda L_{dice_i} \quad (7.12)$$

Donde *bce* indica la pérdida calculada mediante Binary Cross-Entropy Loss, λ es un coeficiente que permite balancear ambas métricas (dar más importancia a una de ellas) y *dice* sería la pérdida calculada mediante Dice Coefficient Loss.

Como método de optimización, se vuelve a recurrir al uso de **Adam**.

Capítulo 8

EXPERIMENTACIÓN Y RESULTADOS

Una vez comentados y explicados los diferentes métodos, tanto tradicionales como aquellos que aplican *Deep Learning*, se deben de ejecutar o entrenar dichos algoritmos para comprobar cual sería su rendimiento en el campo de la Detección de cambios. Uno de los objetivos de este trabajo consistía en el estudio y comparación de los diferentes métodos existentes para observar el impacto de estos en el desarrollo de este campo, por lo que con estas experimentaciones se intentarán extraer unas conclusiones.

8.1. Experimentaciones

Los diferentes resultados de los algoritmos se irán comentando en el mismo orden en el que fueron descritos con anterioridad, por lo que en un primer momento se encontrarán los algoritmos tradicionales que trabajan a nivel de píxel para continuar con diferentes métodos donde se aplica Aprendizaje Profundo.

Como ya se mencionó en el apartado anterior de [5 DATASETS](#), los conjuntos de datos escogidos para el estudio serían:

- **OSCD**: un dataset no muy extenso, pero las máscaras muestran numerosos cambios y las imágenes son tomadas a gran distancia de la superficie terrestre, lo que se pretende con este dataset es demostrar si los diferentes algoritmos funcionan bien ante este tipo de imágenes en la que no aparecen tantos detalles que se pueden ver a pie de calle como coches o árboles. Este dataset se explica más extensamente en el sección [5.1.3 OSCD](#)

- **LEVIR**: bastante más extenso que el anterior, en este caso los pares están tomados desde una posición más cercana, pudiendo observar en estos más detalles en la vegetación, caminos, carreteras, casas, etc. El objetivo de usar este dataset es estudiar el funcionamiento de estos algoritmos con todos estos detalles, ver si son resistentes ante la aparición de sombras o de otros elementos que realmente no deberían influir en la detección de cambios. Para más detalles de este dataset, ver el apartado [5.1.6 LEVIR-CD](#)
- **CROP_FIELD**: en el apartado de Datasets se comentaba la problemática de las máscaras, y es que la mayoría de los conjuntos de datos solo las incluyen mostrando cambios para elementos modificados por el hombre, es decir, construcciones. Sin embargo otro aspecto interesante en este campo sería la detección de modificaciones en el terreno, en campos de cultivo, desplazamientos de tierra, etc. Para hacer un estudio sobre este tipo de cambios se necesita un dataset que muestre en sus máscaras todos los cambios producidos, por lo que se desarrolló este dataset. Consultar el apartado [5.3.4 Crear un dataset sintético](#) para más información sobre el desarrollo de este dataset.
- **CROP_FIELD_2**: para realizar los experimentos también se ha recurrido a otro dataset sintético que ha sido proporcionado por el grupo de investigación de SIMIDAT. Toma como base el mismo que se usaba anteriormente. En él se incorporan artefactos a una imagen de referencia extraídos de otra. Estos parches tiene diferentes formas geométricas y localizaciones aleatorias en los diferentes pares. Al igual que con el dataset anterior, a la segunda imagen del par se le aplica una corrección en la iluminación para que varíe con respecto a la original.

En la Tabla [8.1](#) se resume el tamaño de los diferentes conjuntos aplicados, tales como el conjunto de testeo, el de validación y el de entrenamiento usados para cada dataset.

	Entrenamiento	Validación	Testeo	TOTAL
OSCD	13	3	8	24
LEVIR	445	64	192	701
CROP_FIELD	135	4	24	163
CROP_FIELD_2	150	3	22	175

Tabla. 8.1: Tamaños de los conjuntos de entrenamiento, testeo y validación de los diferentes datasets.

Se puede observar como OSCD es de los que cuenta con una menor cantidad de imágenes, lo cual como se verá más adelante, repercutirá en los resultados de este, a lo que hay que sumarle la naturaleza de estas imágenes.

8.1.1. Experimentaciones con Image Differencing

Los resultados obtenidos con estos métodos no son del todo destacables. Se debe considerar que este método de detección de cambios es muy sensible a los colores ya que trabaja directamente con las intensidades de los píxeles, sin tener en cuenta por ejemplo la vecindad o la variación en la iluminación. Por estas mismas razones son muy sensibles a cualquier tipo de variación, generando mucho ruido, es decir, marcan como cambio muchos píxeles que realmente tienen una ligera variación en el color. Esto resultará un problema sobre todo cuando se encuentren grandes variaciones en la iluminación, sombras, etc.

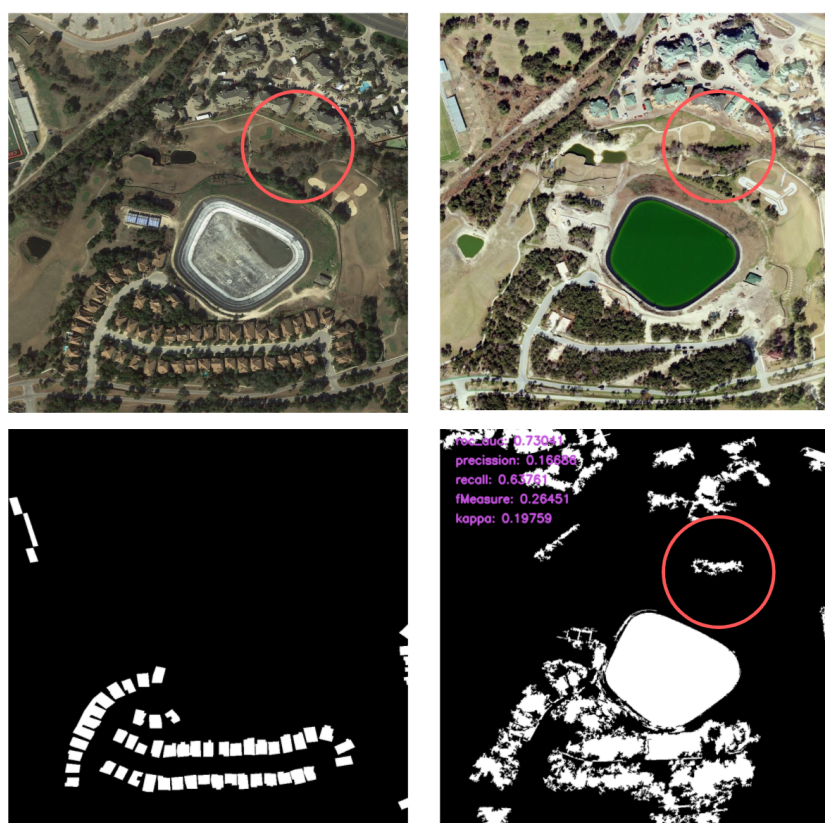


Figura 8.1: Resultado del algoritmo en el dataset LEVIR con la imagen de prueba nº19, en la parte superior se encuentran los dos pares comparados en diferentes instantes; en la parte inferior se encuentra la máscara cedida por el propio autor (izq) y la máscara obtenida por el algoritmo (dcha).

A la hora de revisar los resultados, sólo se pueden considerar aquellos que aparecen señalados en las máscaras originales para poder aplicar cualquier tipo de métrica. Es por ello que los resultados no son del todo fiables, ya que si por ejemplo se mira la Figura 8.1 se puede ver como hay muchos de ellos que no se marcan, como el embalse, y que se ven a simple vista. También se pueden observar fallos, como se había mencionado, en las zonas donde se producen sombras, este error se encuentra

marcado en rojo. A pesar de todos estos problemas, la máscara puede resultar bastante acertada, si se consideran también los cambios que no han sido marcados en la máscara original.

	Método	AUC	Precision	Recall	F-Measure	Kappa
OSCD	Img_diff	0,53810	0,06100	0,60587	0,10027	0,03290
	Compare_ssim	0,73477	0,24077	0,57241	0,30931	0,27678
LEVIR	Img_diff	0,52903	0,06027	0,18823	0,06366	0,01401
	Compare_ssim	0,50011	0,03948	0,06210	0,02087	-0,00696
Crop_field	Img_diff	0,73786	0,84811	0,49179	0,59706	0,53743
	Compare_ssim	0,82796	0,79562	0,70710	0,73868	0,68238
3TB_2	Img_diff	0,57590	0,13854	0,70835	0,16828	0,12374
	Compare_ssim	0,92654	0,69997	0,88851	0,75077	0,74236

Tabla. 8.2: Resultados de los algoritmos en sus dos variantes.

Como se puede observar en la Tabla 8.2 los resultados varían mucho con respecto al dataset utilizado y método. En cuanto a los datasets, esto se puede deber a la naturaleza de las imágenes que contiene cada uno. OSCD muestra imágenes más alejadas de la superficie, por lo que es difícil que se vean las sombras o diferentes elementos tales como personas, coches y demás detalles; mientras que LEVIR tiene pares tomados desde una perspectiva más cercana, por lo que estos detalles como pequeñas construcciones, árboles o sombras van a marcar mucho la ejecución de estos algoritmos a nivel de píxel. Los datasets sintéticos obtienen resultados bastante destacables con respecto a los demás, esto se puede deber a que los parches aplicados a las imágenes son fáciles de identificar por este algoritmo ya que estos a veces resultan demasiado “llamativos”, es decir, la variación de la intensidad de los píxeles es muy alta. Como ya se había comentado anteriormente, este algoritmo es muy sensible a estos cambios.

Por lo general se puede ver un mejor funcionamiento de estos algoritmos en el dataset de CROP_FIELD_2 por la problemática comentada anteriormente. En el caso de OSCD, se obtiene el mejor resultado para la segunda opción de este algoritmo, al ser imágenes más distantes, puede centrarse en los cambios producidos por modificaciones en las estructuras y terrenos. Aun así no se obtiene una precisión mayor al 30 %, es decir, hay demasiados falsos positivos si se refiere a las máscaras de los autores.

Un ejemplo del buen funcionamiento de estos métodos se encuentra en la Figura 8.2, donde el resultado del algoritmo marca con precisión (para compare_ssim) los cambios que aparecen en la máscara de los autores.

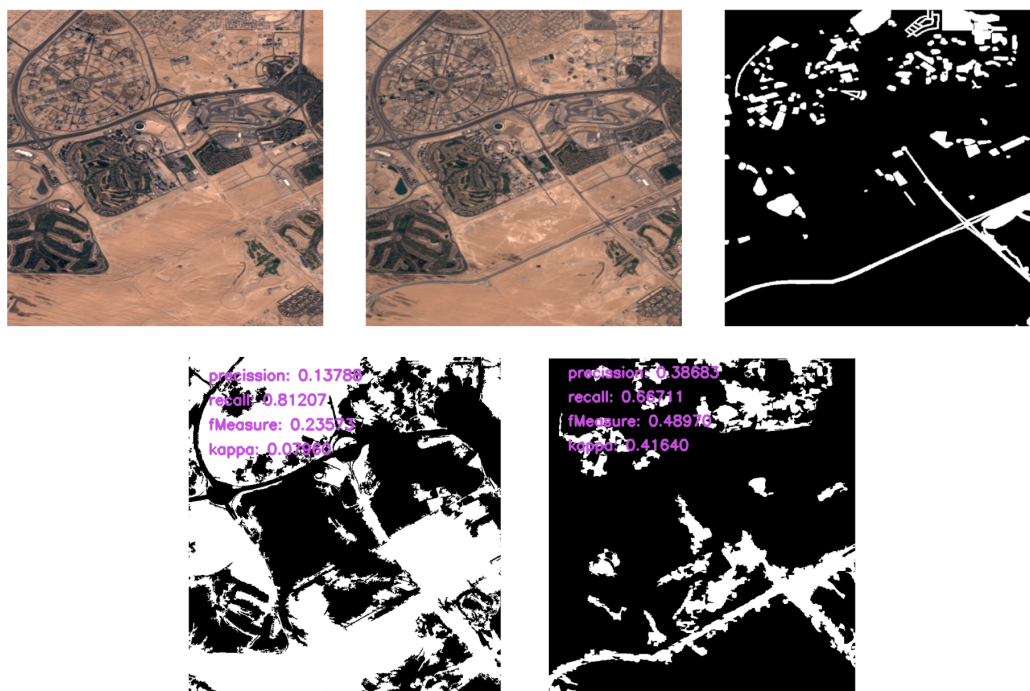


Figura 8.2: Muestra del funcionamiento de estos algoritmos sobre OSCD, en la parte inferior se observa el resultado y métricas obtenidas para `img_diff` (izq) y `compare_ssim` (dcha).

Tras ver que los métodos anteriores no proporcionaban muy buenos resultados, se propuso una posible solución para mejorar el funcionamiento del algoritmo. Con el objetivo de atenuar los efectos de las sombras y los colores, se opta por una umbralización adaptativa para conseguir así centrarse en los detalles de la imagen, más que en su color o sombras (ya que como se ve en la Figura 6.5, desaparecen totalmente).

A pesar de esta modificación del algoritmo, si se observa la Tabla 8.3, los resultados empeoran, pero solo en unas cuantas décimas y sobretodo para OSCD, donde no se supera en ningún momento el 20 % para Kappa cuando para `compare_ssim` anteriormente se alcanzaba el 27 %. Para el dataset de `CROP_FIELD` y `CROP_FIELD_2` no se aprecia ninguna mejoría destacable, llegando incluso a comportarse peor que el caso anterior. Vistos los resultados, se puede concluir que el algoritmo no ha cumplido con las expectativas esperadas.

Al analizar los diferentes resultados, se puede concluir que este método no es muy eficaz para el campo de la Detección de cambios, no llegando a más del 0.24 en precisión, para datasets no sintéticos, y de 0.84 para los sintéticos, siendo ambos los mejores resultados. Si se estudian otras métricas se puede ver como el rendimiento de estos es incluso peor, obteniendo para Kappa un máximo de 0.27 y 0.74, de nuevo en el mejor de los casos.

	Adaptative	AUC	Precision	Recall	F-Measure	Kappa
OSCD	img_diff (11x11)	0,56868	0,31418	0,16532	0,20254	0,18114
	compare_ssim (11x11)	0,68825	0,10226	0,76880	0,16686	0,11147
	img_diff (31x31)	0,56819	0,26943	0,17733	0,18559	0,16123
	compare_ssim (31x31)	0,67381	0,10608	0,68971	0,17055	0,11646
	img_diff (101x101)	0,56334	0,24656	0,17861	0,17333	0,14732
	compare_ssim (101x101)	0,66688	0,09694	0,72009	0,15912	0,10254
LEVIR	img_diff (11x11)	0,52253	0,03109	0,11483	0,02325	-0,01491
	compare_ssim (11x11)	0,53982	0,05119	0,36344	0,07653	0,00530
	img_diff (31x31)	0,50708	0,02804	0,20193	0,03149	-0,02845
	compare_ssim (31x31)	0,52628	0,04660	0,21143	0,05577	-0,00167
	img_diff (101x101)	0,46302	0,02802	0,31566	0,04066	-0,03569
	compare_ssim (101x101)	0,51646	0,03711	0,10450	0,03032	-0,00851
Crop_field	img_diff (11x11)	0,61002	0,83068	0,23200	0,35693	0,29531
	compare_ssim (11x11)	0,79260	0,65675	0,68451	0,65598	0,56677
	img_diff (31x31)	0,69978	0,76782	0,43688	0,54331	0,46255
	compare_ssim (31x31)	0,81423	0,58684	0,78252	0,66565	0,55799
	img_diff (101x101)	0,72402	0,66684	0,52097	0,56760	0,47348
	compare_ssim (101x101)	0,80372	0,48637	0,84238	0,61208	0,47272
3TB_2	img_diff (11x11)	0,70494	0,92669	0,41033	0,54706	0,54089
	compare_ssim (11x11)	0,94076	0,53084	0,94610	0,63284	0,61827
	img_diff (31x31)	0,74793	0,87670	0,49728	0,63262	0,62573
	compare_ssim (31x31)	0,95441	0,62571	0,95294	0,71065	0,70027
	img_diff (101x101)	0,73483	0,73715	0,47432	0,56888	0,55981
	compare_ssim (101x101)	0,96765	0,55212	0,96776	0,66909	0,65808

Tabla. 8.3: Se muestra el resultado de los algoritmos aplicando diferente tamaño en los bloques de la umbralización adaptativa.

Se debe considerar la problemática comentada para los datasets sintéticos, y el porqué dichos resultados no deberían de considerarse demasiado, ya que los cambios generados son fácilmente detectables por este algoritmo.

8.1.2. Experimentaciones con Canny

Como se vio en el apartado correspondiente de Canny ([6.2 Canny](#)), se puede observar como se tienen numerosos parámetros que ajustar, tales como los umbrales para el paso de la histéresis o el tamaño del *kernel* aplicado posteriormente para producir un emborronado en la imagen. Se ha realizado una experimentación previa con diferentes parámetros del algoritmo, pero el estudio realizado muestra que los resultados obtenidos no son nada parecidos a los del algoritmo anterior, llegando incluso a ser peores (se pueden comprobar dichos resultados en la [Tabla 8.5](#)). Tras las diferentes pruebas, se extrajeron unos parámetros comunes a todos los datasets, los cuales ofrecían los mejores resultados, se pueden observar los parámetros usados en

la Tabla 8.4.

Umbral Superior	90
Umbral Inferior	225
Tamaño Filtro Emborronamiento	2x2

Tabla. 8.4: Parámetros usados para la ejecución de Canny.

Para el dataset de OSCD no se obtienen buenos resultados, tal y como se puede observar en la Figura 8.3, donde la salida del algoritmo no tiene nada que ver con la máscara cedida por los autores (solo marca la carretera que aparece en la parte inferior de esta), ni marca ningún cambio que se pueda a ver a simple vista.

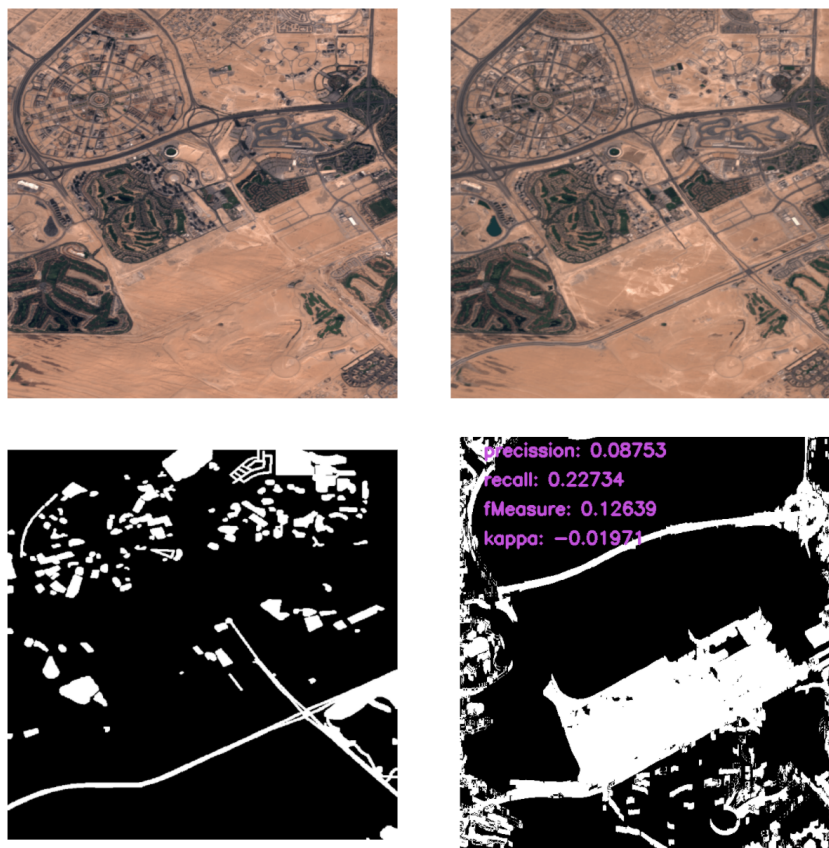


Figura 8.3: Ejemplo de funcionamiento de Canny sobre el dataset de OSCD.

En el caso del conjunto de datos de LEVIR se puede observar como este algoritmo presenta problemas con la vegetación, ya que las sombras que producen en conjunto los árboles, o las sombras de las propias hojas... Producen bordes cuando realmente no los hay, en este sentido Canny es muy sensible y aunque se cambien los valores de umbralización los resultados obtenidos no son mucho mejores. Se puede ver en la Figura 8.4 como se producen grandes cantidades de ruido tras haber aplicado Canny, el proceso de emborronamiento y tras haber buscado y rellenado los diferentes contornos. Este ruido no solo aparece en las zonas boscosas, sino en cualquier superficie

que no sea regular como por ejemplo el agua del embalse, sinodo esto un indicio del mal funcionamiento del algoritmo.

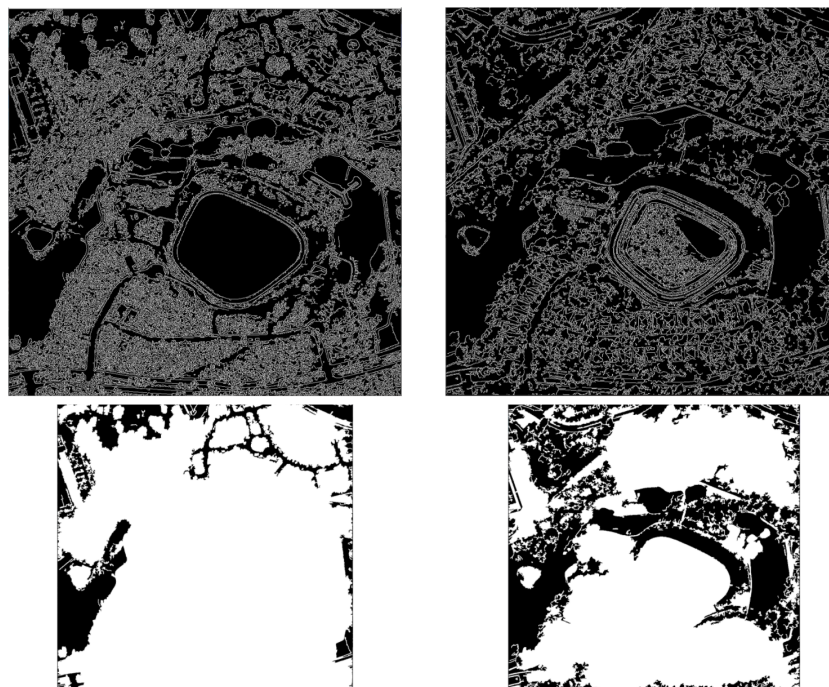


Figura 8.4: Aquí se muestran unos de los pasos intermedios del algoritmo, en la primera fila se mostrará el resultado de Canny sobre la imagen original, mientras que en la parte inferior se muestran los contornos ya rellenados.

Como se ha podido ver en la figura anterior el comportamiento no es del todo adecuado, por lo que los resultados obtenidos no serán muy destacables sobre este dataset, se puede ver el resultado de dicho ejemplo en la Figura 8.5

Con el dataset de CROP_FIELD se ve un comportamiento similar al anterior, tal y como se refleja en la Figura 8.6 se producen grandes fallos.

Si se observa la tabla de resultados (ver 8.5) se pueden ver los diferentes resultados obtenidos con Kappa, obteniendo en el mejor de los casos un 0.24 para el dataset de CROP_FIELD, el siguiente “mejor” resultado es encontrado para el dataset de OSCD con 0.04. Si se observa la precisión, en el caso de OSCD, CROP_FIELD y CROP_FIELD_2 no se obtiene más de un 0.06, unos resultados pésimos comparados con el 0.85 encontrado para el algoritmo anterior. Con los resultados en LEVIR, los cuales son negativos para Kappa, se puede observar como la vegetación y los diferentes problemas comentados han causado bastante confusión en este algoritmo.

Finalmente se ha querido añadir una gráfica donde se visualizan mejor los resultados de este algoritmo para poder hacer una comparativa visual de los diferentes conjuntos de datos. Ver Figura 8.7.

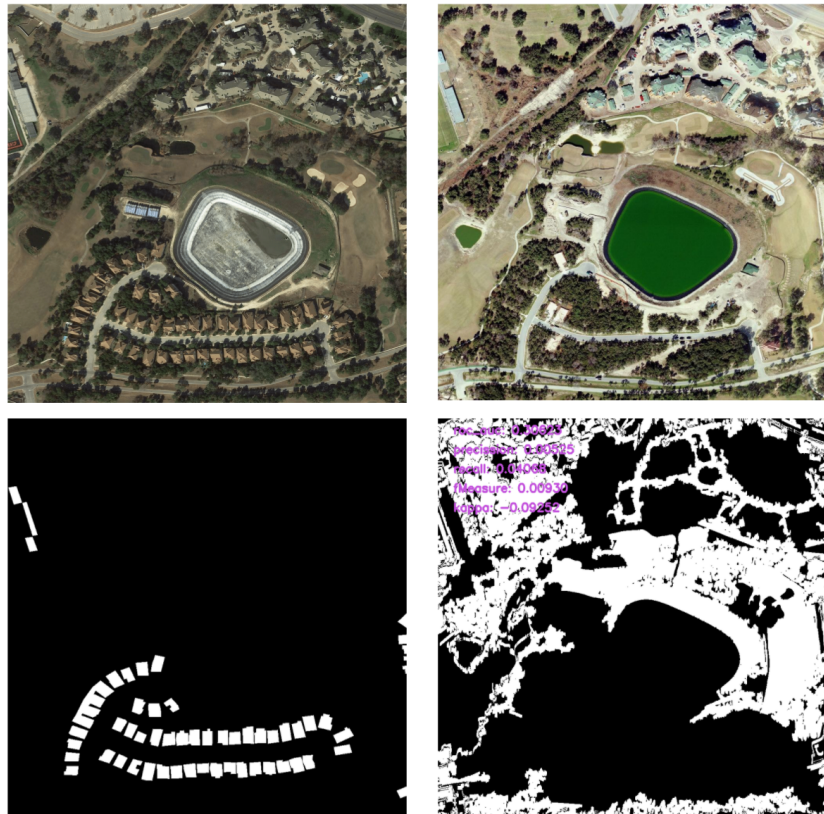


Figura 8.5: Ejemplo de funcionamiento de Canny sobre un par de LEVIR.

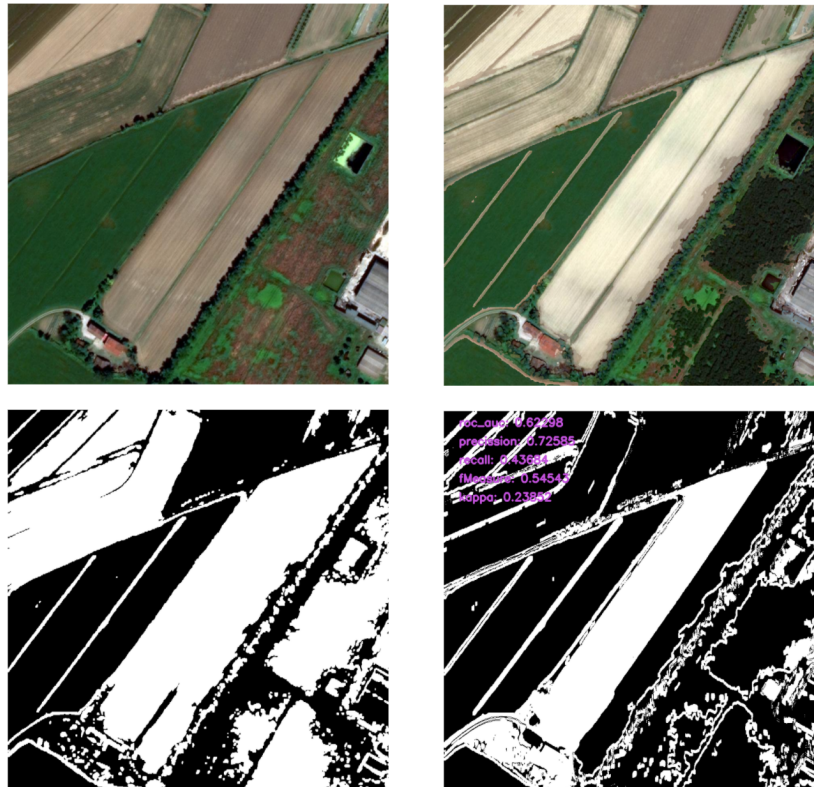


Figura 8.6: Ejemplo de funcionamiento de Canny sobre CROP_FIELD

	AUC	Precision	Recall	F-Measure	Kappa
OSCD	0,57929	0,06281	0,35784	0,09255	0,03796
LEVIR	0,50239	0,04161	0,39662	0,06637	-0,00929
Crop_Field	0,65078	0,38439	0,49700	0,40697	0,24800
CROP_FIELD_2	0,49887	0,02720	0,17874	0,04149	-0,00127

Tabla. 8.5: Tabla con los resultados en las diferentes métricas del algoritmo de Canny

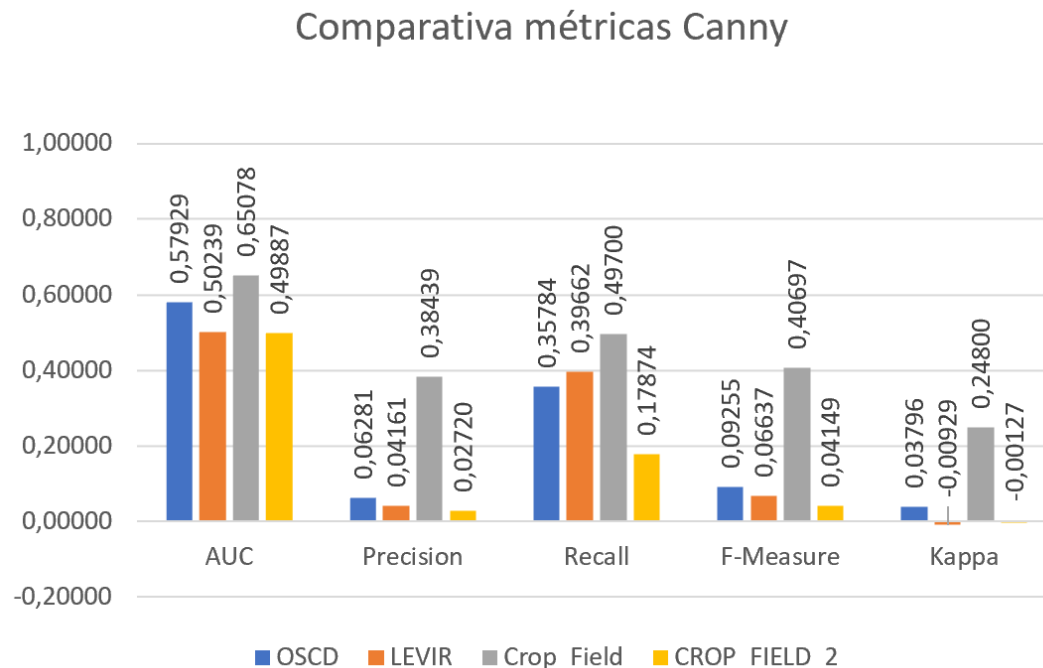


Figura 8.7: Gráfica con los resultados de Canny sobre los diferentes datasets.

8.1.3. Experimentaciones con PCA K-means

Con la adaptación propia realizada sobre el código desarrollado por Abhijeet Kumar (ver el Apartado 6.4.3), se realizó una ejecución con los diferentes datasets. El coste computacional de este algoritmo es bastante más elevado, lo cual influye en el tiempo de procesamiento el cual crece notoriamente con respecto a los anteriores.

	AUC	Precision	Recall	F-Measure	Kappa
OSCD	0,67308	0,28878	0,41639	0,28285	0,25639
LEVIR	0,57385	0,08355	0,30607	0,10842	0,05045
Crop_Field	0,68559	0,90795	0,38354	0,50634	0,44542
CROP_FIELD_2	0,76422	0,57698	0,57428	0,52475	0,50556

Tabla. 8.6: Resultados del algoritmo de PCA K-means aplicado a los diferentes datasets

Los resultados son algo mejores (ver Tabla 8.6) en todas las métricas si se compara con los de **Image Differencing** y **Canny** para los datasets no sintéticos. Se ve

como el algoritmo de PCA K-means produce bastantes fallos en el dataset de LEVIR, pudiendo deberse a que las máscaras están incompletas o que haya detectado demasiado ruido. Se puede estudiar un ejemplo de este en la Figura 8.8, pudiendo encontrar una explicación al bajo rendimiento de este algoritmo sobre este dataset. Se ve como las sombras de la vegetación confunden al algoritmo, generando mucho ruido en la máscara obtenida, por lo que al final la precisión sobre este dataset se queda en un 0.08, una mejora insignificante en comparación con el 0.06 - 0.04 obtenido con Image Differencing y Canny respectivamente.

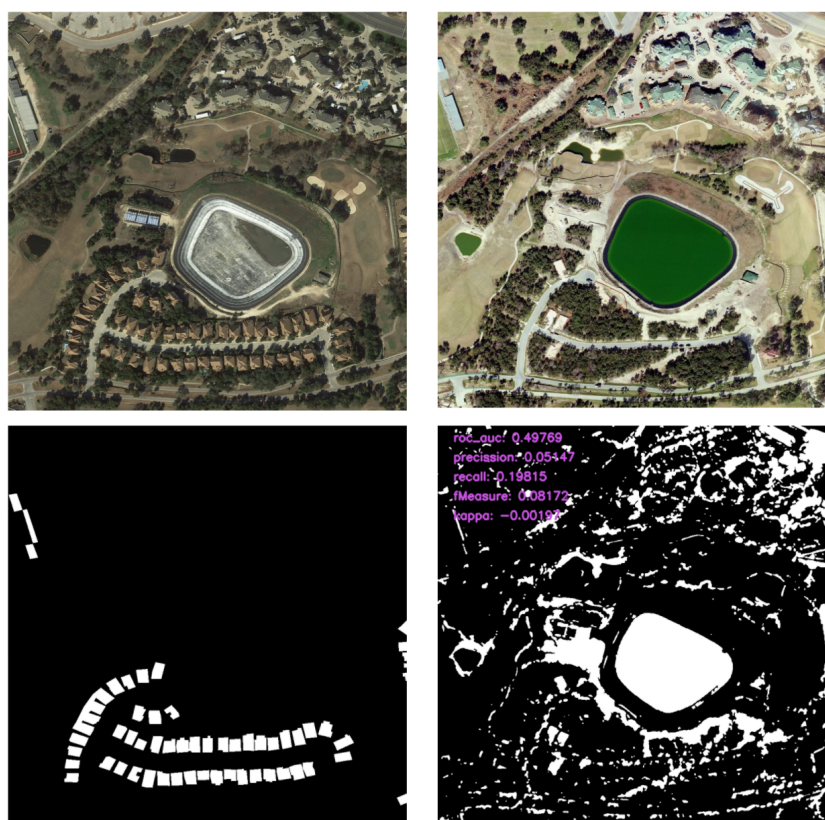


Figura 8.8: Funcionamiento de PCA sobre el dataset de LEVIR.

Si se observa ahora la Figura 8.9 se puede apreciar como también se encuentran fallos, pero estos no se deben a que se marquen cambios fuera de los señalados por la máscara original, sino porque no se terminan de marcar todos estos. A pesar de ello, se puede ver como se marcan los cambios donde realmente deberían de estar, y por ello se obtiene una precisión del 50 % para esta imagen, una puntuación bastante aceptable considerando resultados anteriores.

En este caso para los datasets sintéticos no se obtienen los resultados tan destacables que se encontraban para **Image Differencing**. Se puede ver un ejemplo de funcionamiento sobre ambos en la Figura 8.10, en este caso para CROP_FIELD ha funcionado correctamente, obteniendo una precisión de 0.99; mientras que para

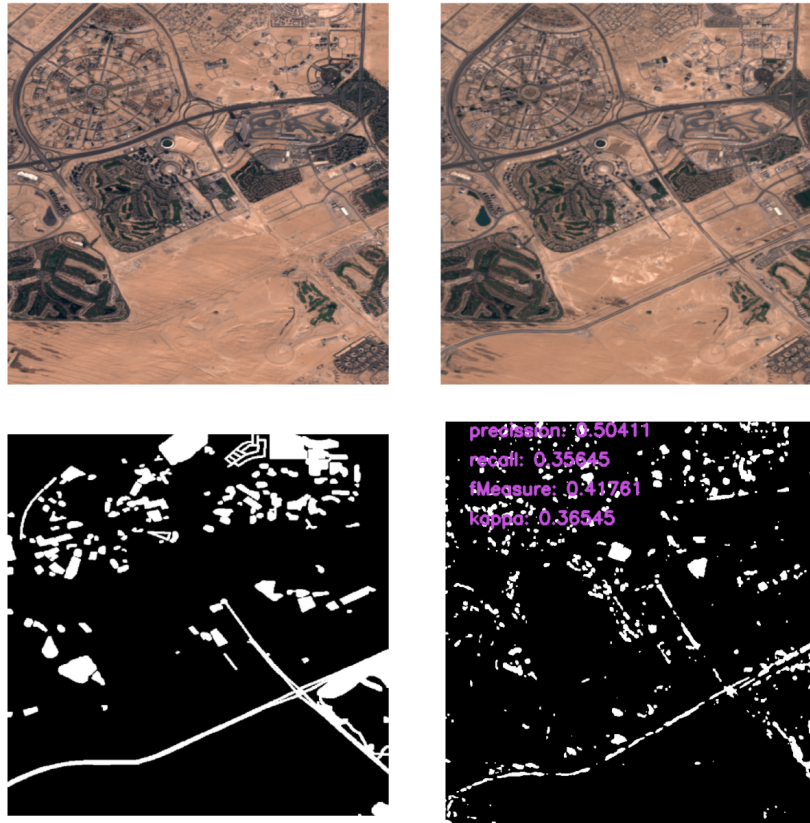


Figura 8.9: Funcionamiento de PCA sobre el dataset de OSCD.

CROP_FIELD_2 se puede remarcar el problema de este algoritmo con la vegetación, mostrando demasiadas sombras como ruido y por consiguiente obteniendo una precisión de 0.07.

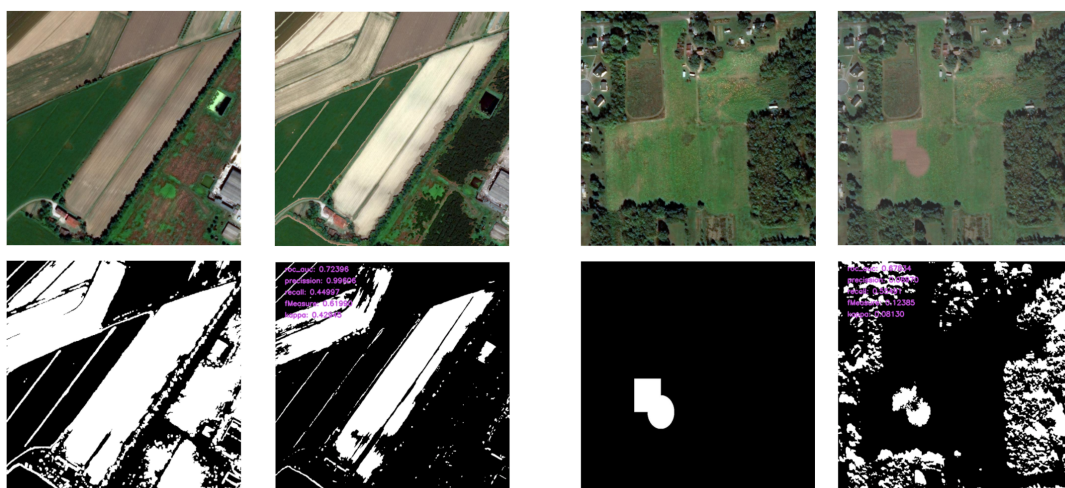


Figura 8.10: Funcionamiento de PCA sobre el dataset de CROP_FIELD (izq) y CROP_FIELD_2 (dcha).

En la Figura 8.11 se puede ver la recopilación gráfica de las diferentes métricas. Por

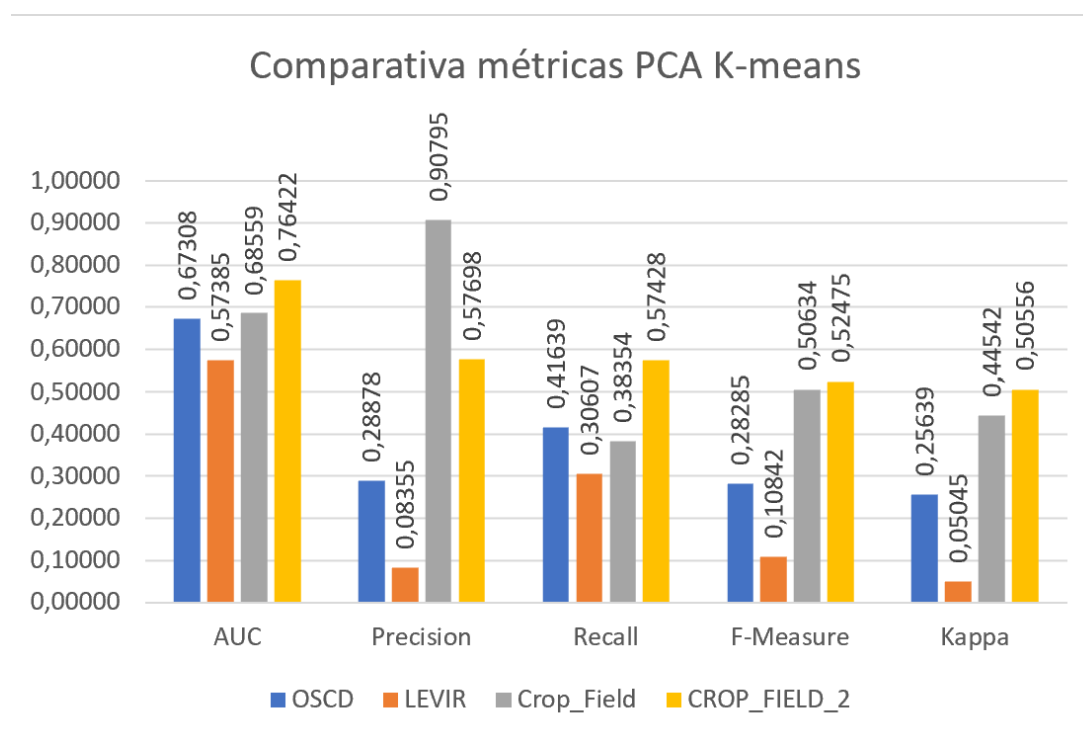


Figura 8.11: Presentación visual de los resultados de PCA K-means.

lo general parece que este algoritmo funciona mejor que los anteriores, esto se puede deber a que, a pesar de trabajar a nivel de píxel, si se tiene en cuenta la vecindad para marcar si un píxel ha sufrido un cambio o no.

8.1.4. Experimentaciones con Fully Convolutional siamese networks for change detection

Como ya se adelantó en apartados anteriores, a la hora de trabajar con Redes Neuronales se deben tener en cuenta uno de los pasos clave, se debe entrenar la red con los datos de entrenamiento para que esta sea capaz de aprender la relación entre los datos y los patrones. A la hora de entrenar estas redes, se almacena el modelo del epoch que haya obtenido unos mejores resultados con respecto al anterior, por lo que de los 250 epoch durante los cuales se van a entrenar las diferentes redes, se almacenará el mejor modelo encontrado. Para decidir que modelo es mejor, se observa la pérdida producida durante ese periodo de entrenamiento.

En la tabla 8.7 se pueden observar los diferentes resultados obtenidos para los diferentes modelos descritos en el apartado de 7.3 Fully Convolutional siamese networks for change detection. Se puede comprobar como los resultados mejoran en gran medida con respecto a los métodos tradicionales anteriores, donde para CROP_FIELD, el

mayor valor de Kappa obtenido era 0.68 con **ImgDiff2**, mientras que con este algoritmo no se descienden de los 0.96. Con LEVIR se obtenían unos resultados por debajo de 0.04 para todos los algoritmos, y como se puede observar, el resultado es 15 veces mejor con respecto al resultado más bajo obtenido por estos modelos.

	Modelo	AUC	Precision	Recall	F-Measure	Kappa
OSCD	FC-EF (117)	0,57720	0,67852	0,16324	0,21570	0,20098
	FC-SC (150)	0,65678	0,48860	0,36529	0,34435	0,31673
	FC-SD (184)	0,63126	0,55850	0,29466	0,32956	0,30487
LEVIR	FC-EF (220)	0,83499	0,75469	0,67438	0,70214	0,69537
	FC-SC (101)	0,84638	0,76126	0,69775	0,71910	0,71237
	FC-SD (167)	0,86178	0,76730	0,72824	0,74047	0,73455
CROP	FC-EF (210)	0,98321	0,96900	0,97564	0,97222	0,96459
	FC-SC (249)	0,97998	0,98811	0,96408	0,97585	0,96893
	FC-SD (248)	0,99111	0,98591	0,98574	0,98581	0,98225
CROP_FIELD_2	FC-EF (230)	0,98337	0,98976	0,96705	0,97772	0,97722
	FC-SC (64)	0,99752	0,94703	0,99644	0,97082	0,97007
	FC-SD (230)	0,98337	0,98976	0,96705	0,97772	0,97722

Tabla. 8.7: Resultados de los diferentes modelos aplicados. Entre paréntesis se encuentran los mejores epochs.

Un detalle a destacar es que para OSCD no se obtienen muy buenos resultados, de hecho no mejora mucho con respecto a los 0.28 obtenidos con **PCA K-means**, esto se puede deber a la escasa información espacial que tiene el dataset. Al estar las imágenes tomadas a tanta escala y no ser de gran calidad, hay muchos datos que se pierden y confunden a la red para que no funcione correctamente. Se puede ver en la Figura 8.12 como el algoritmo tiene bastantes dificultades para detectar los diferentes cambios.

En la Figura 8.13 se resumen las diferentes puntuaciones de Kappa para los diferentes conjuntos de datos, como ya ocurría anteriormente, los dataset sintéticos ofrecen resultados demasiados buenos. Se puede remarcar la problemática de aplicar estos datasets, añadir estos parches resulta demasiado llamativo e intuitivo, por lo que, estos algoritmos no encuentran ninguna dificultad a la hora de detectar los bordes de estos artefactos.

También se puede ver, como para estos datasets en concreto, las redes de FC-SC y FC-SD tienen resultados muy similares, aunque FC-EF tampoco es muy dispar a estos. En el artículo de donde se extrajeron estas arquitecturas, se menciona que la red de FC-SD obtiene un mayor rendimiento que el resto, lo cual realmente se cumple en cierta medida.

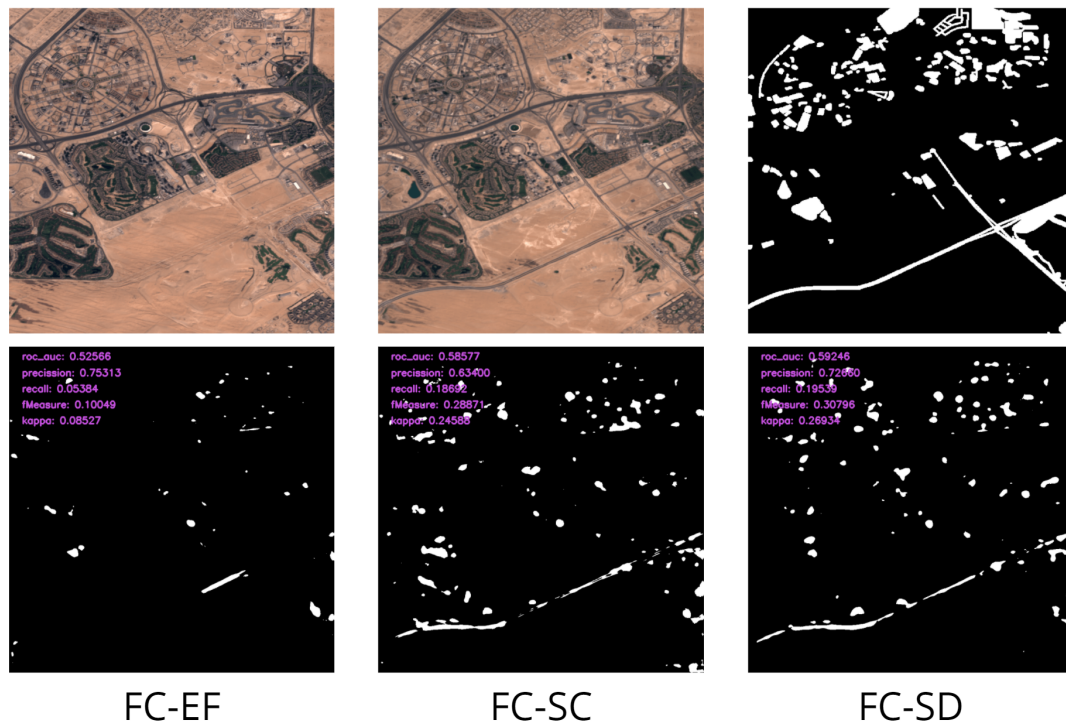


Figura 8.12: Resultados obtenidos de los diferentes modelos sobre el dataset de OSCD.

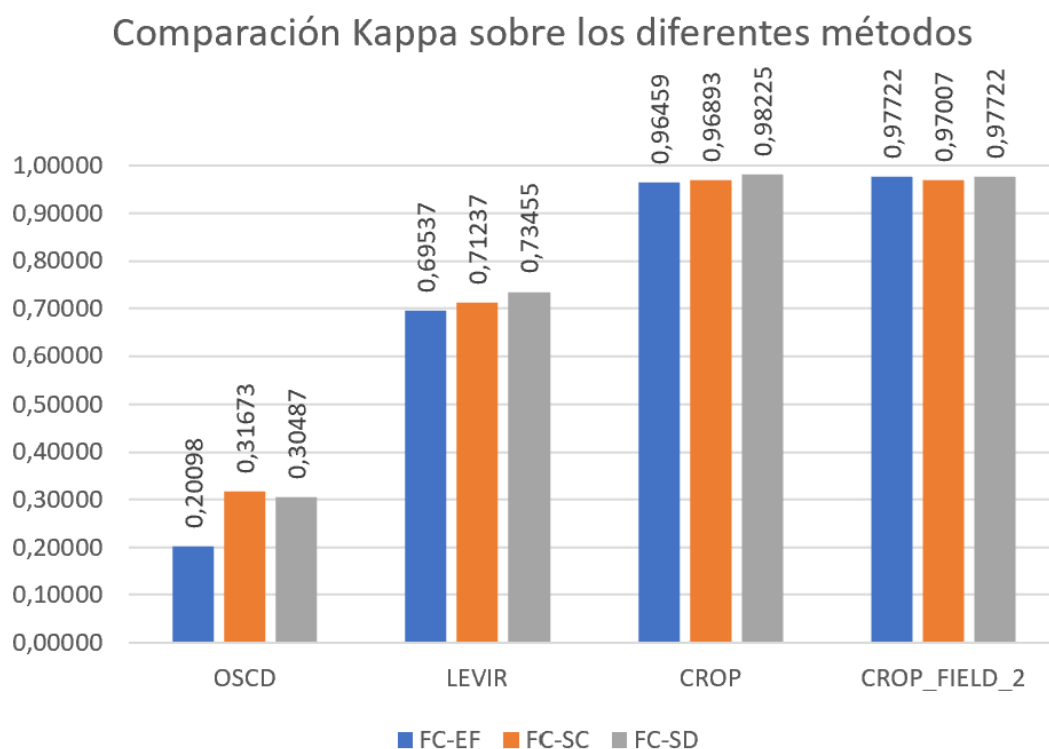


Figura 8.13: Presentación visual de los resultados de Kappa de los diferentes modelos.

8.1.5. Experimentaciones con DSMS-FCN

Al igual que con la experimentación anterior, se realiza el entrenamiento con 250 epochs, almacenando siempre el mejor modelo encontrado, aquel que ofrezca una menor pérdida.

En la Tabla 8.8 se pueden apreciar los resultados de este algoritmo. Lo más destacable de estos resultados es la mejora con el dataset de OSCD, ya que antes no se obtenía más de 0.31 para Kappa, mientras que con este método se consigue un 0.38, la mejora no es mucha, pero si se compara de nuevo con los métodos tradicionales se puede observar una mejoría del 13 %. Teniendo en cuenta los diferentes problemas a los que se enfrentan los algoritmos con este dataset, los resultados son destacables.

	AUC	Precision	Recall	F-Measure	Kappa
OSCD (141)	0,71256	0,59018	0,49396	0,40940	0,38105
LEVIR (143)	0,87090	0,75222	0,74750	0,74263	0,73683
Crop_Field (220)	0,98288	0,96568	0,97483	0,96984	0,96198
CROP_FIELD_2 (130)	0,97079	0,91037	0,94269	0,92609	0,92539

Tabla. 8.8: Resultados DSMS-FCN sobre los datasets. Entre paréntesis se encuentra el número de epoch del cual se ha extraído el modelo.

Con el resto de datasets no parece haber ninguna mejora, obteniendo mejores resultados con el modelo FC-SD anterior. En el caso de LEVIR las métricas difieren en unas pocas décimas, mejorando para este algoritmo, mientras que con CROP_FIELD se empeora en un 2 %. Profundizando un poco más, la medida de Kappa obtenida para CROP_FIELD_2 de 0.92 es inferior a la mostrada anteriormente de 0.97.

Se puede observar una muestra del resultado de este algoritmo sobre todos los datasets en la Figura 8.14. Se puede ver como en general, las máscaras generadas son bastante acertadas con respecto a las máscaras de los autores.

En la Figura 8.15 se puede realizar una revisión gráfica sobre las métricas obtenidas para los diferentes datasets. Se puede concluir que los mejores resultados son para CROP_FIELD en todos los casos, seguido por CROP_FIELD_2. Estos resultados de nuevo se pueden deber a que los parches añadidos sintéticamente pueden resultar demasiado llamativos para el algoritmo, por lo que es fácil para el localizar los bordes de estos y marcarlos.

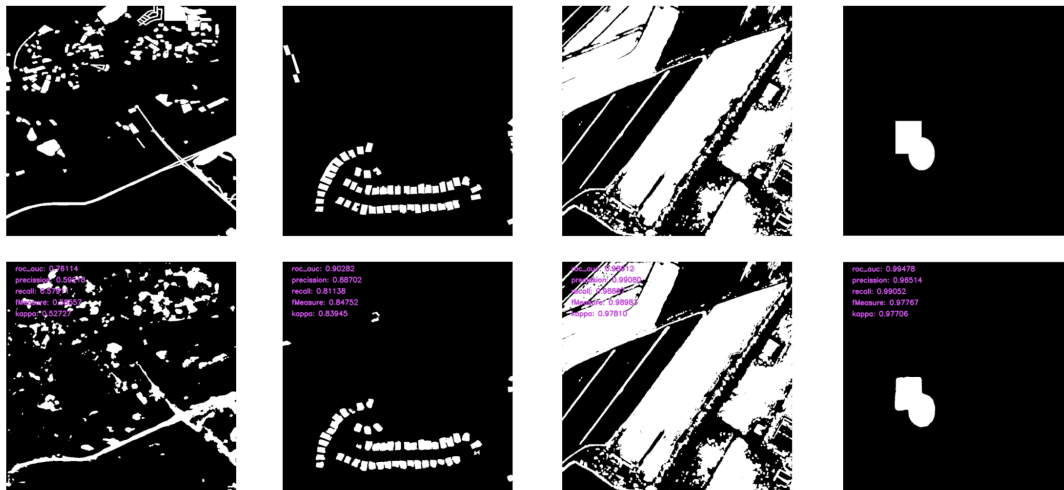


Figura 8.14: En orden, resultados sobre los datasets de OSCD, LEVIR, CROP_FIELD y CROP_FIELD_2 con DSMS-FCN.

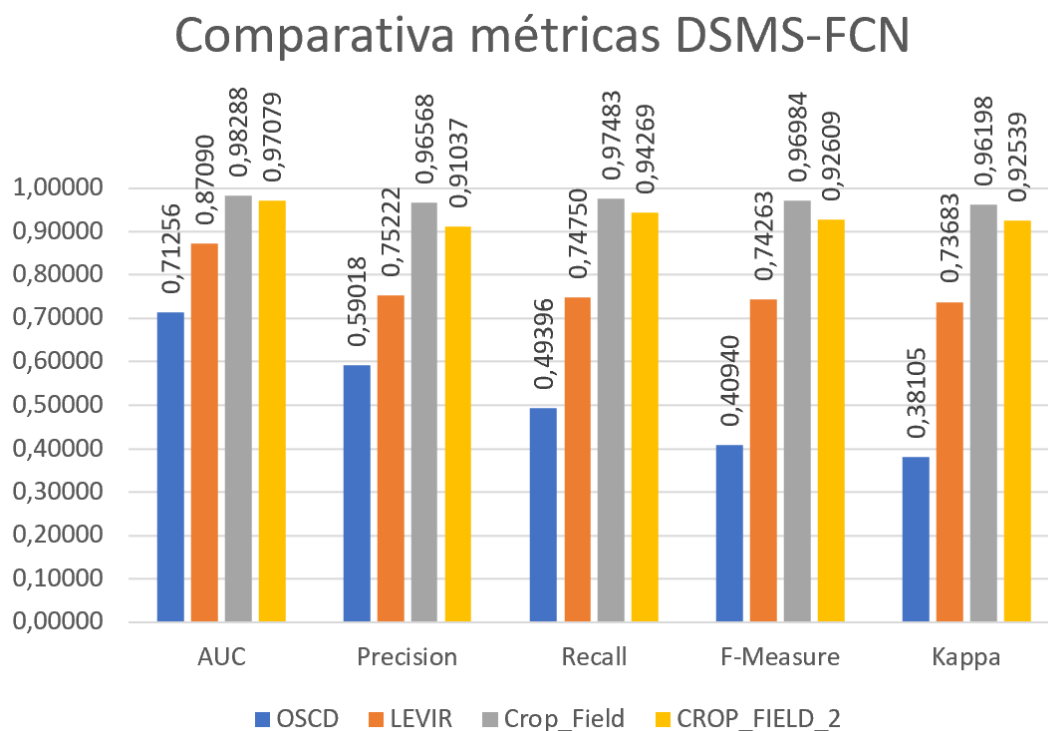


Figura 8.15: Presentación visual de los resultados de Kappa sobre DSMS-FCN.

8.1.6. Experimentaciones con Improved UNet++

Con la experimentación de este método surgió una problemática, ya que tras ejecutarlo con los diferentes datasets, en la mayoría de ellos no se podía obtener un resultado concluyente. Se puede ver un ejemplo de lo ocurrido con uno de los datasets (CROP_FIELD) en la Figura 8.16, donde los números marcan la cantidad de epochs

usados para cada una de los procesos de aprendizaje y ejecuciones realizadas para este algoritmo. Se puede observar como, a mayor cantidad de epochs, el algoritmo no es capaz de detectar ningún supuesto cambio en el par de imágenes. Estas máscaras van perdiendo coherencia a medida que aumenta el número de epoch con el que se esté realizando la experimentación, junto con la cantidad de marcas, de las cuales cada vez aparecen en menor cantidad y en posiciones aparentemente arbitrarias. Estas máscaras de cambio tampoco se crean correctamente, ya que estas deberían de contener únicamente dos valores (0 o 255), pero se puede observar como estas aparecen en escala de grises, por lo que tampoco son útiles para el estudio.

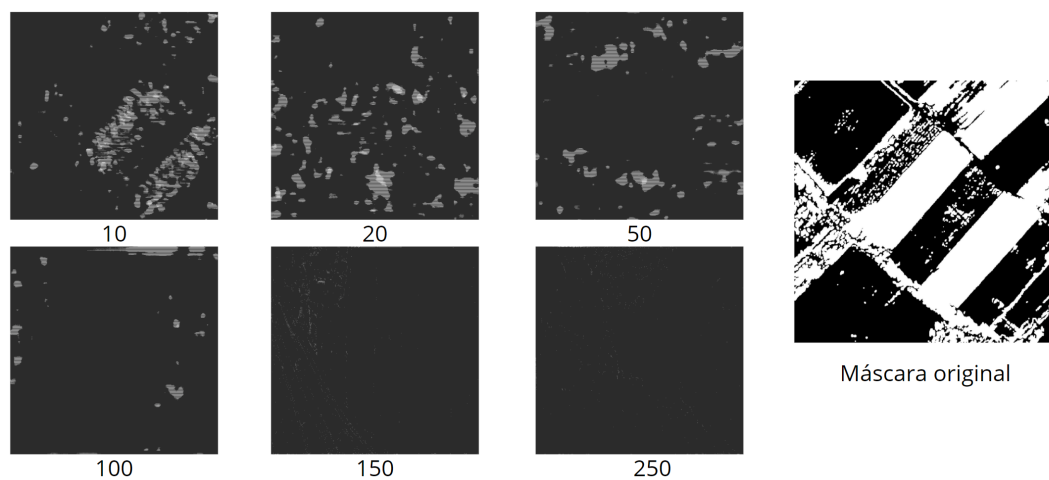


Figura 8.16: Errores encontrados durante la ejecución de los diferentes algoritmos. En la parte inferior se menciona el número de epoch en el que fue extraída la imagen para el dataset de CROP_FIELD.

Esta inestabilidad en la obtención de los resultados se ha repetido en distintos datasets, salvo para el conjunto de datos de LEVIR, para el cual se adjuntan los resultados. Estos resultados posteriormente no serán aplicados a la comparación final con el resto de algoritmos, debido a que no hay datos para el resto de datasets.

Estos resultados se pueden estudiar sobre la Tabla 8.9. Se obtiene una precisión de 0.77 y una puntuación Kappa de 0.72, los cuales pueden ser comparados con los mejores anteriores, donde se obtenía una precisión de 0.77 para el modelo de FC-SD y un Kappa de 0.74 por DSMS-FCN. Se puede ver como este método no obtiene una mejora comparado con los anteriores, las métricas no varían en gran medida, llegando incluso a empeorar con respecto a las demás propuestas.

	AUC	Precision	Recall	F-Measure	Kappa
LEVIR	0,84456	0,77438	0,69372	0,72354	0,71633

Tabla. 8.9: Resultados de UNet++ mejorada sobre LEVIR.

Se puede observar en la Figura 8.17 un ejemplo en el que la red no ha ofrecido muy buen rendimiento, y otro en el que ha obtenido una gran precisión. Estos pequeños fallos son los que han producido que disminuyan las diferentes métricas.

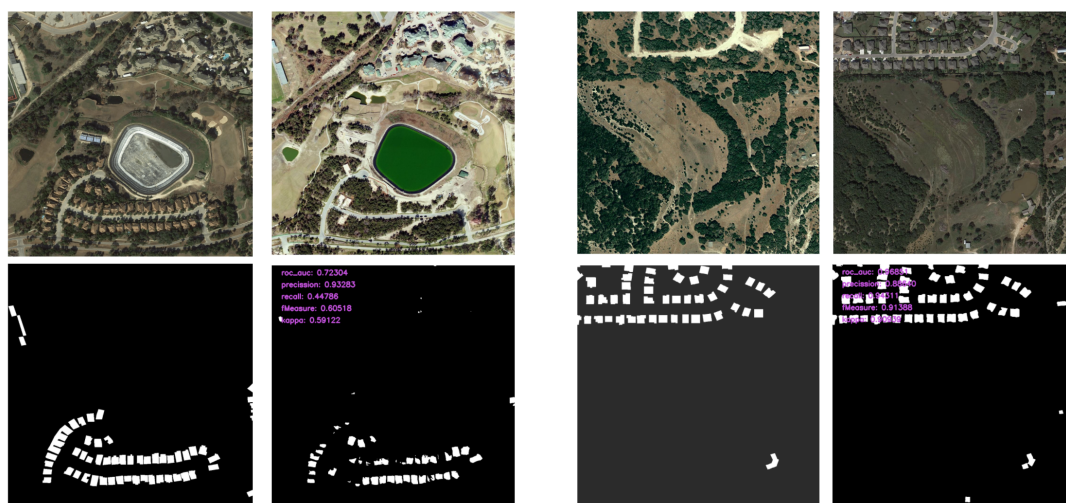


Figura 8.17: Ejemplos de resultados obtenidos por UNet++ mejorada sobre el dataset de LEVIR.

8.2. Comparación de los diferentes métodos

Para realizar la comparación sobre los diferentes métodos se recurrirá a la métrica de Kappa, debido a que esta es una de las más precisas, y a lo largo de la literatura es la más usada para realizar este tipo de comparaciones.

En la Tabla 8.10 se muestra un *ranking* con los diferentes resultados obtenidos para cada dataset, como resultaba lógico, en las posiciones superiores se pueden encontrar siempre los métodos que aplican Aprendizaje Profundo, a excepción de OSCD, donde se obtienen mejores resultados aplicando el método *Image Differencing* de *compare_ssim*.

Por lo general se puede observar un buen rendimiento para los dataset sintéticos, obteniendo para todos los métodos con Aprendizaje Profundo una puntuación mayor a 0.95. Como se ha mencionado anteriormente, esto se debe a que la red detecta fácilmente estos artefactos, ya que la variación entre la intensidad de los píxeles es muy elevada y los bordes no están del todo atenuados, por lo que se detecta perfectamente donde se localizan estos.

Para LEVIR se puede observar una puntuación de Kappa superior al 0.70 en la mayoría de los casos con los métodos que aplican Aprendizaje Profundo, a diferencia

OSCD		LEVIR	
DSMS-FCN	0,38105	DSMS-FCN	0,73683
FC-SC	0,31673	FC-SD	0,73455
FC-SD	0,30487	Unet++	0,71633
compare_ssim	0,27678	FC-SC	0,71237
PCA	0,25639	FC-EF	0,69537
FC-EF	0,20098	PCA	0,05045
Adap ImgDiff (11x11)	0,18114	ImgDiff	0,01401
Adap compare_ssim (31x31)	0,11646	Adap compare_ssim (11x11)	0,00530
Canny	0,03796	compare_ssim	-0,00696
ImgDiff	0,03290	Canny	-0,00929
		Adap ImgDiff (11x11)	-0,01491

CROP_FIELD		CROP_FIELD_2	
FC-SD	0,98225	FC-EF	0,97722
FC-SC	0,96893	FC-SD	0,97722
FC-EF	0,96459	FC-SC	0,97007
DSMS-FCN	0,96198	DSMS-FCN	0,92539
compare_ssim	0,68238	compare_ssim	0,74236
Adap compare_ssim (11x11)	0,56677	Adap compare_ssim(31x31)	0,70027
ImgDiff	0,53743	Adap ImgDiff (31x31)	0,62573
Adap ImgDiff (101x101)	0,47348	PCA	0,50556
PCA	0,44542	ImgDiff	0,12374
Canny	0,24800	Canny	-0,00127

Tabla. 8.10: *Ranking* de los diferentes métodos sobre los datasets.

de con los clásicos, donde la caída es de casi 0.65 puntos.

Para OSCD se obtiene en general los peores resultados de todos los algoritmos, ni los métodos clásicos ni de Aprendizaje Profundo son capaces de funcionar correctamente con estas imágenes de gran escala, no son capaces de encontrar los cambios en un contexto tan grande, a pesar de que se han estudiado métodos de Aprendizaje Profundo que aseguran que son capaces de aprender cambios producidos a grandes escalas. Se obtiene 0.38 de puntuación, justo con el modelo que especificaba que funcionaba mejor con cambios a gran escala.

8.2.1. Análisis estadístico de los resultados

A partir de la tabla anterior, con los diferentes resultados obtenidos por dataset, se puede extraer un *ranking* general con los diferentes métodos. Para ello es tan sencillo como aplicar la media a las posiciones en las que se han posicionado. Se puede ver en la Tabla 8.11 el resultado final. Para este no se ha podido añadir la red de *Unet++*

mejorada ya que sus resultados solo se aplicaban sobre un dataset.

FC-SD	2
DSMS-FCN	2,5
FC-SC	2,5
FC-EF	3,5
compare_ssim	5,5
PCA	6,75
Adap compare_ssim	6,75
Adap ImgDiff	8
ImgDiff	8
Canny	9,5

Tabla. 8.11: Ranking con los diferentes métodos.

Para poder analizar estos métodos también se recurre a test no-paramétricos, ya que no se puede garantizar que los resultados obtenidos sigan una distribución normal, [Demšar \[2006\]](#). Se pretende encontrar si existen diferencias significativas entre los resultados de los diferentes algoritmos, para lo que se necesitan aplicar dos test estadísticos.

- **Test de Friedman:** se utiliza para comparar varias mediciones, en ese caso la métrica generada por los diferentes algoritmos, y determinar que la diferencia entre estos no se deban al azar, es decir, que esta sea estadísticamente significativa. Para aplicarlo se debe de tener en cuenta una hipótesis nula, la cual indica que todos los métodos son equivalentes. Si el valor calculado por este test es menor a 0.05, se rechaza la hipótesis nula, por lo que los métodos si serían considerados significativamente distintos.
- **Test de Nemenyi:** Una vez rechazada la hipótesis nula, se debe realizar una comparación entre todos los métodos. Para hacer esta comparación se basa en el calculo de una distancia crítica, gracias a esta se puede calcular que grupos se encuentran más diferenciados. De nuevo, siguiendo el valor de $\alpha = 0.05$, si alguna de las distancias se encuentra por debajo de este valor de α , quiere decir que dichos métodos son estadísticamente diferentes de manera significativa.

Para aplicar estos test se recurre a una implementación en Python basada en el artículo [Zach \[2021\]](#), donde se explica con detalle cómo aplicar dichos tests.

Se obtiene así como resultado un p-value de 0.00065 para el test de **Friedman**, que al estar bajo 0.05 indica que hay métodos que muestran una gran diferencia. En la tabla 8.12 se observa la salida del test de **Nemenyi**. Se pueden observar de nuevo valores menores a 0.05, más concretamente entre los grupos 1, 2, 3 con el 9. Estos

	0	1	2	3	4	5	6	7	8	9
0	1.000000	0.900000	0.900000	0.900000	0.900000	0.900000	0.558149	0.900000	0.558149	0.156237
1	0.900000	1.000000	0.900000	0.900000	0.773540	0.407017	0.117126	0.407017	0.117126	0.013558
2	0.900000	0.900000	1.000000	0.900000	0.900000	0.594045	0.232504	0.594045	0.232504	0.036105
3	0.900000	0.900000	0.900000	1.000000	0.900000	0.594045	0.232504	0.594045	0.232504	0.036105
4	0.900000	0.773540	0.900000	0.900000	1.000000	0.900000	0.900000	0.900000	0.900000	0.665845
5	0.900000	0.407017	0.594045	0.594045	0.900000	1.000000	0.900000	0.900000	0.900000	0.900000
6	0.558149	0.117126	0.232504	0.232504	0.900000	0.900000	1.000000	0.900000	0.900000	0.900000
7	0.900000	0.407017	0.594045	0.594045	0.900000	0.900000	0.900000	1.000000	0.900000	0.900000
8	0.558149	0.117126	0.232504	0.232504	0.900000	0.900000	0.900000	0.900000	1.000000	0.900000
9	0.156237	0.013558	0.036105	0.036105	0.665845	0.900000	0.900000	0.900000	0.900000	1.000000

Tabla. 8.12: Tabla con los resultados del test de Nemenyi.

más concretamente serían los métodos de FC-SD, FC-SC, DSMS-FCN frente a Canny, lo cual indica, que con respecto a los diferentes métodos, los tres primeros destacan en gran medida con respecto al de Canny.

Se puede observar como los resultados del test anterior coinciden con los resultados obtenidos en el *ranking*, situando en las primeras posiciones FC-SD, DSMS-FCN y FC-SC frente a la última posición de Canny.

Capítulo 9

CONCLUSIONES

En este proyecto se ha realizado una extensa revisión del campo de la detección de cambios, detallando su definición, tareas y sus diferentes elementos (técnicas y herramientas más utilizadas, los diferentes datasets que se pueden usar, métricas, etc). Concretamente se describen de forma detallada los diferentes métodos aplicados para marcar estos cambios entre pares de imágenes, revisando en un primer lugar aquellos métodos más tradicionales, aquellos que trabajan a nivel de píxel, para posteriormente pasar a aquellos más complejos, que aplican diferentes técnicas de Aprendizaje Profundo y que son capaces de aprender diferentes patrones para actuar en consecuencia. También se ha realizado una experimentación con estos para poder extraer conclusiones.

Como se pudo revisar en el Capítulo 3, las principales herramientas utilizadas en general en el ámbito de la ciencia de datos tienden a ser bastante intuitivas y fáciles de usar, para poder facilitar así la tarea de análisis de los datos. Destaca el uso de Python como lenguaje de programación, que junto con TensorFlow y Keras, se obtiene una herramienta potente capaz de ejecutar y entrenar diferentes métodos que apliquen Aprendizaje Profundo.

En cuanto a los datasets, en el Capítulo 5, se dejan claros los diferentes problemas que se encuentran al tratar sobre estos. Los pocos conjuntos de datos existentes puede que no resulten válidos para aplicar a la detección de cambios ya que estas máscaras pueden considerarse como incompletas, debido a que se centran en cambios específicos (como edificaciones y expansión urbana en general), cuando quizás interesaría investigar sobre otros ámbitos. Al hacer uso de datasets sintéticos se han podido observar los diferentes problemas que estos implican, en una futura línea de trabajo, sería interesante mejorar el proceso de generación de estos, para que los

cambios generados no sean tan abruptos. El problema que se ha encontrado durante el desarrollo de estos, es que se solapan los píxeles como si fueran parches directamente sobre la imagen, por lo que el borde de estos parches (y el cambio abrupto de color) hace que el algoritmo pueda detectarlo sin ningún problema. Se ha podido comprobar como el proceso de creación de un dataset es muy complejo y la obtención de las máscaras no es un proceso trivial.

El problema principal a la hora de desarrollar todo este trabajo ha sido la búsqueda y aplicación de los algoritmos a los diferentes datasets, ya que existen numerosos problemas a la hora de obtener estas imágenes: diferente exposición a la luz solar (lo que genera cambios en las tonalidades de los colores, o que las sombras aparezcan desplazadas), la alineación geográfica, las condiciones atmosféricas, etc. El preprocesamiento de las imágenes no es un proceso sencillo y automático, ya que la mayoría de las veces necesitará de una posterior revisión manual, y considerando el tamaño de los conjuntos de datos, puede resultar una tarea bastante tediosa.

Una línea de investigación para futuros trabajos podría ser el estudio del preprocesamiento de los datasets y la creación de los mismos a partir de pares de imágenes.

Las métricas explicadas en el Capítulo 4 ayudan al posterior análisis y comparación de los diferentes métodos. La mayoría de estas métricas son las que se han encontrado a lo largo del proceso de documentación para este trabajo, considerándolas como las más utilizadas. Con la descripción de estas lo que se pretendía era dar a conocer su funcionamiento para poder así interpretarlas mejor.

Posteriormente a todo esto se han mostrado los diferentes métodos escogidos para la experimentación de este trabajo, encontrando tanto métodos tradicionales como aquellos que aplican Redes Neuronales profundas. En este apartado cabe destacar las implementaciones utilizadas, ya que algunas de estas fueron fáciles de encontrar, en todos los casos hubo que adaptarlas para que funcionaran correctamente en el equipo o para arreglar diferentes errores, este último era el caso de PCA K-means. En el caso de los métodos tradicionales, hubo que realizar las implementaciones de algunos de estos, por ejemplo de Canny o de Image Differencing y su versión adaptativa. En el caso de los métodos de Aprendizaje Profundo, se encontraron implementaciones para todos ellos, pero tuvieron que ser adaptadas para que todas ellas trabajasen con una versión concreta de las librerías.

Al hacer las ejecuciones y entrenamientos de los diferentes algoritmos, solo se tienen en cuenta las máscaras del autor, de esta manera se pretende hacer un estudio equitativo sobre todos los datasets. Como ya se ha mencionado antes, estas máscaras pueden considerarse como incompletas, pero no se ha encontrado una forma de

comprobar si estos algoritmos marcaban cambios que realmente no aparecían en las máscaras. Puede que muchos de los resultados obtenidos sean mejor de lo que se indican, pero debido a esta problemática, no es posible averiguarlo.

Como se ha podido comprobar, los métodos que aplican Aprendizaje Profundo destacan frente a los tradicionales en gran medida. Destaca entre todos ellos **FC-SD**, encontrando a **DSMS-FCN** y **FC-SC** igualados en segunda posición. Se puede concluir entonces que las arquitecturas que combinan redes Siamesas y Codificadoras-Decodificadoras son las que ofrecen mejores resultados en este campo, lo cual puede deberse a que se extraen las características de cada par de forma independiente en cada ramificación. Mientras que el codificador y decodificador ayuda a extraer las características relevantes a medida que se introduce en la profundidad de la red.

Tras realizar diferentes test estadísticos sobre los mejores resultados obtenidos para cada algoritmo, se comprueba que no suele haber diferencias estadísticas entre los diferentes métodos, solo destacan los citados anteriormente con respecto a **Canny**, que es el algoritmo para el que se obtienen peores resultados. De nuevo se puede destacar el gran rendimiento de estos métodos frente a los tradicionales.

Se destaca también, que dependiendo de los datasets, se ha podido comprobar como hay métodos que funcionan con un mayor o menor rendimiento. Si se quiere resolver un problema en concreto se debería hacer un estudio sobre los diferentes métodos para comprobar cual funcionaría mejor.

Como posibles líneas de trabajo futuras adicionales, se podrían seguir buscando diferentes métodos. En el Capítulo 2 no solo se mencionan los estudiados en este proyecto, sino otros que se han encontrado en la literatura y que destacan en el campo, pero para los cuales no se encontró una implementación o dataset sobre el que poder aplicarlos. Para los métodos con Redes Neuronales profundas, se debería intentar explorar otras arquitecturas diferentes a las ya mencionadas (Siamesas y Encoder-Decoder) con el objetivo de comprobar si realmente estas son las que ofrecen mejor rendimiento en el campo de la detección de cambios.

Bibliografía

A. Rosebrock. *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch, 2017. URL <https://books.google.es/books?id=9U1-tgEACAAJ>.

Grand View Research. Computer vision market size, share trends analysis report by component (hardware, software), by product type (smart camera-based, pc-based), by application, by vertical, by region, and segment forecasts, 2020 - 2027, 2020. URL <https://www.grandviewresearch.com/industry-analysis/computer-vision-market>.

Tiobe. Tiobe index for june 2021, 2021. URL <https://pythoneyes.wordpress.com/2017/06/19/filtro-de-gauss-con-python-3/>. comprobado en 2021-07-15.

Curva roc, 2021. URL https://es.wikipedia.org/wiki/Curva_ROC.

Yang Long. On creating benchmark dataset for aerial image interpretation: Reviews, guidances and million-aid, 2021. URL <https://captain-whu.github.io/DiRS/>. comprobado en 2021-05-28.

Csaba Benedek. Sztaki airchange benchmark set, 2013. URL http://web.eee.sztaki.hu/remotesensing/airchange_benchmark.html. comprobado en 2021-05-28.

Nicolas Bourdis. Change detection dataset, 2020. URL <https://computervisiononline.com/dataset/1105138664>. comprobado en 2021-05-28.

Rodrigo Caye Daudt. Onera satellite change detection dataset, 2018. URL <https://rcdaudt.github.io/oscd/>. comprobado en 2021-05-28.

Rodrigo Caye Daudt. High resolution semantic change detection dataset, 2021. URL <https://ieee-dataport.org/open-access/hrscd-high-resolution-semantic-change-detection-dataset>. comprobado en 2021-05-28.

xbd dataset, annotated high-resolution satellite imagery for building damage assessment, 2021. URL <https://xview2.org/dataset>. comprobado en 2021-05-28.

- Hao Chen and Zhenwei Shi. A spatial-temporal attention-based method and a new dataset for remote sensing image change detection. *Remote Sensing*, 12(10), 2020. ISSN 2072-4292. doi: 10.3390/rs12101662. URL <https://www.mdpi.com/2072-4292/12/10/1662>. comprobado en 2021-06-01.
- Shiqi Tian. Hi-ucd: A large-scale dataset for urban semantic change detection in remote sensing imagery, 2020. comprobado en 2021-06-01.
- Sn7: Multi-temporal urban development challenge, 2021. URL <https://spacenet.ai/sn7-challenge/>. comprobado en 2021-05-28.
- Jason Brownlee. How to configure image data augmentation in keras, 2019. URL <https://bit.ly/2Tqfcw3>. comprobado en 2021-06-05.
- Raziel. Filtro de gauss con python 3, 2017. URL <https://www.tiobe.com/tiobe-index/>. comprobado en 2021-06-15.
- OpenCV. Image processing in opencv, 2021. URL https://docs.opencv.org/4.5.2/d2/d96/tutorial_py_table_of_contents_imgproc.html.
- Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL <https://proceedings.neurips.cc/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf>.
- Nikhil Kumar Tomar. Unet implementation in tensorflow using keras api, 2021. URL <https://idiotdeveloper.com/unet-implementation-in-tensorflow-using-keras-api/>. comprobado en 2021-07-07.
- Rodrigo Caye Daudt, Bertrand Le Saux, and Alexandre Boulch. Fully convolutional siamese networks for change detection, 2018.
- Hongruixuan Chen, Chen Wu, Bo Du, and Liangpei Zhang. Change detection in multi-temporal vhr images based on deep siamese multi-scale convolutional networks, 2020.
- Stuart J. Russell. Inteligencia artificial : un enfoque moderno, 2006.
- David James. *Introduction to Machine Learning with Python: A Guide for Beginners in Data Science*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 1st edition, 2018. ISBN 1726230872.

- MATLAB. *Machine Learning with MATLAB*. 2021. URL <https://es.mathworks.com/content/dam/mathworks/ebook/gated/machine-learning-ebook-all-chapters.pdf>.
- Kaggle. Intro to deep learning. 2021. URL <https://www.kaggle.com/ryanholbrook/a-single-neuron>.
- Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edition, 2005.
- S. Minu and Amba Shetty. A comparative study of image change detection algorithms in matlab. *Aquatic Procedia*, 4:1366–1373, 2015. ISSN 2214-241X. doi: <https://doi.org/10.1016/j.aqpro.2015.02.177>. URL <https://www.sciencedirect.com/science/article/pii/S2214241X15001789>. INTERNATIONAL CONFERENCE ON WATER RESOURCES, COASTAL AND OCEAN ENGINEERING (ICWRCOE'15).
- Richard Szeliski. Computer vision algorithms and applications, 2011. URL <http://dx.doi.org/10.1007/978-1-84882-935-0>.
- William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57, Sep. 1992. doi: 10.1609/aimag.v13i3.1011. URL <https://ojs.aaai.org/index.php/aimagazine/article/view/1011>.
- Jesús García, José M. Molina, Antonio Berlanga, Miguel A. Patricio, Álvaro L. Bustamante, and Whashington R. Padilla. *Ciencia de datos. Técnicas analíticas y aprendizaje estadístico en un enfoque práctico: Big Data*. Publicaciones Altaira, 2018. ISBN 9788494731969. URL https://d1wqtxts1xzle7.cloudfront.net/64031156/Ciencia_de_datos_2018.pdf.
- Álvaro. Problemas comunes en aprendizaje automático, 2018. URL <https://machinelearningparatodos.com/problemas-comunes-en-aprendizaje-automatico/>.
- O. Maimon and L. Rokach. *Data mining and knowledge discovery handbook*, volume 14. Springer, 2010. URL http://scholar.google.de/scholar.bib?q=info:wwQ2svFQuQEJ:scholar.google.com/&output=citation&hl=de&as_sdt=0,5&ct=citation&cd=0.
- Jorge Valverde-Rebaza. Detección de bordes mediante el algoritmo de canny. 10 2007. URL https://www.researchgate.net/publication/267240432_Deteccion_de_bordes_mediante_el_algoritmo_de_Canny. comprobado en 2021-06-05.

- Ian T. Jolliffe and Jorge Cadima. *Principal component analysis: a review and recent developments*. Royal Society, 2nd edition, 2016. doi: <https://doi.org/10.1098/rsta.2015.0202>. URL <https://royalsocietypublishing.org/doi/full/10.1098/rsta.2015.0202>.
- Turgay Celik. Unsupervised change detection in satellite images using principal component analysis and k -means clustering. *IEEE Geoscience and Remote Sensing Letters*, 6(4):772–776, 2009. doi: 10.1109/LGRS.2009.2025059.
- D. Lu Corresponding author, P. Mausel, E. Brondízio, and E. Moran. Change detection techniques. *International Journal of Remote Sensing*, 25(12):2365–2401, 2004. doi: 10.1080/0143116031000139863. URL <https://doi.org/10.1080/0143116031000139863>.
- Rodrigo Borrego Lorena, João Roberto Dos Santos, Yosio Edemir Shimabukuro, Irving Foster Brown, Hermann Johann, and Heinrich Kux. A change vector analysis technique to monitor land use/land cover in sw brazilian amazon: Acre state.
- C. Huang, B. Wylie, L. Yang, C. Homer, and G. Zylstra. Derivation of a tasselled cap transformation based on landsat 7 at-satellite reflectance. *International Journal of Remote Sensing*, 23(8):1741–1748, 2002. doi: 10.1080/01431160110106113. URL <https://doi.org/10.1080/01431160110106113>.
- Kevin Louis de Jong and Anna Sergeevna Bosman. Unsupervised change detection in satellite images using convolutional neural networks, 2019.
- Wahyu Wiratama, Jongseok Lee, Sang-Eun Park, and Donggyu Sim. Dual-dense convolution network for change detection of high-resolution panchromatic imagery. *Applied Sciences*, 8(10), 2018. ISSN 2076-3417. doi: 10.3390/app8101785. URL <https://www.mdpi.com/2076-3417/8/10/1785>.
- Numpy. Página oficial de numpy, 2020. URL <https://numpy.org/>.
- scikit learn. Página oficial de scikit-learn, 2021. URL <https://scikit-learn.org/stable/>.
- Keras. Documentación keras, 2021. URL <https://keras.io/api/>.
- Aumento de datos, 2021. URL https://www.tensorflow.org/tutorials/images/data_augmentation. comprobado en 2021-07-11.
- Ryan Mukherjee. fmow, 2020. URL <https://github.com/fMoW/dataset>. comprobado en 2021-05-28.

- Anannaya Uberoi. Image registration using opencv | python, 2019. URL <https://www.geeksforgeeks.org/image-registration-using-opencv-python/>. comprobado en 2021-06-21.
- S. Wright. *Digital Compositing for Film and Video*. Focal Press visual effects and animation series. Elsevier/Focal Press, 2010. ISBN 9780240813097. URL https://books.google.es/books?id=oEamKQBtn_0C.
- Anastasia Murzova and Sakshi Seth. Otsu's thresholding with opencv, 2020. URL <https://learnopencv.com/otsu-thresholding-with-opencv/>. comprobado en 2021-07-12.
- Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861.
- Turgay Celik. Change-detection-in-satellite-imagery, 2019. URL <https://github.com/abhijeet3922/Change-Detection-in-Satellite-Imagery>.
- Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24574-4.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.
- Daifeng Peng, Yongjun Zhang, and Haiyan Guan. End-to-end change detection for high resolution satellite images using improved unet++. *Remote Sensing*, 11(11), 2019. ISSN 2072-4292. doi: 10.3390/rs11111382. URL <https://www.mdpi.com/2072-4292/11/11/1382>.
- Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation, 2018.
- Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006. ISSN 1532-4435.
- Zach. How to perform the nemenyi post-hoc test in python, 2021. URL <https://www.statology.org/nemenyi-test-python/>.