UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

DATA AUGMENTATION BASADO EN CAMBIOS DE ILUMINACIÓN USANDO REDES GAN PARA APLICACIONES DE RECONOCIMIENTO DE ENTORNOS

TRABAJO FIN DE GRADO

Diciembre - 2024

AUTOR: Carlos García López

DIRECTOR/ES: Mónica Ballesta Galdeano

ÍNDICE

1. INTRODUCCIÓN

La robótica se ha convertido en una disciplina indispensable en la actualidad en diversos sectores que van desde la industria, hasta el mundo de la medicina, pasando por el transporte y la exploración del espacio. Esto se debe a la capacidad de los robots para automatizar procesos, mejorar la precisión y eficiencia, y asumir tareas que pueden resultar peligrosas para los humanos.

Uno de los avances más significativos en la robótica es la aparición de robots autónomos que cuenten con la habilidad de reconocer un entorno y poder desenvolverse en él. Dicha habilidad es conocida como localización y mapeo simultáneo (*SLAM*, en inglés), y es lo que permite a un robot poder construir un mapa del entorno a la vez que determina su posición dentro del propio mapa [1]. De esta manera, el robot puede moverse de manera segura y eficiente, a la misma vez que ejecuta tareas complejas y se adapta a espacios dinámicos.

Sin embargo, para realizar esta tarea no es necesario un robot de última generación que se encuentre diseñado para operar en un entorno peligroso o desconocido. Un ejemplo más cotidiano sería el de los robots aspiradores domésticos que utilizan esta habilidad para trabajar de manera más eficiente en nuestros propios hogares, lo que demuestra que no solo se usan para aplicaciones industriales y que muestra cómo esta tecnología puede ser accesible para todo el mundo.

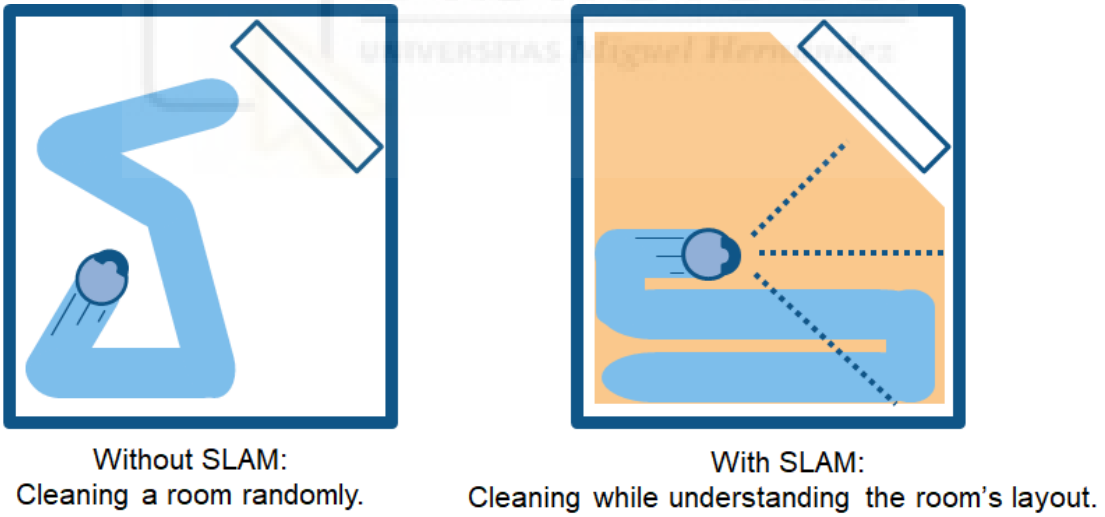


Figura 1: Ejemplo de un robot aspirador que muestra la eficiencia del *SLAM*.

La clave está en que el robot pueda reconocer el entorno incluso ante cambios en condiciones como la iluminación, el clima o la presencia de obstáculos. Para lograr esto, se emplean tecnologías avanzadas como son los reconocidos sensores *LIDAR* [2] y distintos algoritmos de visión por computadora. Lo que se pretende conseguir aquí es implementar algoritmos que a partir de escaneos *LIDAR* se consiga construir un mapa del entorno y recuperar la trayectoria del robot.

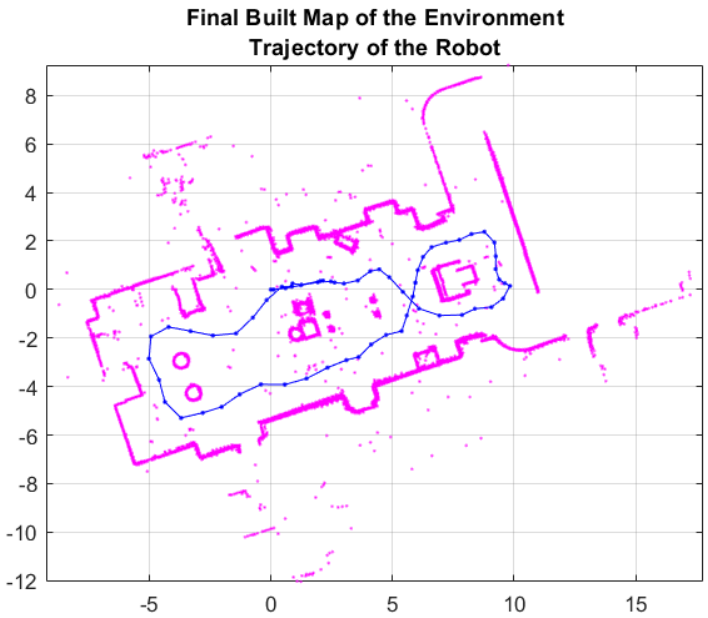


Figura 2: reconstrucción del mapa que ha escaneado el robot con *LIDAR*

Sin embargo, en los últimos años se ha impulsado el avance en la Inteligencia Artificial [3], la cual refuerza los sistemas *SLAM* mediante el procesamiento de información sensorial en tiempo real, el aprendizaje de la experiencia y la toma de decisiones optimizada.

Nosotros, en cambio, vamos a utilizar la IA con la idea de preparar al robot para diferentes posibles cambios de iluminación que pueden ocurrir en un lugar determinado, de modo que sea capaz de identificar el entorno en cualquier situación.

1.1 Objetivo

El objetivo principal del Trabajo de Fin de Grado (TFG) es la realización de un proceso de *Data Augmentation*, es decir, la ampliación de una base de datos de imágenes de un entorno de oficinas mediante la generación de imágenes sintéticas que reflejan cambios en las condiciones de iluminación (soleado, nublado y de noche). Para lograr este objetivo, se hará uso de redes generativas antagónicas (*GANs*) [4], una técnica de Inteligencia Artificial que permite crear imágenes nuevas basadas en las características aprendidas de imágenes reales. De esta manera queremos conseguir que el reconocimiento de entornos sea más robusto y preciso, independientemente de las variaciones lumínicas en los datos.

Partiremos de tres bases de datos independientes que contienen imágenes del entorno en días soleados, en días nublados y cuando es de noche. Todas las bases de datos contienen imágenes de los mismos lugares de la oficina, ya que el robot utilizado realiza el mismo recorrido en las tres iluminaciones.

Nuestra red GAN utilizada es la conocida como Cyclegan, la cual está compuesta por 2 redes GAN que trabajan de manera simultánea y que permite la creación de imágenes con 2 dominios distintos. Es decir, partiremos de dos de esas tres bases de datos (soleado y de noche, por ejemplo) y crearemos imágenes sintéticas que transformen las imágenes reales

de una de esas dos bases de datos en el dominio de la otra base de datos utilizada (transformar imágenes soleadas en imágenes de noche). Procederemos a describir esta red de una manera más exhaustiva en el apartado 2.4.

Como queremos hacer esto con las tres bases de datos y con que se generen imágenes unas a partir de las otras dos, será necesario realizar 3 experimentos diferentes con el que conseguiremos imágenes nubladas y nocturnas a partir de soleadas, imágenes nubladas a partir de soleadas y de noche, y también imágenes nocturnas a partir de las soleadas y las nubladas. Por eso tendremos tres modelos distintos de entrenamiento de la red Cyclegan:

- *Sunny-Night*
- *Sunny-Cloudy*
- *Night-Cloudy*

A través de este proceso, se pretende generar un conjunto de imágenes sintéticas precisas con diversas condiciones de iluminación que amplíen significativamente la base de datos original. Con una base de datos más extensa y variada, será posible utilizar este conjunto de datos en futuras aplicaciones de reconocimiento de entornos, permitiendo a sistemas autónomos o de visión artificial ser más robustos frente a cambios en las condiciones lumínicas del entorno. Esto puede mejorar aplicaciones en áreas como la navegación autónoma, el reconocimiento de escenas o el mapeo y localización simultánea (SLAM) en entornos de oficinas.

Sin embargo, para alcanzar este objetivo principal, hemos planteado distintos objetivos más pequeños y específicos:

- **Selección de las rutas de la base de datos Cold para entrenamiento y test**: La base de datos Cold, la cual es la base de datos principal, contiene imágenes de entornos de oficinas que se utilizarán como punto de partida. El primer paso será seleccionar adecuadamente las rutas que se emplearán para el entrenamiento de la red GAN, para la validación del entrenamiento y aquellas que se utilizarán para pruebas. Es importante que estas rutas representen una variedad suficiente de situaciones del entorno para asegurar que el modelo capture la mayor cantidad de detalles posibles.

- **Adaptación de las imágenes al formato de entrada de la red GAN**: Antes de entrenar la red GAN, será necesario realizar una preprocesamiento de las imágenes de la base de datos. Aquí nos encargamos de convertir las imágenes omnidireccionales procedentes de la base de datos original, en imágenes panorámicas que son las necesarias para aumentar la base de datos del objetivo.

- **Entrenamiento de la red GAN para la generación de imágenes con cambios en la iluminación**: Una vez preparada la base de datos, se entrenará la red GAN. Durante este proceso, la red generadora creará nuevas imágenes basadas en las reales, mientras que la red discriminadora evaluará la autenticidad de estas imágenes. A través de este proceso antagónico, ambas redes mejorarán, permitiendo que la red generadora cree

imágenes sintéticas cada vez más realistas con diferentes condiciones de iluminación. Aquí es donde tendremos que crear tres redes distintas para trabajar con todos los cambios de iluminación.

- **Evaluación de la calidad de las imágenes generadas**: Después del entrenamiento, se evaluarán las imágenes generadas para verificar su calidad. Se espera que las imágenes sintéticas sean realistas y muestren correctamente las variaciones en la iluminación deseadas. Este análisis incluirá tanto evaluaciones visuales como métricas cuantitativas, como el ***Cycle Consistency Loss***, que mide la similitud entre la imagen real de partida y la que se recrea a partir de la imagen general. Es decir comparamos la imagen soleada real que se introduce en el generador de imágenes nocturnas, con dicha imagen generada nocturna cuando se introduce en el generador de imágenes soleadas, de esta manera vemos si es capaz de crear la imagen exactamente a la original.

En resumen, el objetivo de este TFG es no solo generar imágenes sintéticas, sino también asegurar que estas aporten valor real al proceso de entrenamiento de futuros sistemas de reconocimiento de entornos. El uso de GANs para la manipulación de iluminación en imágenes de oficinas permitirá una mejor preparación para enfrentar escenarios reales, donde la iluminación puede variar drásticamente y afectar el rendimiento del sistema de reconocimiento o navegación.

1.2. Estructura del TFG

Nuestro TFG se va a estructurar de la siguiente manera:

En primer lugar, en el **apartado 2** de esta memoria vamos a hacer una introducción con los **conceptos previos** que pongan en contexto todo lo relacionado con la inteligencia artificial y sobre todo con nuestra red *GAN* utilizada (Cyclegan). Aquí sentaremos las bases y desarrollaremos todos los conceptos necesarios para comprender el funcionamiento de la red, comenzando por explicar de manera superficial lo que es el machine learning y una red neuronal, hasta profundizar en la estructura de la propia red.

A continuación, en el **punto 3** pasaremos a hablar sobre las **herramientas** utilizadas para llevar a cabo este trabajo. Aquí explicaremos la base de datos utilizada así como el software en el que se ha desarrollado todo el proyecto.

Después avanzamos a la **sección 4** donde trataremos todo el tema de **experimentos**. Aquí se explicarán los procedimientos aplicados para conseguir los distintos objetivos comentados anteriormente. En este apartado veremos con profundidad como se ha llevado a cabo el entrenamiento y qué resultados hemos obtenido tanto en el entrenamiento como en el test.

Por último, **concluimos** el trabajo realizado en el **punto 5**, explicando los resultados obtenidos y los posibles futuros trabajos que se pueden realizar a partir de este TFG.

Cabe destacar que esta memoria se completará al final con la **bibliografía** (apartado número **6**) utilizada para todas las referencias que se mencionan.

2. CONCEPTOS PREVIOS

El marco teórico en el que se basa nuestro trabajo está relacionado con el campo del *Machine Learning* por lo que es importante introducir este tema previamente.

2.1 *Machine Learning*

El *Machine Learning* o aprendizaje automático [5] es un campo científico y, más particularmente, una subcategoría de inteligencia artificial. Consiste en dejar que los algoritmos descubran *«patterns»*, es decir, patrones recurrentes, en conjuntos de datos. Esos datos pueden ser números, palabras, imágenes, estadísticas, etc.

Todo lo que se pueda almacenar digitalmente puede servir como dato para el *Machine Learning*. Al detectar patrones en esos datos, los algoritmos aprenden y mejoran su rendimiento en la ejecución de una tarea específica.

En resumen, los algoritmos de *Machine Learning* aprenden de forma autónoma a realizar una tarea o hacer predicciones a partir de datos y mejorar su rendimiento con el tiempo. Una vez entrenado, el algoritmo podrá encontrar los patrones en nuevos datos.

¿Cómo funciona?

El primer paso es seleccionar y preparar un conjunto de datos de entrenamiento. Esos datos se utilizarán para alimentar el modelo de *Machine Learning* para aprender a resolver el problema para el que se ha diseñado.

Los datos se pueden etiquetar para indicarle al modelo las características que debe identificar. También pueden estar sin etiquetar, entonces será el modelo el que deberá detectar y extraer características recurrentes por sí mismo.

En ambos casos, los datos deben prepararse, organizarse y limpiarse cuidadosamente. De lo contrario, el entrenamiento del modelo de *Machine Learning* puede estar sesgado. Los resultados de sus predicciones futuras se verán afectados directamente.

El segundo paso es seleccionar un algoritmo para ejecutar sobre el conjunto de datos de entrenamiento. El tipo de algoritmo que se emplea depende del tipo y del volumen de datos de entrenamiento y del tipo de problema que haya que resolver.

El tercer paso es entrenar el algoritmo. Es un proceso de repetición. Las variables se ejecutan a través del algoritmo y los resultados se comparan con los que debería haber producido. Los «pesos» y el sesgo se pueden ajustar para aumentar la precisión del resultado.

Después se vuelve a ejecutar las variables hasta que el algoritmo produzca el resultado correcto en la mayoría de los casos. El algoritmo entrenado es el modelo de *Machine Learning*.

El cuarto y último paso es el uso y la mejora del modelo. Utilizamos el modelo sobre nuevos datos, cuyo origen depende del problema que haya que resolver.

2.2 Redes neuronales convolucionales

Redes neuronales

Una red neuronal [6], el cual es un subcampo del *Machine Learning*, es un modelo simplificado que emula el modo en que el cerebro humano procesa la información: Funciona simultaneando un número elevado de unidades de procesamiento interconectadas que parecen versiones abstractas de neuronas.

Una neurona artificial tiene una entrada x y una salida a, también llamada activación. Para obtener la activación, la neurona ejecuta dos operaciones matemáticas:

- Transformación: es el resultado de tomar el dato de entrada (x) y aplicar la operación $wx + b$. Los parámetros w y b se calculan de forma automática durante el entrenamiento, generalmente usando el algoritmo de Gradiente Descendente (el cual encuentra el mínimo de una función).
- Función de activación: es una función matemática no lineal que transforma el valor obtenido de la operación $wx + b$ al rango de valores 0 a 1, usando la función sigmoidal. De forma similar, en la regresión multiclase, la función *softmax* permite transformar $wx + b$ en una probabilidad entre 0 y 1.

Podemos decir entonces que una neurona artificial se encarga de ponderar los datos de entrada y de posteriormente modificar este resultado usando una función de activación no lineal. Esta neurona es el bloque fundamental de toda red neuronal.

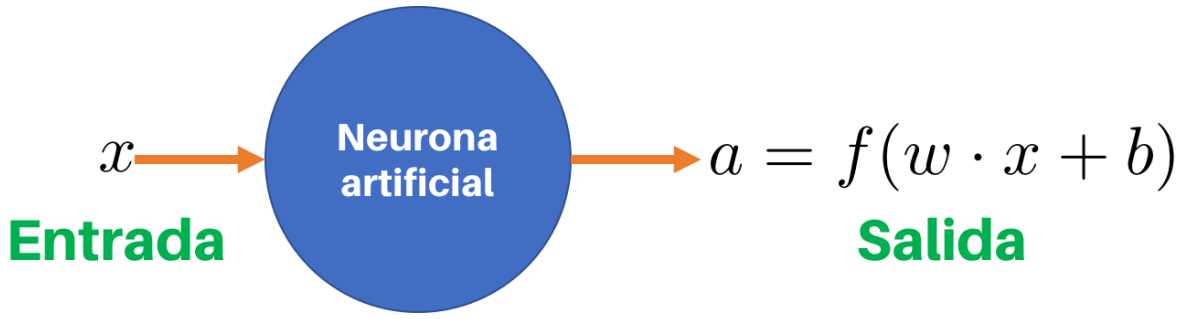


Figura 3: Una neurona artificial

Las unidades de procesamiento se organizan en capas. Hay tres partes normalmente en una red neuronal : una capa de entrada, con unidades que representan los campos de entrada; una o varias capas ocultas; y una capa de salida, con una unidad o unidades que

representa el campo o los campos de destino. Las unidades se conectan con fuerzas de conexión variables (o ponderaciones). Los datos de entrada se presentan en la primera capa, y los valores se propagan desde cada neurona hasta cada neurona de la siguiente capa. Al final, se envía un resultado desde la capa de salida.

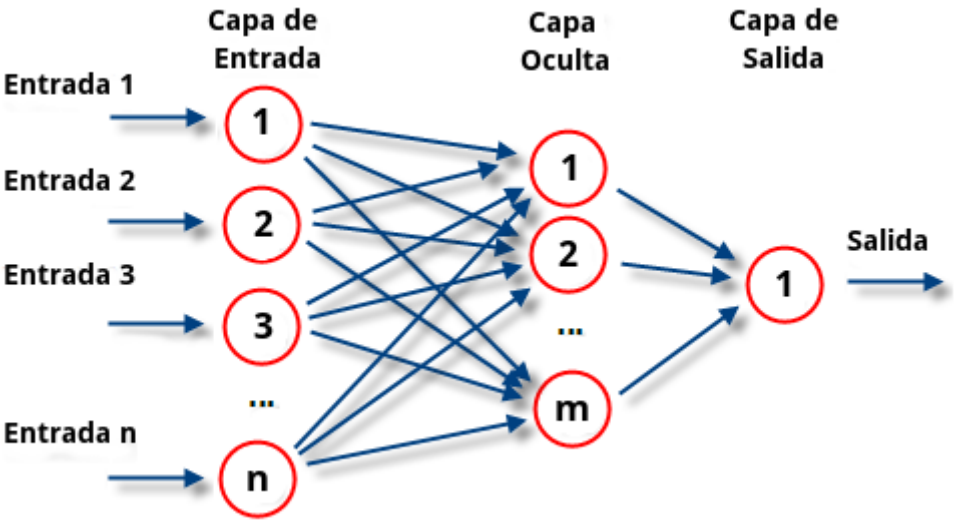


Figura 4: Esquema de una red neuronal

La red aprende examinando los registros individuales, generando una predicción para cada registro y realizando ajustes a las ponderaciones cuando realiza una predicción incorrecta. Este proceso se repite muchas veces y la red sigue mejorando sus predicciones hasta haber alcanzado uno o varios criterios de parada.

Al principio, todas las ponderaciones son aleatorias y las respuestas que resultan de la red son, posiblemente, disparatadas. La red aprende a través del entrenamiento. Continuamente se presentan a la red ejemplos para los que se conoce el resultado, y las respuestas que proporciona se comparan con los resultados conocidos. La información procedente de esta comparación se pasa hacia atrás a través de la red, cambiando las ponderaciones gradualmente. A medida que progresa el entrenamiento, la red se va haciendo cada vez más precisa en la replicación de resultados conocidos. Una vez entrenada, la red se puede aplicar a casos futuros en los que se desconoce el resultado.

Redes Convolucionales

Al igual que las Redes Neuronales, las Redes Convolucionales [7] también permiten detectar patrones en los datos de entrada, con la única diferencia de que en el caso de las Redes Convolucionales los datos de entrada son imágenes.

En este caso vemos la imagen de una fruta que es procesada por una Red Convolucional. En esta red se entrenan diferentes filtros (o *kernels*) y estos filtros permiten extraer características de esa imagen y posteriormente realizar la clasificación en una de diferentes categorías:

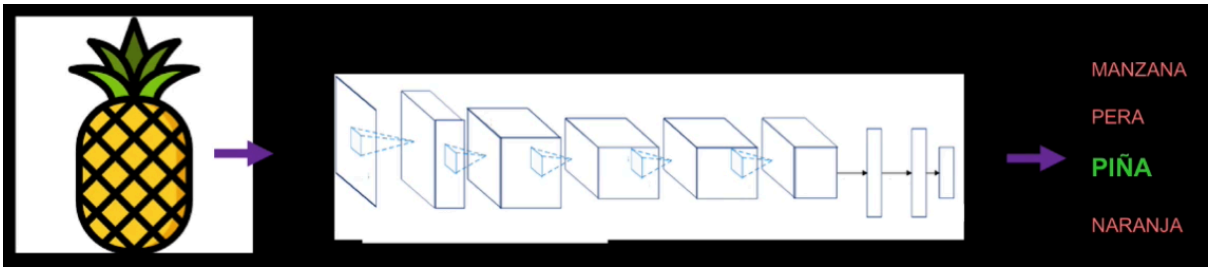


Figura 5: El principio básico de la clasificación con Redes Convolucionales

Pero ¿cómo logran hacer esto las Redes Convolucionales? Básicamente lo que hacen es imitar el cerebro humano y la forma como este procesa las imágenes a través de la corteza visual.

Aquí vemos por ejemplo que la corteza visual, ubicada en la parte trasera de nuestro cerebro, tiene diferentes regiones y diferentes grupos de neuronas encargados de varias funciones:

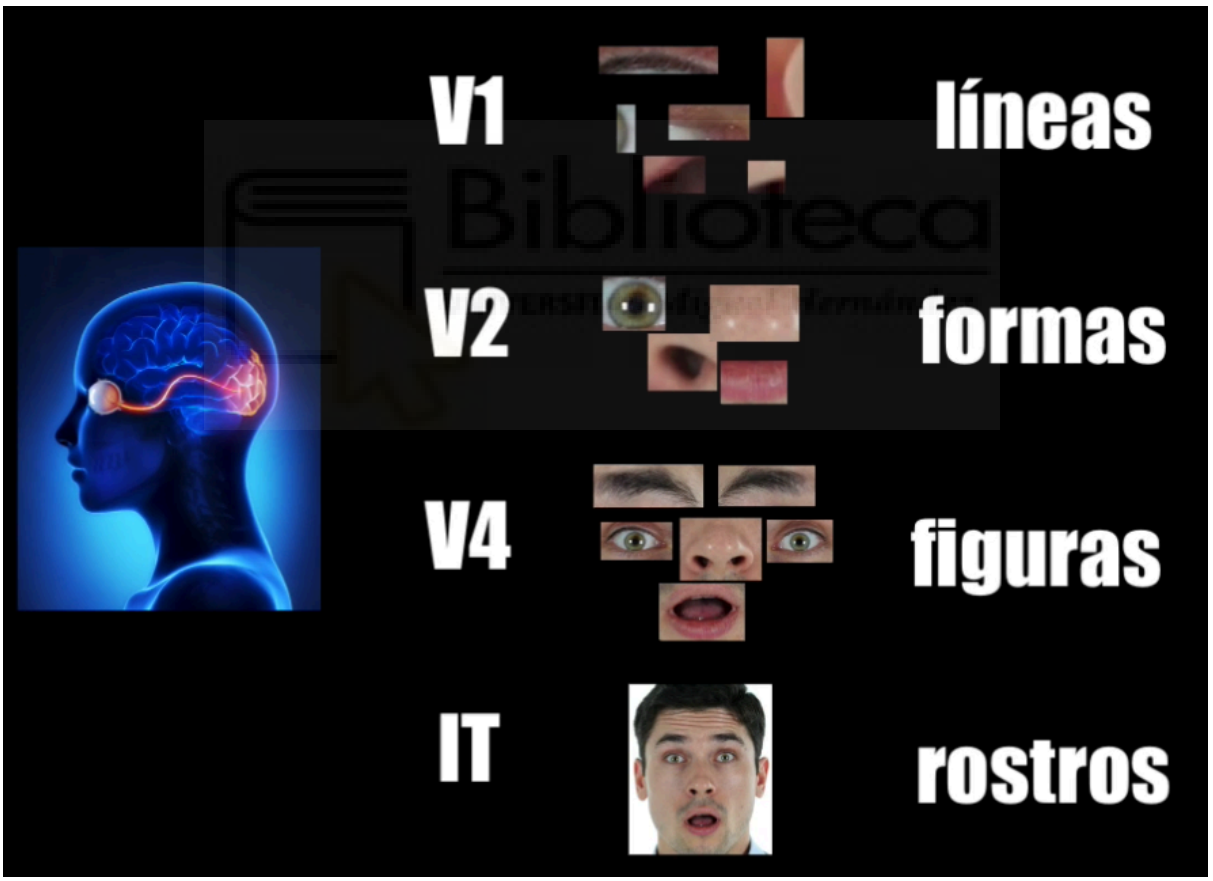


Figura 6: El cerebro humano y las diferentes capas de la corteza visual encargadas del procesamiento de imágenes

Por ejemplo, las neuronas de la capa “V1” están especializadas en detectar patrones muy simples, como líneas o bordes. Posteriormente esa información pasa a la capa “V2”, una capa más compleja y especializada, que es capaz de interconectar esos elementos y de detectar diferentes formas. Y poco a poco las capas se van especializando más y más,

tomando la información de las capas anteriores hasta que se logran detectar características o elementos más complejos dentro de ese objeto que estamos observando.

En la capa “V4” del cerebro las neuronas se especializan en detectar ojos, narices, cejas y boca. Y ya en la región inferior temporal se combina la información proveniente de las capas anteriores y es posible determinar que estos elementos corresponden precisamente a un rostro.

Pues bien, las Redes Convolucionales hacen una tarea similar a la del cerebro humano al momento de procesar una imagen. A la entrada se tiene una imagen en formato digital, obviamente, y las primeras capas de esa Red Convolucional extraen patrones básicos, como líneas y bordes, y poco a poco, a medida que vamos más profundo en esa Red Convolucional, estos elementos básicos se van combinando en formas y en figuras cada vez más complejas hasta que al final la red es capaz de detectar un rostro o capaz de determinar a qué persona corresponde ese rostro que aparece en la imagen.

Principio de funcionamiento de una Red Convolucional

Para entender cómo es que logra hacer esto una Red Convolucional, primero tenemos que entender o recordar cómo es que se representa digitalmente una imagen.

Abajo vemos un ejemplo de una imagen, que para el caso particular contiene el número 8. En el computador esta imagen es representada como una matriz con valores desde 0 hasta 255, correspondientes a cada uno de los pixeles:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	12	0	11	39	137	37	0	152	147	84	0	0	0
0	0	1	0	0	0	41	160	250	255	235	162	255	238	206	11	13	0
0	0	0	16	9	9	150	251	45	21	184	159	154	255	233	40	0	0
10	0	0	0	0	0	145	146	3	10	0	11	124	253	255	107	0	0
0	0	3	0	4	15	236	216	0	0	38	109	247	240	169	0	11	0
1	0	2	0	0	0	253	253	23	62	224	241	255	164	0	5	0	0
6	0	0	4	0	3	252	250	228	255	255	234	112	28	0	2	17	0
0	2	1	4	0	21	255	253	251	255	172	31	8	0	1	0	0	0
0	0	4	0	163	225	251	255	229	120	0	0	0	0	0	11	0	0
0	0	21	162	255	255	254	255	126	6	0	10	14	6	0	0	9	0
3	79	242	255	141	66	255	245	189	7	8	0	0	5	0	0	0	0
26	221	237	98	0	67	251	255	144	0	8	0	0	7	0	0	11	0
125	255	141	0	87	244	255	208	3	0	0	13	0	1	0	1	0	0
145	248	228	116	235	255	141	34	0	11	0	1	0	0	0	1	3	0
85	237	253	246	255	210	21	1	0	1	0	0	6	2	4	0	0	0
6	23	112	157	114	32	0	0	0	0	2	0	8	0	7	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 7: Ejemplo de una imagen digital

Pues la Red Convolucional toma este dato digital de entrada y, progresivamente, a través del entrenamiento de múltiples capas convolucionales (conformadas por filtros que realizan

diferentes operaciones sobre la imagen) se extraen progresivamente características cada vez más complejas de la imagen para lograr su reconocimiento:

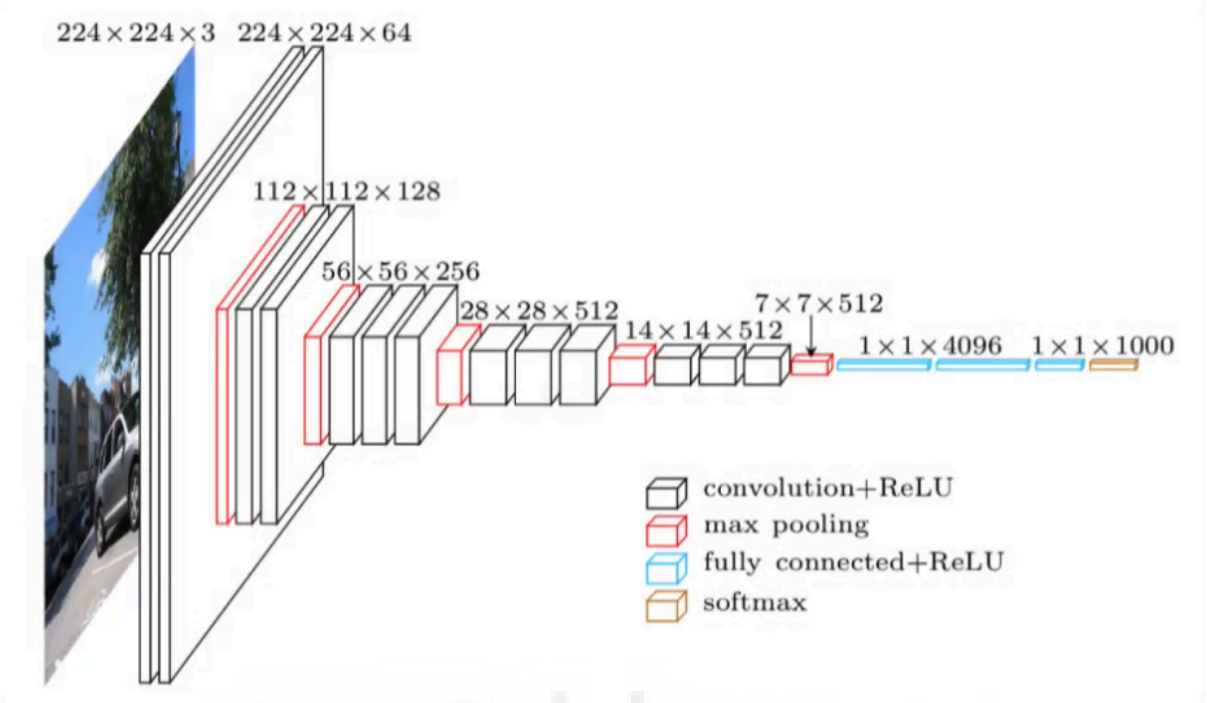


Figura 8: Ejemplo de la estructura de una red convolucional

Así, cuando la imagen es procesada por la red cada bloque de filtros se encargará de extraer diferentes características. En las primeras capas algunos de ellos se encargarán por ejemplo de detectar tonalidades azules, otros de tonalidades verdes, otros por ejemplo podrán detectar líneas rectas y otros podrán detectar tonalidades oscuras.

Posteriormente, la salida de estos filtros se lleva a unas capas más profundas, las que aparecen en color rojo en la imagen anterior, que se llaman max-pooling y que permiten reducir la cantidad de datos, la cantidad de información, y extraer así los datos más representativos de los filtros que se utilizaron anteriormente.

Luego, a medida que vamos más profundo en la red, se repiten estos procedimientos (filtrado y max pooling) y cada vez el tamaño de las imágenes resultantes va siendo más pequeño y la cantidad de filtros usados se va incrementando.

Así, inicialmente se usan 64 filtros, luego 128 y luego 256. Es decir que a medida que nos vamos más profundo en la red estamos extrayendo más y más características de esa imagen de entrada, siendo cada vez más complejas hasta que resulta posible combinarlas en formas que permiten determinar el objeto que se está intentando clasificar.

Generalmente en la etapa final de la Red Convolucional lo que tenemos es una Red Neuronal convencional, con pocas capas, que permite tomar las características extraídas por las capas convolucionales, representarlas como un vector de datos y realizar la clasificación final de la imagen, usando para esto una función “softmax” o sigmoidal.

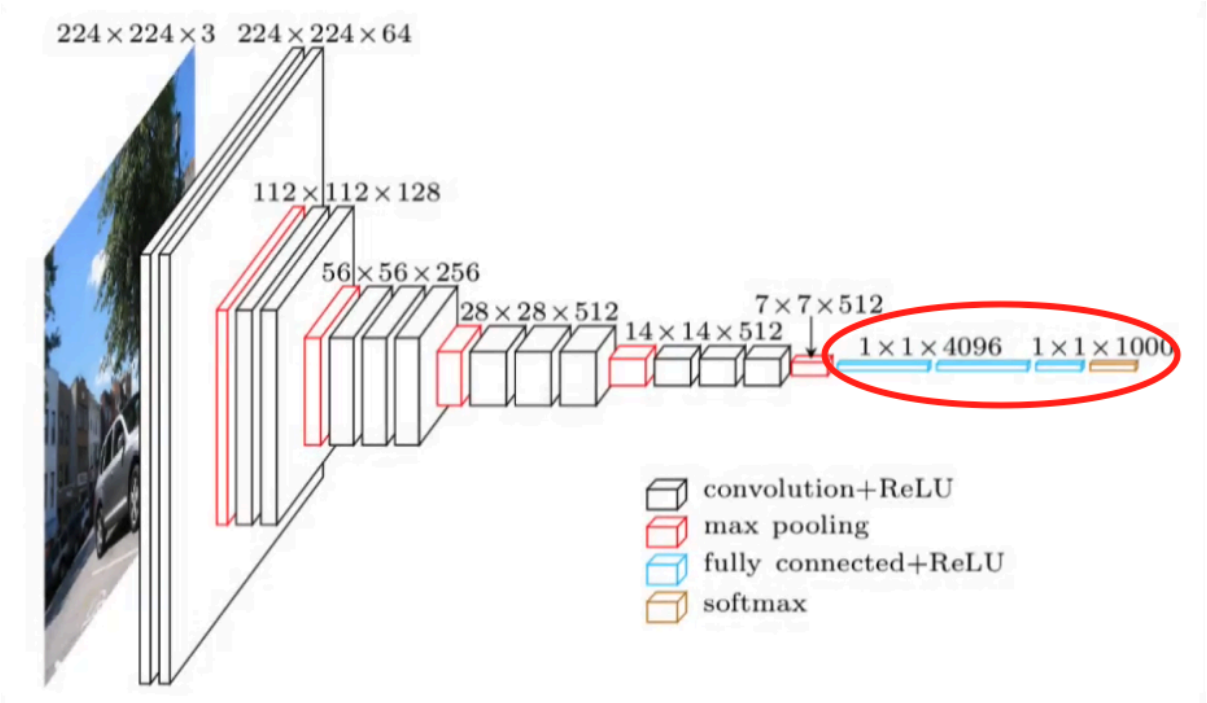


Figura 9: En rojo la capa de salida de una Red Convolucional

2.3 Redes GAN

Las redes adversarias generativas (*GANs*) son un método basado en el entrenamiento de dos redes neuronales, una denominada generadora y otra discriminadora, compitiendo entre sí para generar nuevas instancias que se asemejen a las de la distribución de probabilidad de los datos de entrenamiento.

Las *GANs* tienen una amplia gama de aplicaciones en campos como la visión por computadora, la segmentación semántica, la síntesis de series temporales, la edición de imagen, el procesamiento del lenguaje natural y la generación de imagen a partir de texto, entre otros [4]. Los modelos generativos modelizan la distribución de probabilidad de un conjunto de datos, pero en lugar de proporcionar un valor de probabilidad, generan nuevas instancias cercanas a la distribución original. Las *GANs* utilizan un esquema de aprendizaje que permite codificar los atributos definitorios de la distribución de probabilidad en una red neuronal, lo que permite generar instancias que se asemeje a la distribución de probabilidad original.

La arquitectura *GAN* está formada por dos redes neuronales constituyentes: una denominada discriminadora (D) y otra generadora G. La red G se encarga de generar nuevas instancias del mismo dominio que el del conjunto de datos de origen. La red D se encarga de discriminar si los datos de entrada son reales, esto es pertenecientes al conjunto de datos de entrada o bien son ficticios, esto es generados artificialmente. Ambas redes se entrenan de manera conjunta de manera que G maximice sus posibilidades de no ser detectada por D y D de forma que haga cada vez más sofisticados sus métodos de detección de los datos generados artificialmente por G. Estas dos redes adversarias

compiten en un juego de suma cero en el que se hipotetiza que eventualmente llegan a un equilibrio de Nash.

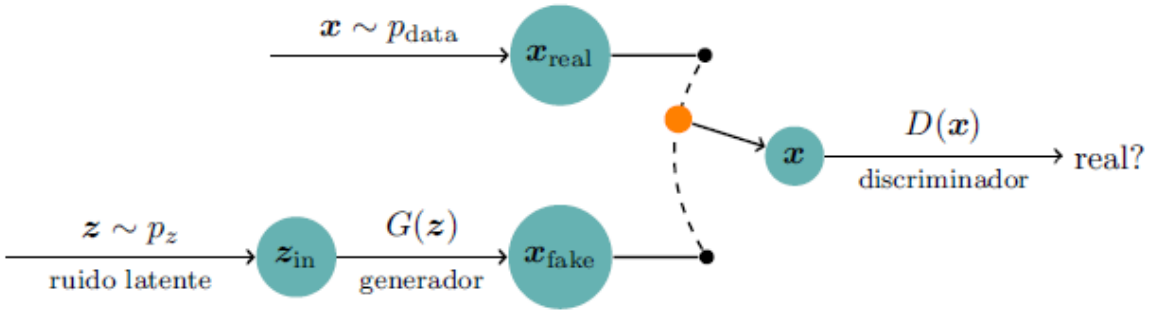


Figura 10: Diagrama representativo del proceso de entrenamiento de las redes adversarias generativas (*GANs*)

En la figura 10 se muestra un diagrama representativo del proceso de optimización de las *GAN*. Un vector z es muestreo de una distribución de probabilidad aleatoria pz, z ~ pz y alimentado como entrada a G. El propósito de la optimización es conseguir que G(z) ~ pg acabe siendo una estimación de la distribución de probabilidad Pdata. Las *GAN* se optimizan con la función min-max de un juego de suma cero expresado por la ecuación 1.

$$min_G max_D E_{x\sim pr} log[D(x)] + E_{z\sim pz} log[1 - D(G(z))] \qquad (1)$$

Equivalentemente, sean pθ , Dω las redes neuronales generadora y discriminadora de una GAN, siendo θ los parámetros de G y ω los de D. Ambas redes se optimizan en conjunto con la función objetivo definida por la ecuación 2.

$$min_\theta max_\omega E_{x\sim Q} log[D_\omega(x)] + E_{x\sim p\theta} log[1 - D_\omega(x))] \qquad (2)$$

Durante el proceso de optimización la red D recibirá como entrada de manera aleatoria datos pertenecientes al conjunto de datos y otros procedentes de la red G. Se optimizará su funcionamiento para que su discriminación sea efectiva (ecuación 3).

$$\nabla_{\theta D} \frac{1}{m} \sum_{i=1}^{m} [logD(x^{(i)}) + log(1 - D(G(z^{(i)})))] \qquad (3)$$

Al mismo tiempo, cuando D reciba una entrada procedente de G, éste se optimizará para mejorar sus predicciones y hacer cada vez más difícil el papel de D. Esto únicamente se puede conseguir mejorando la calidad de los datos generados y haciéndolos más parecidos al conjunto de datos original (ecuación 4).

$$\nabla_{\theta G} \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(z^{(i)}))) \qquad (4)$$

Se establece así una competición entre las dos redes (de ahí el nombre de adversarias) de forma que idealmente en el progreso de este proceso ambas mejoran su funcionamiento al punto que idealmente el generador acaba produciendo datos cada vez más parecidos a los del conjunto de datos original.

La competencia entre estos dos modelos [8] se puede ver a través de una analogía: el Generador es como un falsificador, que intenta producir billetes falsos sin que estos sean detectados, mientras que el Discriminador es como el policía, que intenta detectar estos billetes falsos. Esta competición lleva a ambos equipos a mejorar sus métodos, siendo el objetivo final que el falsificador gane este juego y engañe al policía.

Estructura

En primer lugar crearemos el Discriminador para que, por ejemplo, sea capaz de reconocer rostros humanos. Este discriminador será simplemente un clasificador, como una Red Convolucional. De manera que, idealmente, si ingresamos una imagen con un rostro humano, la salida generada por el discriminador será igual a 1, mientras que si ingresamos otra imagen diferente, la salida será 0:

Figura 11: Discriminador

Ahora creamos el segundo modelo, el Generador y el objetivo será entrenarlo para que sea capaz de tomar una entrada aleatoria y a la salida generar algo muy parecido a una imagen de un rostro:

Figura 12: Generador

Pues juntando ambos modelos combinados obtendremos una Red Adversaria Generativa:

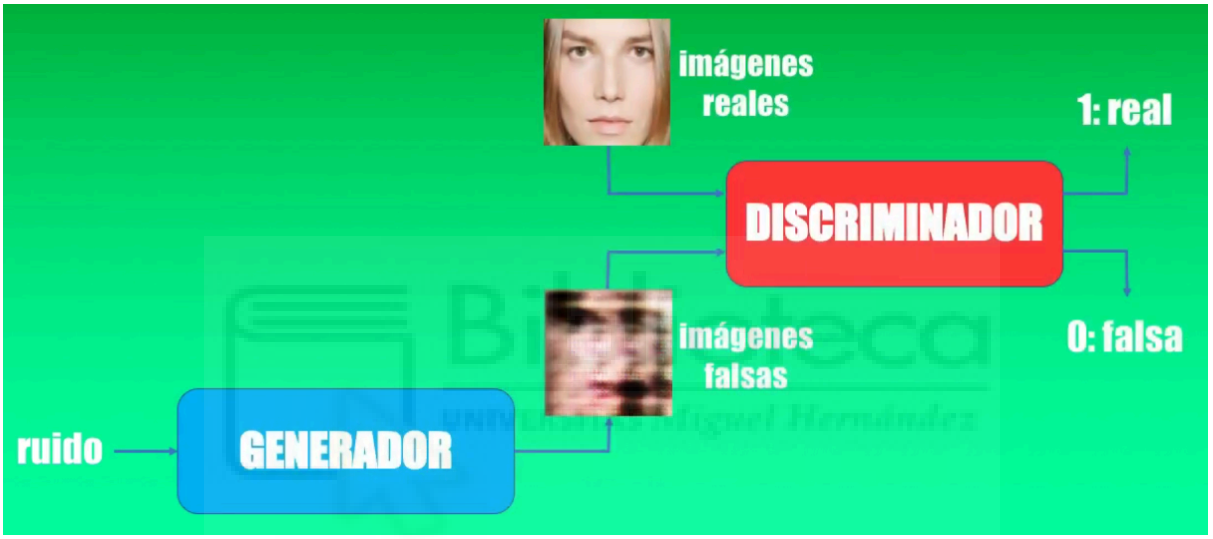


Figura 13: Estructura red GAN

La idea es entrenar estos dos modelos simultáneamente buscando que al final sea el Generador el vencedor en esta competición, como hemos comentado anteriormente.

¿Cómo saber si el entrenamiento es adecuado? En ese caso debemos analizar el error del Discriminador, tanto con imágenes reales como con imágenes falsas, obtenidas con el Generador.

Al inicio del entrenamiento es de esperar que las imágenes generadas no sean parecidas a un rostro por lo que el Discriminador será capaz de descartarlas fácilmente, así que el error en uno y otro caso será muy pequeño.

Sin embargo, a medida que avanza el proceso de entrenamiento, el Generador aprenderá poco a poco a producir imágenes cada vez más parecidas a un rostro humano, por lo que logrará que el error del discriminador aumente. Idealmente la salida del Discriminador será de un 0.5, lo que querrá decir que habrá sido engañado por completo por el Generador.

Por su parte, en ese caso podemos concluir que el Generador ha aprendido la distribución de los datos de entrada y por tanto ha aprendido a replicar con precisión esta distribución:

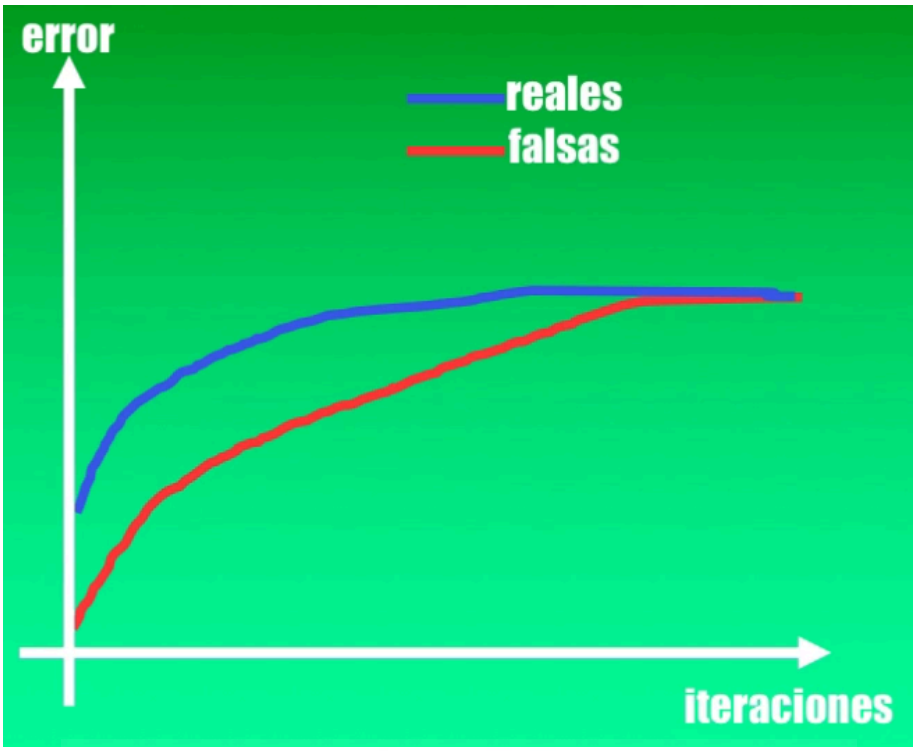


Figura 14: error ideal del Discriminador

Ventajas

Desde su introducción en 2014, las *GANs* han despertado un gran interés sobre todo en el campo de la generación de imagen. Esto ha sido debido a que presentan una serie de ventajas sobre el otro paradigma dominante hasta el momento en lo que a modelos generativos se refiere, los VAEs. Dichas ventajas son las siguientes:

• Imágenes más nítidas: Las *GANs* producen imágenes más nítidas que otros modelos generativos disponibles hasta el momento. Los modelos de difusión que veremos más adelante son una excepción posterior en este aspecto.

• Tamaño configurable: El tamaño de la variable aleatoria no está restringido pudiéndose enriquecer en caso de ser necesario.

• Generador versátil: El paradigma de diseño basado en *GANs* soporta distintos tipos de funciones generadoras, a diferencia de otros modelos generativos que pueden tener restricciones debido a su arquitectura. Los *VAEs*, por ejemplo, obligan a utilizar una función Gaussiana en la primera capa del Generador.

La arquitectura también tiene sus desventajas entre las que están las siguientes:
• Colapso de modo: Durante el entrenamiento sincronizado de generador y discriminador, el generador puede tener tendencia a reproducir únicamente un modo específico que es capaz de burlar al discriminador. A pesar de que este patrón puede estar minimizando la función objetivo, lo hace sin cubrir todo el dominio del conjunto de datos.

• Desvanecimiento de gradientes: A veces el discriminador se optimiza demasiado rápido en su función. En estos casos, los gradientes que propaga pueden ser demasiado bajos para asegurar la optimización del generador.

• Inestabilidad: A menudo durante el entrenamiento los parámetros de ambas redes actúan sin encontrar un punto de equilibrio. En estas circunstancias el generador tiene dificultades en encontrar un punto que genere imágenes de alta calidad.

Aplicaciones

Las *GANs* se han utilizado para una variedad de aplicaciones, incluyendo el aprendizaje automático, el procesamiento de imágenes, el procesamiento del lenguaje natural y la generación de contenido [9].

- Aprendizaje automático: las *GANs* se han utilizado para mejorar el rendimiento de los algoritmos de aprendizaje automático. Esto se logra mediante la creación de datos sintéticos para entrenar los algoritmos de aprendizaje automático. Estos datos sintéticos pueden ayudar a los algoritmos a mejorar su rendimiento al proporcionar una mayor cantidad de datos para entrenar.

- Procesamiento de imágenes: las *GANs* también se han utilizado para mejorar la calidad de las imágenes. Esto se logra mediante el uso de GANs para generar contenido realista para rellenar los huecos en las imágenes. Esto puede ayudar a mejorar el enfoque, el brillo y otros aspectos de la imagen.

- Procesamiento del lenguaje natural: las *GANs* también se han utilizado para mejorar la precisión del procesamiento del lenguaje natural. Esto se logra mediante el uso de GANs para generar contenido realista para entrenar los modelos de lenguaje natural. Esto puede ayudar a los modelos a mejorar su precisión al proporcionar una mayor cantidad de datos para entrenar.

- Generación de contenido: las *GANs* también se han utilizado para generar contenido realista a partir de datos de entrada. Esto se logra mediante el uso de *GANs* para generar contenido visual realista, como imágenes de personas, animales y paisajes. Esta tecnología también se ha utilizado para generar contenido textual, como poesía, historias cortas y discursos.

Figura 15: Ejemplos de resultados obtenidos con estas redes [10]

Alternativas valoradas

Conditional Gan

CGAN modifica el método original introduciendo una entrada adicional tanto en el generador como en el discriminador [11]. Esta entrada adicional sirve como condicionante para ambas funciones. Esta nueva información y se fusiona en el generador con el muestreo de la variable aleatoria z para posteriormente generar la nuevas instancias. Lo mismo ocurre en el discriminador donde y se integra con los datos x a analizar. La nueva función de optimización queda como indica la ecuación 5.

$$min_G max_D E_{x\sim pr} log[D(x|y)] + E_{z\sim pz} log[1 - D(G(z|y))] \quad (5)$$

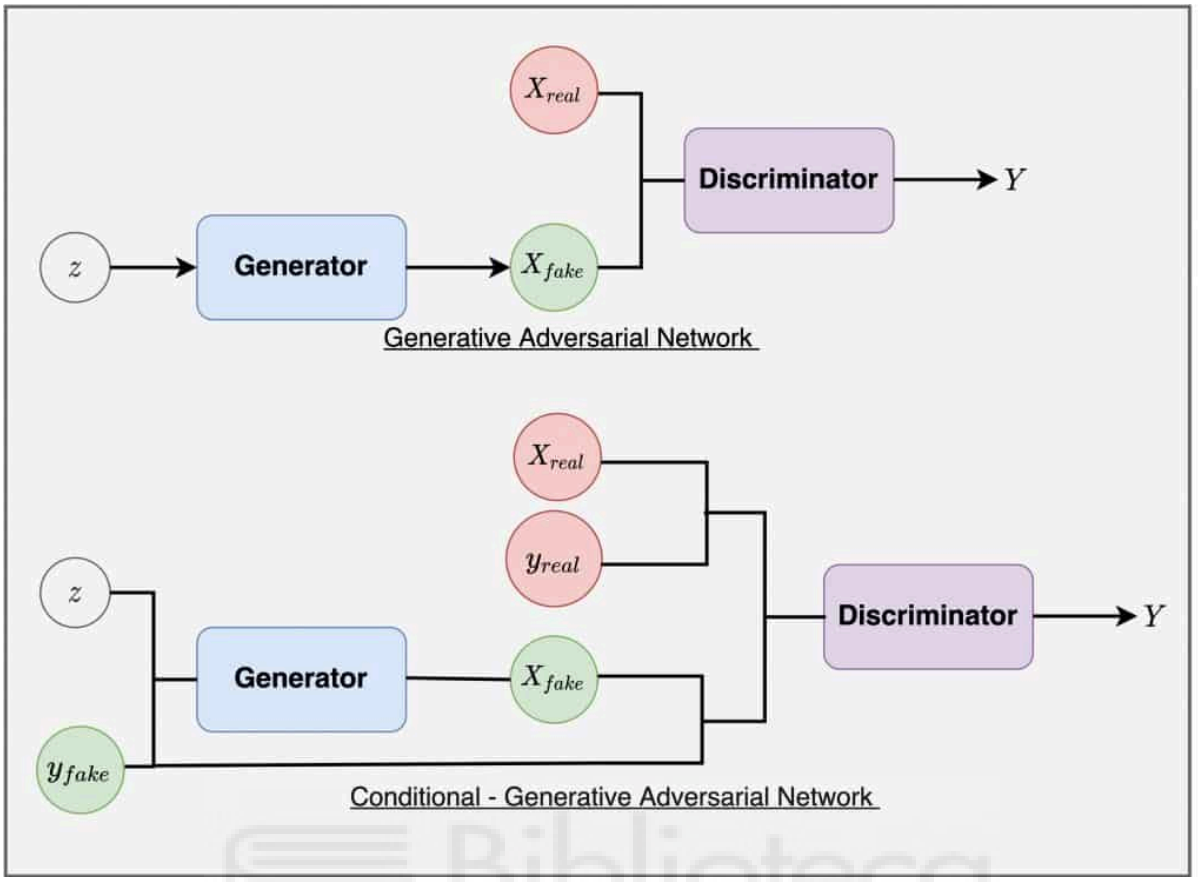


Figura 16: Arquitectura *GAN* vs Arquitectura de *CGan* [12]

Estas redes permiten introducir condiciones en la generación de datos. Por ejemplo, en lugar de generar una imagen cualquiera, el generador puede crear una imagen de un objeto específico, como un coche o una persona con gafas [13]. Por lo que pueden generar imágenes diversas y realistas que reflejan el contenido semántico de la entrada de texto.

Red antagónica generativa convolucional profunda (*DCGAN*)

Las redes antagónicas generativas convolucionales profundas (*DCGAN*) fueron introducidas por primera vez como método para la generación de imágenes. Utilizan convoluciones en el discriminador y convoluciones traspuestas en el generador. Además de mejoras en la resolución de las imágenes generadas, consiguen mejoras en la estabilidad del entrenamiento que en su estudio atribuyen a la introducción de las siguientes modificaciones:

• Sustitución de todas las capas de pooling de las dos redes. En el discriminador se utilizan núcleos con stride mayor que 1 y en el generador convoluciones traspuestas para aumentar el tamaño de la imagen.

• Uso de la normalización por lotes en las dos redes.

• En el discriminador se cambia la función de activación de ReLU a *LeakyReLU* [14]. En el generador se utilizan ReLU en todas las capas excepto en la última donde se usa la función de activación tangente hiperbólica (tanh).

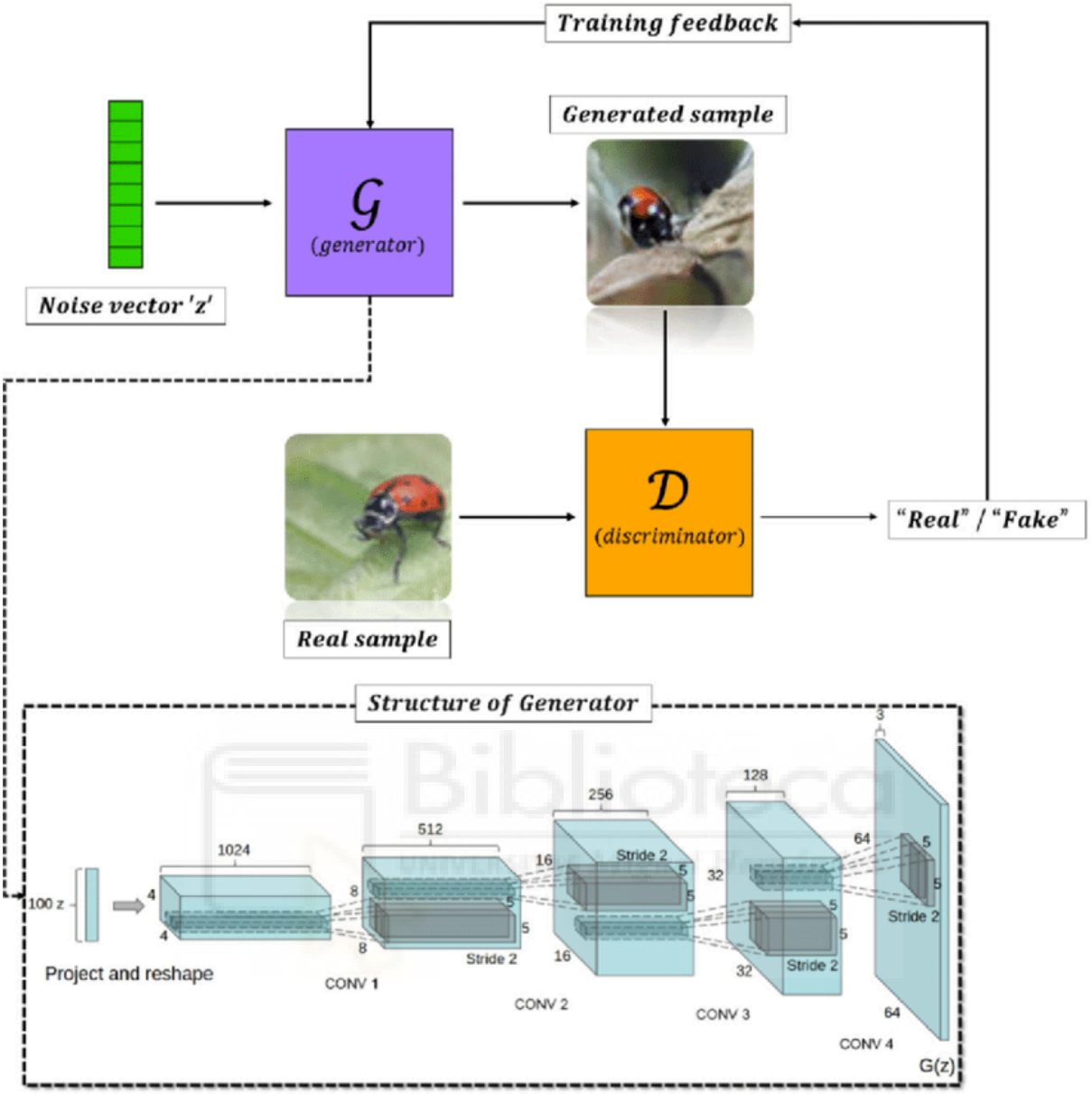


Figura 17: Arquitectura DCGAN [15]

2.4 CycleGan

La traducción de imagen a imagen [16] es una clase de problema de visión donde el objetivo es aprender el mapeo entre una imagen de entrada y una de salida utilizando un conjunto de entrenamiento de pares de imágenes alineadas. Sin embargo, para muchas tareas, los datos de entrenamiento emparejados no estarán disponibles. Es por ello que es necesario presentar un enfoque para aprender a traducir una imagen desde un dominio de origen X a un dominio objetivo Y. De manera que se pueda mapear $G: X \rightarrow Y$ tal que la distribución de imágenes de G(X) es indistinguible de la distribución Y utilizando una pérdida adversarial.

Una de las ventajas que presenta esta red es que le podemos distribuir datos de entrenamiento no emparejados.

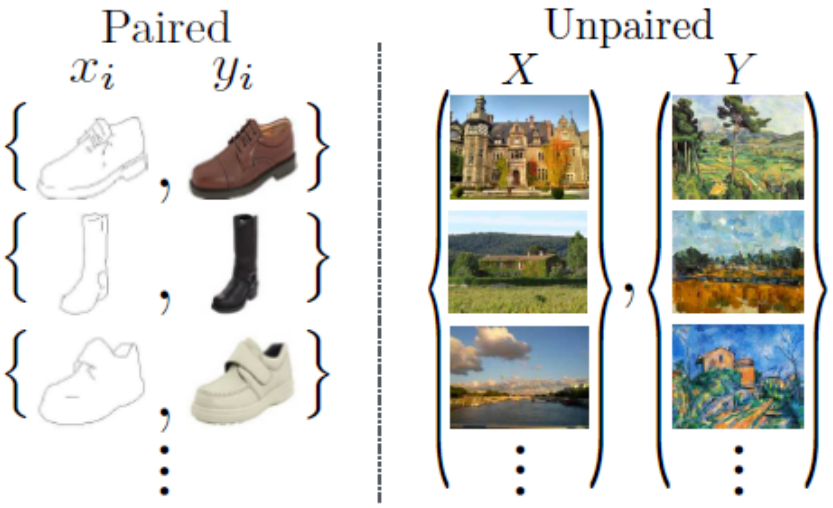


Figura 18: ejemplo de datos emparejados y no emparejados

Los datos de entrenamiento emparejados (izquierda) consisten en ejemplos de entrenamiento $\{x_i, y_i\}_{i=1}^{N}$, donde se da el y_i que corresponde a cada x_i. En datos de entrenamiento no emparejados (derecha), consistentes en un conjunto fuente $\{x_i\}_{i=1}^{N} \epsilon X$ y un conjunto objetivo $\{y_j\}_{j=1}^{M} \epsilon Y$, sin información sobre qué x_i corresponde a qué y_j.

2.4.1 Estructura y funcionamiento

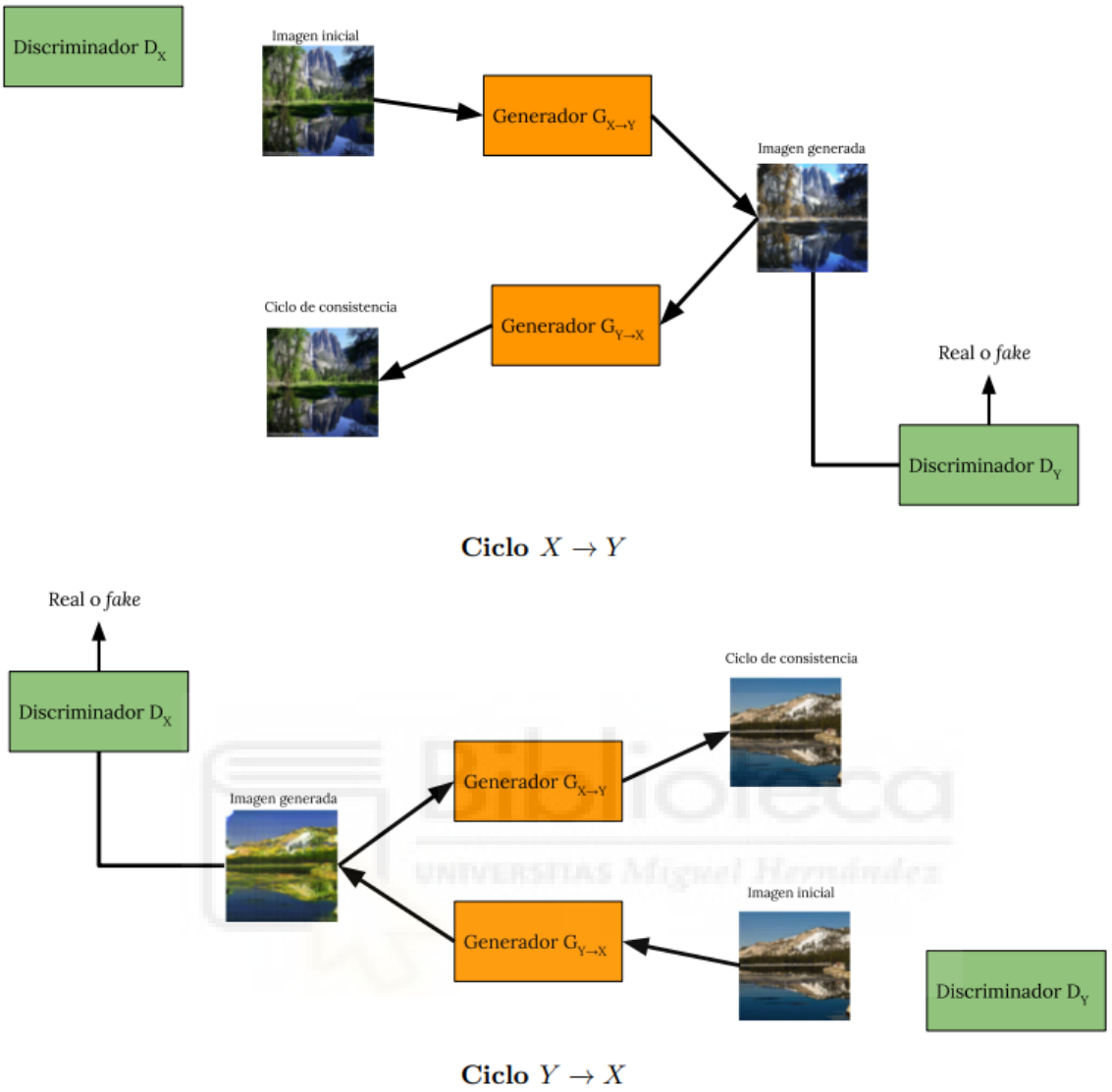


Figura 19: Esquema cyclegan

Como podemos ver en la figura 19, la red generativa antagónica está formada por 2 sistemas GAN, cada uno con su respectivo generador y discriminador, siendo el primer generador el encargado de convertir imágenes del dominio X al dominio Y, y el segundo generador convertirlo del Y al X.

La idea es que a cada generador se le asigna un discriminador encargado de detectar si las imágenes que les entran son reales o generadas y así proporcionar la información necesaria para el entrenamiento, tal como sucedería en una red convolucional, pero en este caso, eso no es suficiente para pasar de un dominio a otro, puesto que no garantiza que la imagen convertida conserve la identidad de la original.

Es aquí donde entran las distintas funciones de pérdida que veremos en los puntos 2.4.2 y 2.4.3.

Arquitectura Generador

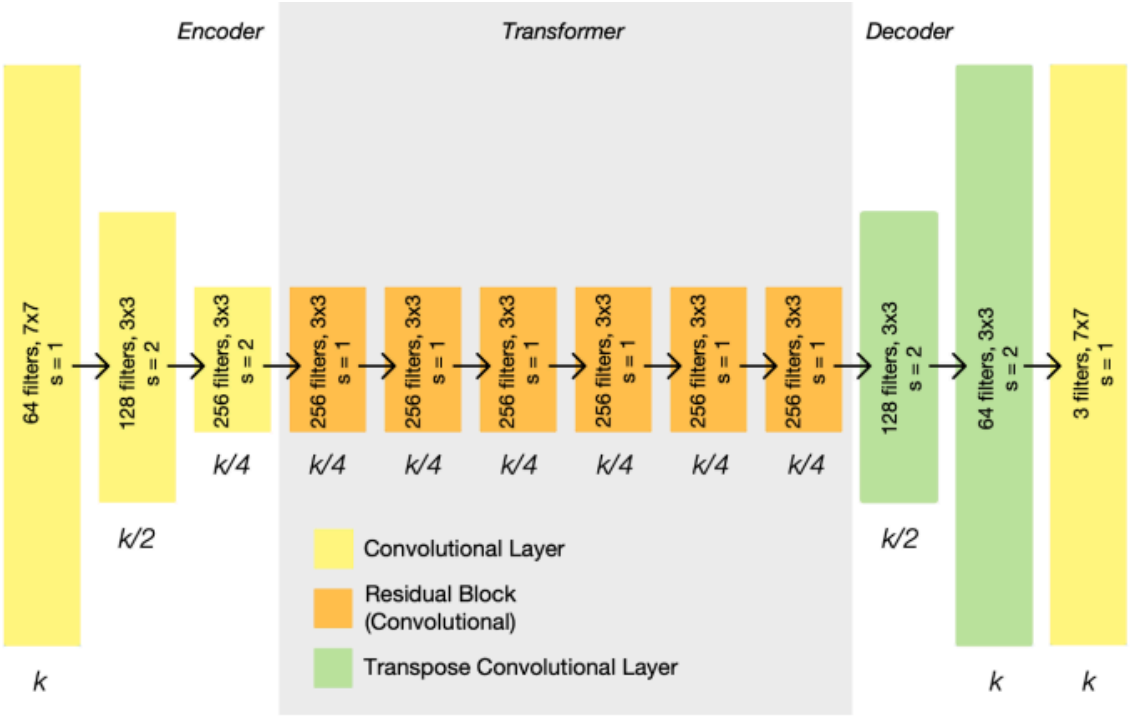


Figura 20: estructura del generador

En la Figura 20 se muestra la estructura encoder − decoder del generador donde el primer elemento se basa en una estructura de capas que produce una reducción de la dimensionalidad espacial de los datos de entrada mediante una serie de convoluciones seguidas de una capa de *Instance normalization* con activación *Relu*. A continuación, los mapas de características obtenidos en el encoder son pasados a través de una serie de bloques residuales que realiza una serie de transformaciones sobre estos. Por último, tras las transformaciones realizadas, el decoder realiza la restitución a la dimensionalidad de entrada mediante una serie de capas convolucionales transpuestas seguidas de una capa de *Instance normalization* con activación *Relu*.

La *Instance normalization* o normalización de instancias [17] (también conocida como normalización de contraste) es una capa de normalización en la que:

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - \mu_{ti})^2.$$

Esto evita el cambio de media y covarianza específica de cada instancia, simplificando el proceso de aprendizaje. Intuitivamente, el proceso de normalización permite eliminar la información de contraste específica de cada instancia de la imagen de contenido en una tarea como la estilización de imágenes, lo que simplifica la generación.

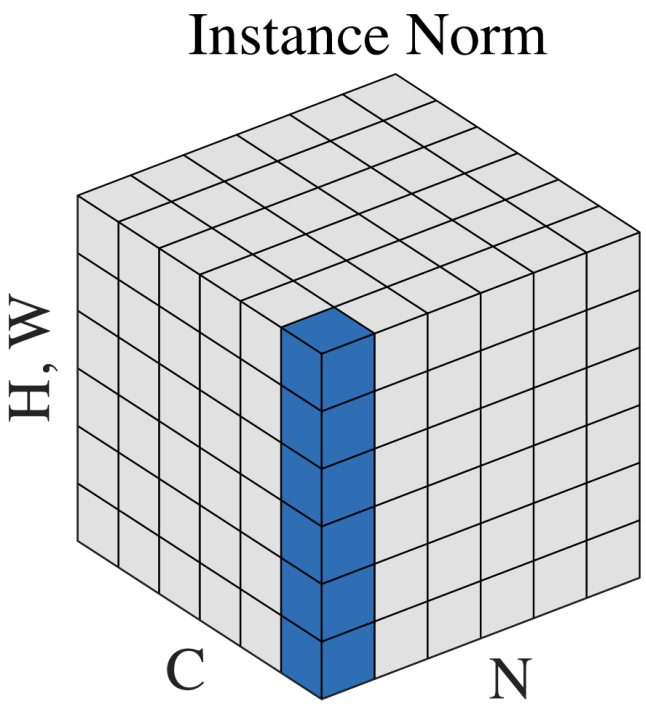


Figura 21: Ejemplo visual de lo que hace *Instance normalization*

Las unidades lineales rectificadas, o *ReLU* [18], son un tipo de función de activación que son lineales en la dimensión positiva, pero nulas en la dimensión negativa. El pliegue de la función es la fuente de la no linealidad. La linealidad en la dimensión positiva tiene la atractiva propiedad de que evita la no saturación de los gradientes (a diferencia de las activaciones sigmoidales), aunque para la mitad de la línea real su gradiente es cero.

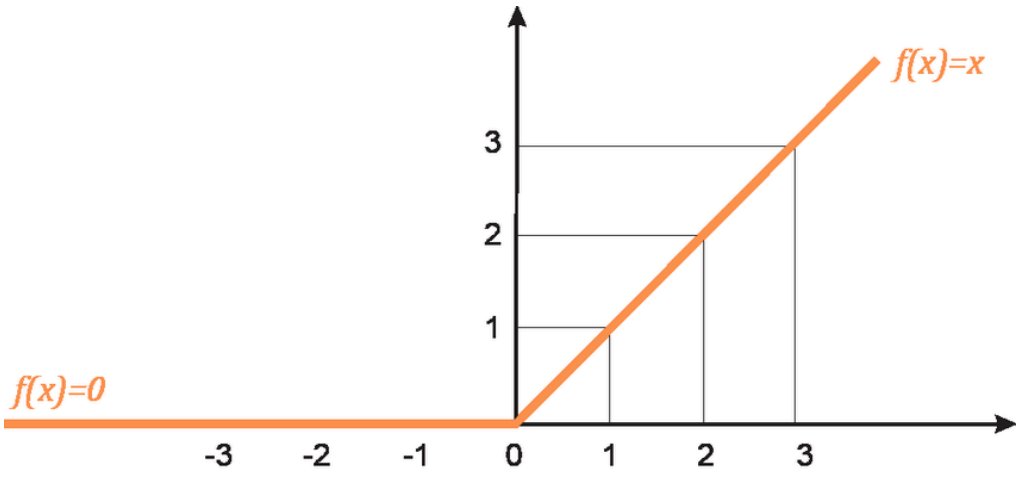


Figura 22: Curva de la función *ReLU*

Arquitectura Discriminador

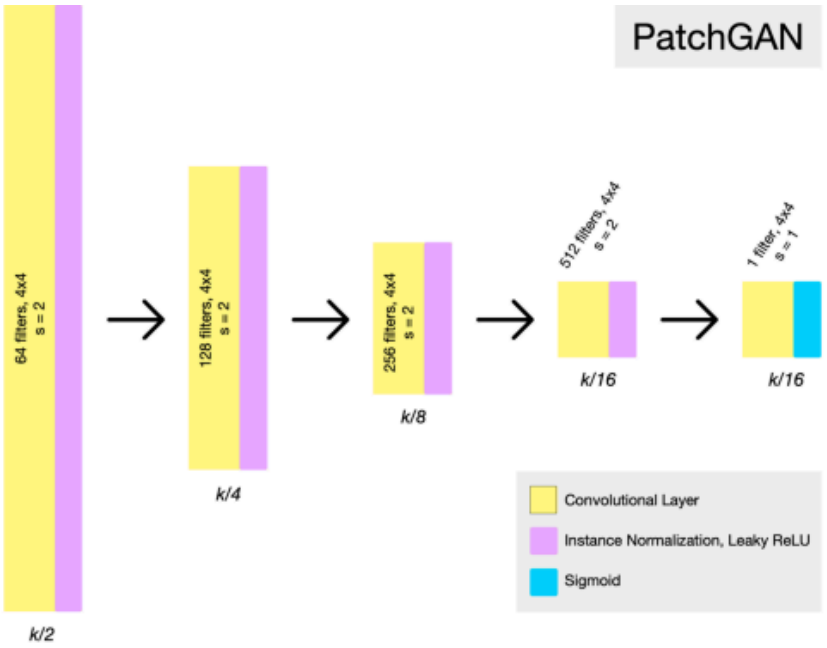


Figura 23: Estructura del discriminador

Para las redes discriminantes utilizamos la arquitectura de la red *Patch-GAN* [19], cuyo objetivo es clasificar si las imágenes superpuestas de 70 × 70 son reales o falsas. Esta arquitectura discriminatoria a nivel de parche tiene menos parámetros que un *full image* discriminador y puede aplicarse a imágenes de tamaño arbitrario de forma totalmente convolucional. Es un tipo de discriminador de redes generativas antagónicas que genera una matriz de valores entre 0 y 1 (falso; real) en lugar de uno solo [20]. Por tanto, este discriminador clasificará si cada parche en una imagen es real o falso en lugar de calificar toda la imagen. Se trata de una red totalmente convolucional, que toma una imagen y produce una matriz de probabilidades, cada una de las cuales se refiere a la probabilidad de que el «fragmento» correspondiente de la imagen sea «real» (en contraposición a una imagen generada). El tamaño de la representación que produce cada capa se indica debajo, en función del tamaño de la imagen de entrada, k. En cada capa se indica el número de filtros, el tamaño de esos filtros y el intervalo.

El *Leaky Rectified Linear Unit*, o *Leaky ReLU* [21], es un tipo de función de activación basada en una *ReLU*, pero tiene una pequeña pendiente para los valores negativos en lugar de una pendiente plana. El coeficiente de pendiente se determina antes del entrenamiento, es decir, no se aprende durante el entrenamiento. Este tipo de función de activación es popular en tareas en las que podemos sufrir gradientes dispersos, por ejemplo entrenando redes generativas adversariales.

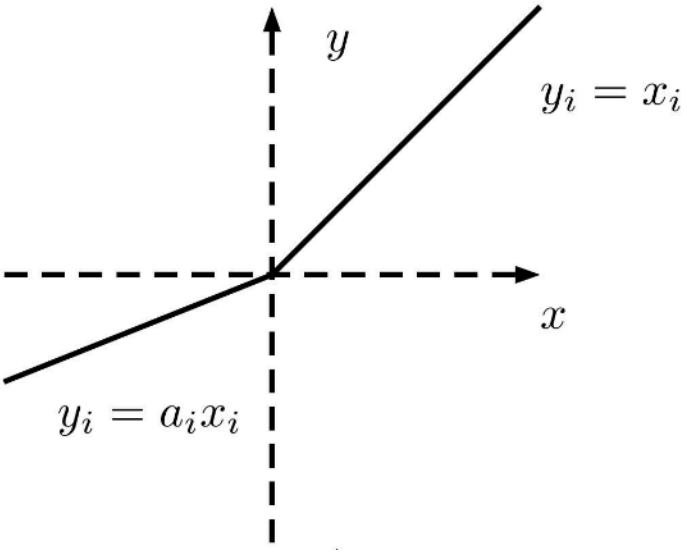


Figura 24: Curva de la función *Leaky ReLU*

La función sigmoide [22] se utiliza comúnmente en las redes neuronales como una función de activación para las capas ocultas. Esta función toma un valor de entrada y lo comprime en un rango comprendido entre 0 y 1, lo que la convierte en una función útil para modelar probabilidades y realizar clasificaciones binarias. La función sigmoide tiene una forma característica de «S» que le da su nombre.

$$f(x) = \frac{1}{1 + e^{-x}}$$

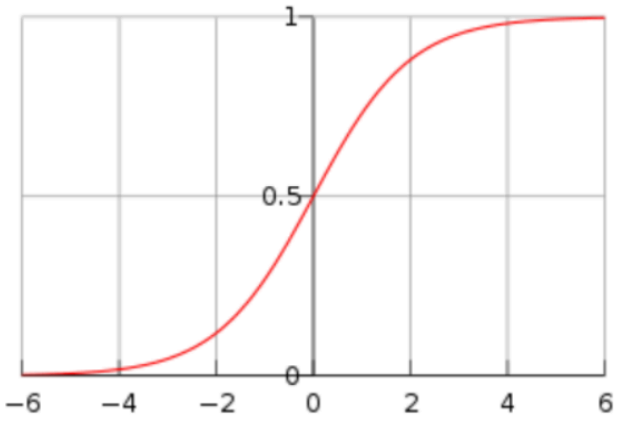


Figura 25: Curva de la función sigmoidal

2.4.2 *Adversarial Loss*

La *adversarial loss* mide la capacidad del generador para crear imágenes "creíbles" y del discriminador para detectar imágenes falsas. De modo que aplicamos pérdidas adversarias a ambas funciones de mapeo.

Para la función de mapeo G: $X \rightarrow Y$ y su discriminador DY , expresamos el objetivo como:

$$L_{GAN}(G, D_Y, X, Y) = E_{y \sim pdata(y)}[logD_Y(y)] + E_{x \sim pdata(x)}[1 - logD_Y(G(x))] \qquad (1)$$

donde G intenta generar imágenes G(x) que se parezca a imágenes del dominio Y , mientras que D_Y intenta distinguir entre muestras traducidas $G(x)$ y las muestras reales y.

Introducimos una pérdida adversarial similar para la función de mapeo F: $Y \rightarrow X$ y su discriminador D_x, es decir, $L_{GAN}(F, D_x, Y, X)$.

En CycleGAN, la *adversarial loss* permite que los generadores aprendan a crear imágenes que se parezcan mucho a las imágenes reales en el dominio de destino. Sin esta pérdida, los generadores no tendrían un incentivo para mejorar su capacidad de producir imágenes que "engañen" a los discriminadores, lo que es crucial para la calidad visual de las imágenes generadas.

2.4.3 *Cycle Consistency Loss*

La *cycle consistency loss* se introduce en CycleGAN para garantizar que las transformaciones entre dos dominios de imágenes sean coherentes. La idea clave es que si conviertes una imagen de un dominio a otro, luego deberías poder revertirla al dominio original sin perder mucha información.

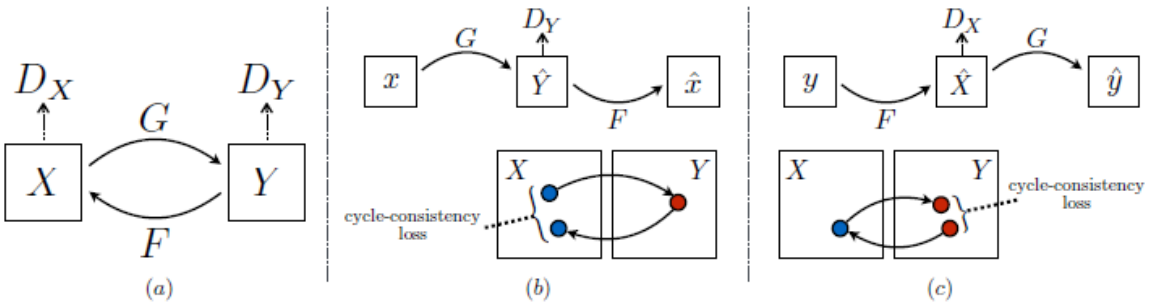


Figura 26: Cycle-consistency loss

(a) Nuestro modelo contiene dos funciones de asignación G: $X \rightarrow Y$ y F: $Y \rightarrow X$, y discriminadores adversariales asociados DY y DX. DY anima a G a traducir X en salidas indistinguibles del dominio Y, y viceversa para DX, F y X. Para regularizar aún más los mapeos, introducimos dos «pérdidas de consistencia cíclica» que capturan la intuición de que si traducimos de un dominio a otro y viceversa, deberíamos llegar al punto de partida.
Donde empezamos: (b) Pérdida de consistencia de ciclo hacia adelante: $x \rightarrow G(x) \rightarrow F(G(x)) \simeq x$, y (c) pérdida de consistencia cíclica hacia atrás: $y \rightarrow F(y) \rightarrow G(F(y)) \simeq y$.

El entrenamiento adversarial puede, en teoría, aprender mapeos G y F que producen salidas idénticamente distribuidas como dominios objetivo Y y X respectivamente (en sentido

estricto, esto requiere que G y F sean funciones estocásticas). Sin embargo, con la suficiente capacidad, una red puede asignar el mismo conjunto de imágenes en cualquier permutación aleatoria de imágenes en el dominio objetivo, donde cualquiera de los mapeos aprendidos puede inducir una distribución de salida que coincida con la distribución objetivo. Para reducir aún más el espacio de posibles funciones de mapeo, argumentamos que las funciones de asignación aprendidas deben ser coherentes con el ciclo: como
Figura 26 (b), para cada imagen x del dominio X, el ciclo de traslación de la imagen debe ser capaz de devolver x a la imagen original, es decir, $x \rightarrow G(x) \rightarrow F(G(x)) \simeq x$. A esto lo llamamos coherencia de ciclo hacia delante. Del mismo modo, como se ilustra en la figura 26 (c), para cada imagen y del dominio Y , G y F también deben satisfacer consistencia de ciclo hacia atrás: $y \rightarrow F(y) \rightarrow G(F(y)) \simeq y$. Podemos incentivar este comportamiento utilizando una consistencia de ciclo:

$$L_{cyc}(G, F) = E_{x \sim pdata(x)}[||F(G(x)) - x||_1] + E_{y \sim pdata(y)}[||G(F(y)) - y||_1]. \quad (2)$$

Por tanto, la *cycle consistency loss* fuerza a los generadores a aprender a realizar transformaciones que no destruyan completamente la información de la imagen original, haciendo que las imágenes sean reversibles y, por lo tanto, más significativas. En resumen, ayuda a evitar que las transformaciones sean triviales o irrelevantes, y fomenta que se conserven los detalles estructurales de las imágenes.

Para ver un poco más claro su importancia, supongamos que queremos transformar imágenes de caballos a cebras. La adversarial loss por sí sola haría que el generador G tratara de hacer imágenes que se vean como cebras, pero sin garantizar que la "esencia" del caballo (como la estructura del cuerpo, la postura, etc.) se mantenga en la imagen generada. Es posible que G cree imágenes que sean incorrectas o irrelevantes, siempre que logren engañar al discriminador Dy.

Aquí es donde entra en juego la *cycle consistency loss*. Al hacer que la transformación de ida y vuelta (de caballo a cebra y luego de nuevo a caballo) produzca la misma imagen inicial, se garantiza que el generador G y el generador F están preservando las características clave de la imagen, mientras realizan las transformaciones.

Beneficios de la cycle-consistency loss:

- Mantiene la coherencia estructural: La *cycle consistency loss* asegura que las transformaciones no cambien radicalmente la estructura general de las imágenes.

- Evita colapsos de modo: Previene que los generadores simplemente produzcan un conjunto limitado de imágenes irrelevantes (un problema común en las GANs).

- Facilita la traducción no pareada: Permite que la red funcione sin tener imágenes pareadas (no necesitas saber qué cebra corresponde a qué caballo).

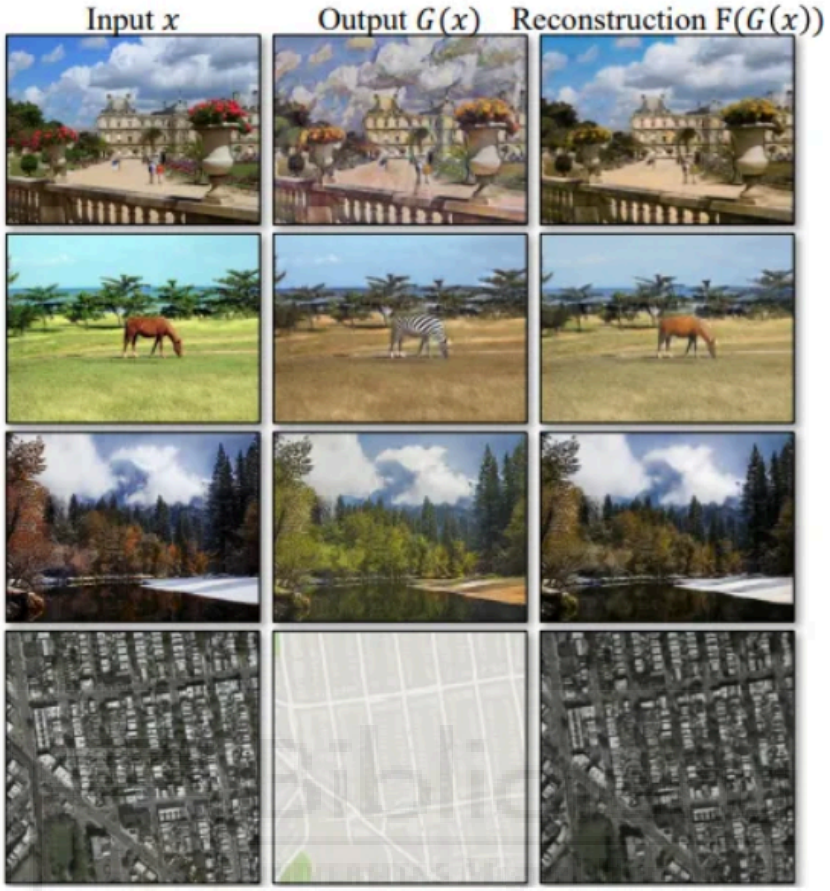


Figura 27: ejemplos del *Cycle consistency*

2.4.4 Resultados y limitaciones

Como vemos a continuación, los resultados obtenidos con esta red son muy buenos, demostrando que consigue a la perfección el cambio de dominio de una imagen a otra.

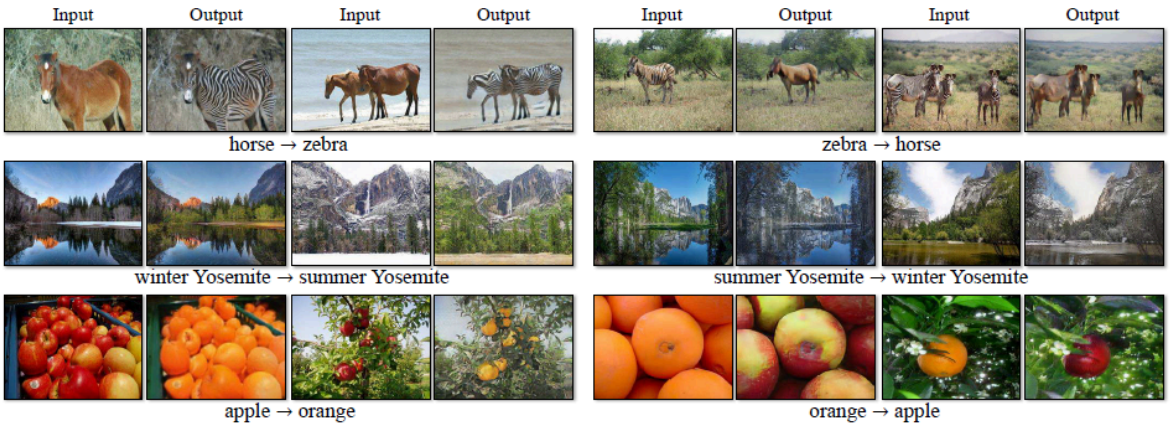


Figura 28: Ejemplos de resultados de red CycleGAN

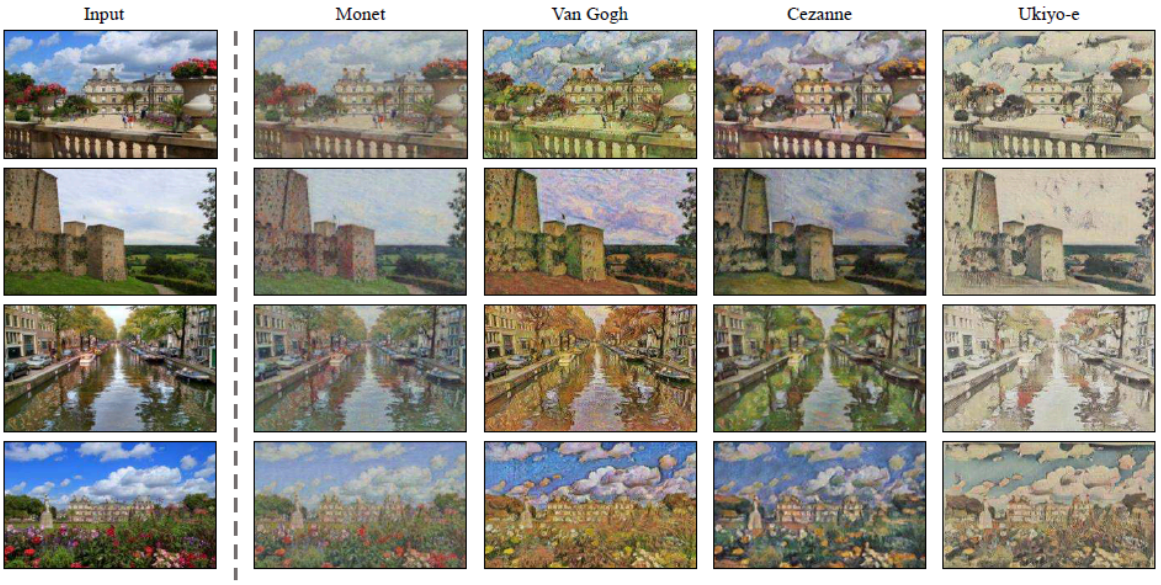


Figura 29: más ejemplos de lo que puede conseguir esta red. En este caso transforma fotografías en distintos estilos de pintura.

Aunque este método puede lograr resultados convincentes en muchos casos, los resultados distan mucho de ser uniformemente positivos. Varios casos típicos de fallo se muestran en la figura 30 . En las tareas de traducción que implican cambios de color y textura, como las vistas anteriormente, el método suele tener éxito. También hemos hecho tareas que requieren cambios geométricos, con poco éxito. Por ejemplo, en la tarea de transfiguración perro→gato, la traducción aprendida degenera en cambios mínimos en la entrada (Figura 27).

El manejo de transformaciones más variadas y extremas, especialmente cambios geométricos, es un problema importante para el trabajo futuro. Algunos fallos se deben a las características de distribución de los conjuntos de datos de entrenamiento. Por ejemplo, la tarea caballo → cebra de la figura 30 ha fallado por completo, porque nuestro modelo se entrenó con los synsets caballo salvaje, cebra de ImageNet, que no contiene imágenes de una persona montando a caballo o a cebra.

Figura 30: Fallos del método

2.5 *Data Augmentation*

El aumento de datos es el proceso de generar artificialmente nuevos datos a partir de datos existentes, principalmente para entrenar nuevos modelos de *machine learning* (ML) [23]. Los modelos de ML requieren conjuntos de datos grandes y variados para el entrenamiento inicial, pero obtener conjuntos de datos del mundo real suficientemente diversos puede ser un desafío debido a los silos de datos, las regulaciones y otras limitaciones. El aumento de datos aumenta artificialmente el conjunto de datos al realizar pequeños cambios en los datos originales. Las soluciones de inteligencia artificial (IA) generativa se utilizan ahora para aumentar los datos de forma rápida y con alta calidad en diversos sectores.

¿Por qué es importante el aumento de datos?

Los modelos de aprendizaje profundo se basan en grandes volúmenes de datos diversos para desarrollar predicciones precisas en diversos contextos. El aumento de datos complementa la creación de variaciones de datos que pueden ayudar a un modelo a mejorar la precisión de sus predicciones. Los datos aumentados son fundamentales en el entrenamiento.

Estos son algunos de los beneficios del aumento de datos:

- **Rendimiento mejorado del modelo**

  Las técnicas de aumento de datos ayudan a enriquecer los conjuntos de datos al crear muchas variaciones de los datos existentes. Esto proporciona un conjunto de datos más grande para el entrenamiento y permite que un modelo encuentre características más diversas. Los datos aumentados ayudan al modelo a generalizar mejor a datos invisibles y a mejorar su rendimiento general en entornos del mundo real.

- **Reducción de la dependencia de los datos**

  La recopilación y preparación de grandes volúmenes de datos para el entrenamiento pueden resultar costosas y consumen mucho tiempo. Las técnicas de aumento de datos aumentan la eficacia de los conjuntos de datos más pequeños, lo que reduce considerablemente la dependencia de conjuntos de datos de gran tamaño en los entornos de entrenamiento. Puede usar conjuntos de datos más pequeños para complementar el conjunto con puntos de datos sintéticos.

- **Mitigación del sobreajuste en los datos de entrenamiento**

  El aumento de datos ayuda a evitar el sobreajuste cuando entrena modelos de ML. El sobreajuste es un comportamiento de ML no deseado en el que un modelo puede proporcionar predicciones precisas para los datos de entrenamiento, pero tiene problemas con los datos nuevos. Si un modelo se entrena solo con un conjunto de datos limitado, se puede sobreajustar y proporcionar predicciones relacionadas únicamente con ese tipo de datos específico. Por el contrario, el aumento de datos

proporciona un conjunto de datos mucho más amplio y completo para el entrenamiento de modelos. Hace que los conjuntos de entrenamiento parezcan exclusivos de las redes neuronales profundas, lo que les impide aprender a trabajar solo con características específicas.

Técnicas de aumento de datos

- **Visión artificial**

El aumento de datos es una técnica central en las tareas de visión artificial. Ayuda a crear diversas representaciones de datos y a abordar los desequilibrios de clase en un conjunto de datos de entrenamiento.

El primer uso del aumento en la visión artificial es mediante el aumento de posición. Esta estrategia recorta, voltea o rota una imagen de entrada para crear imágenes aumentadas. Al recortar, se cambia el tamaño de la imagen o se recorta una pequeña parte de la imagen original para crear una nueva. La transformación de rotación, volteo y cambio de tamaño alteran el original de forma aleatoria con una probabilidad determinada de proporcionar nuevas imágenes.

Otro uso del aumento en la visión artificial es en el aumento del color. Esta estrategia ajusta los factores elementales de una imagen de entrenamiento, como el brillo, el grado de contraste o la saturación. Estas transformaciones de imagen comunes cambian el tono, el equilibrio entre la oscuridad y la luz, y la separación entre las áreas más oscuras y claras de una imagen para crear imágenes aumentadas.

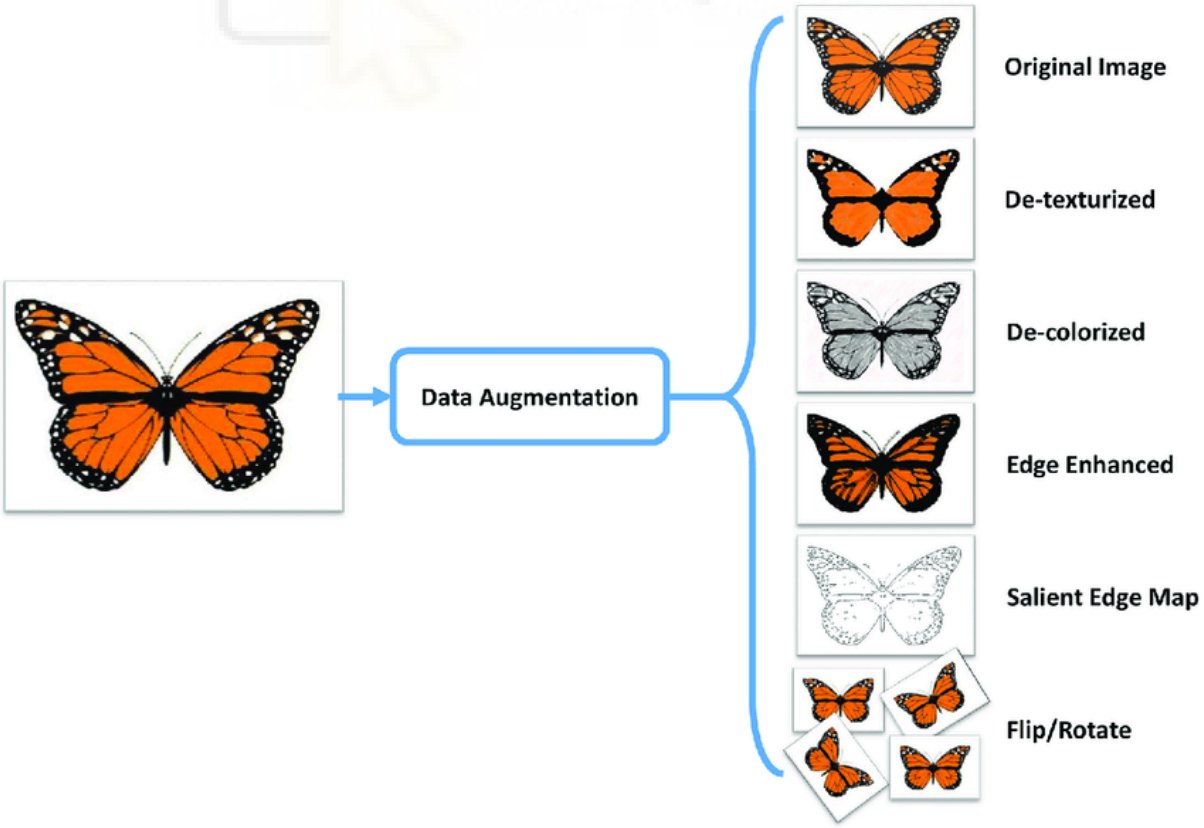


Figura 31: ejemplo de *Data Augmentation*

- **Redes GAN**

Con el tiempo, las *GAN* mejoran continuamente el contenido de salida del generador al centrarse en engañar al discriminador. Los datos que pueden engañar al discriminador cuentan como datos sintéticos de alta calidad, lo que proporciona un aumento de datos con muestras altamente confiables que imitan de cerca la distribución de datos original.

Redes generativas adversarias (*GANs*) puede ofrecer muchas ventajas sobre los métodos tradicionales de aumento de datos [24], como la generación de datos más diversos y realistas que conservan las características semánticas y estructurales, el aprendizaje de la distribución latente de los datos, el manejo de datos complejos y de alta dimensión, y el abordaje del problema del colapso de modo. Las *GAN* pueden generar datos que las técnicas tradicionales no pueden, como caras, textos o voz, y se pueden usar para aumentar los datos con los que es difícil trabajar. Además, se pueden usar diferentes arquitecturas o métodos de regularización para evitar el colapso del modo.

3. HERRAMIENTAS

A continuación vamos a ver las principales herramientas en las que se basa este trabajo, desde la base de datos hasta el software utilizado.

3.1 Base de datos (Cold database)

El nombre COLD es un acrónimo de la base de datos de localización *COsy* (*Cognitive Systems for Cognitive Assistants*). La base de datos representa un esfuerzo por proporcionar un entorno de pruebas flexible y a gran escala para evaluar sistemas de localización basados principalmente en la visión y destinados a funcionar en plataformas móviles en entornos realistas. La base de datos COLD consta de tres conjuntos de datos (COLD-Freiburg, COLD-Ljubljana, COLD-Saarbrücken) adquiridos en tres entornos de laboratorio de interior diferentes situados en tres ciudades europeas distintas: el Laboratorio de Sistemas Inteligentes Autónomos de la Universidad de Friburgo (Alemania); el Laboratorio de Sistemas Cognitivos Visuales de la Universidad de Liubliana (Eslovenia); y el Laboratorio de Tecnología del Lenguaje del Centro Alemán de Investigación en Inteligencia Artificial de Saarbrücken (Alemania). La base de datos contiene secuencias de imágenes captadas con una cámara normal y otra omnidireccional, así como escáneres láser y datos de odometría. Los datos se grabaron utilizando tres plataformas robóticas móviles diferentes y la misma configuración de cámara, en diversas condiciones meteorológicas y de iluminación (con tiempo nublado, soleado y de noche) a lo largo de varios días. En cada uno de los tres laboratorios, la adquisición se realizó en varias salas de distinta funcionalidad. En consecuencia, la base de datos COLD es un banco de pruebas ideal para evaluar la solidez de los algoritmos de localización y reconocimiento/categorización de lugares con respecto a los cambios tanto categóricos como dinámicos (introducidos por las variaciones de iluminación y la actividad humana) [25].

Las siguientes imágenes ilustran los tipos de variabilidad que recoge la base de datos COLD:

Figura 32: Influencia de las variaciones de iluminación en el aspecto del entorno.

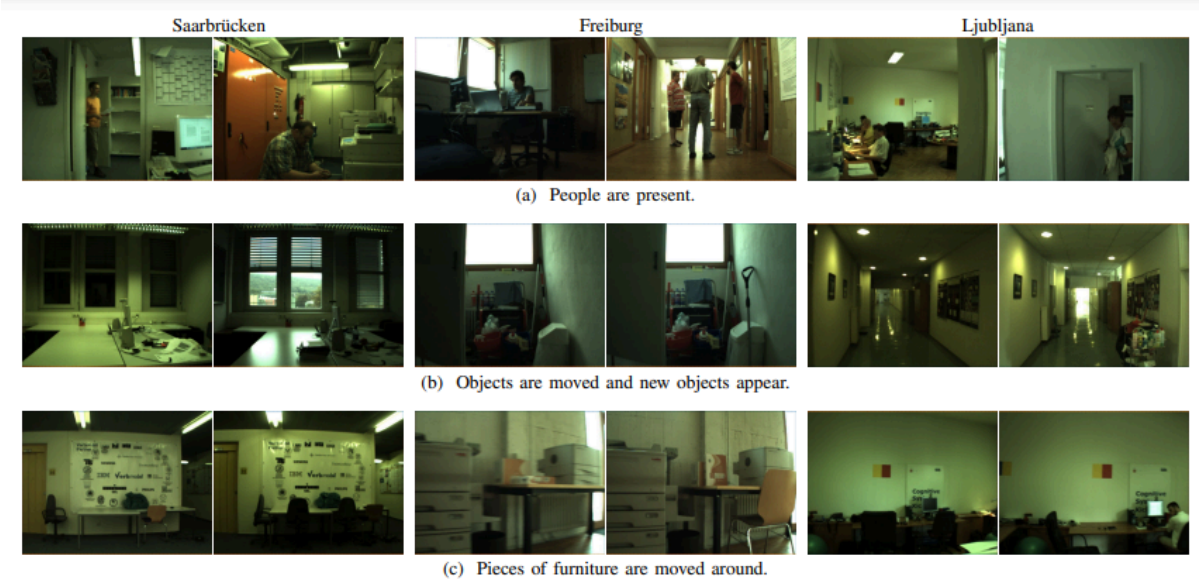
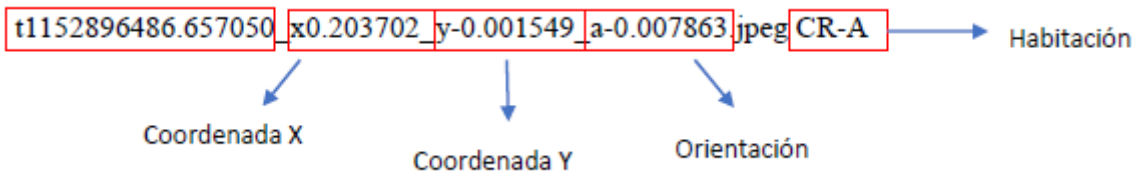


Figura 33: Variaciones introducidas por la actividad humana en el medio ambiente.

Figura 34: Variabilidad intra categoría para tres categorías de habitaciones.

Las imágenes contenidas en la base de datos se etiquetan de la siguiente manera:



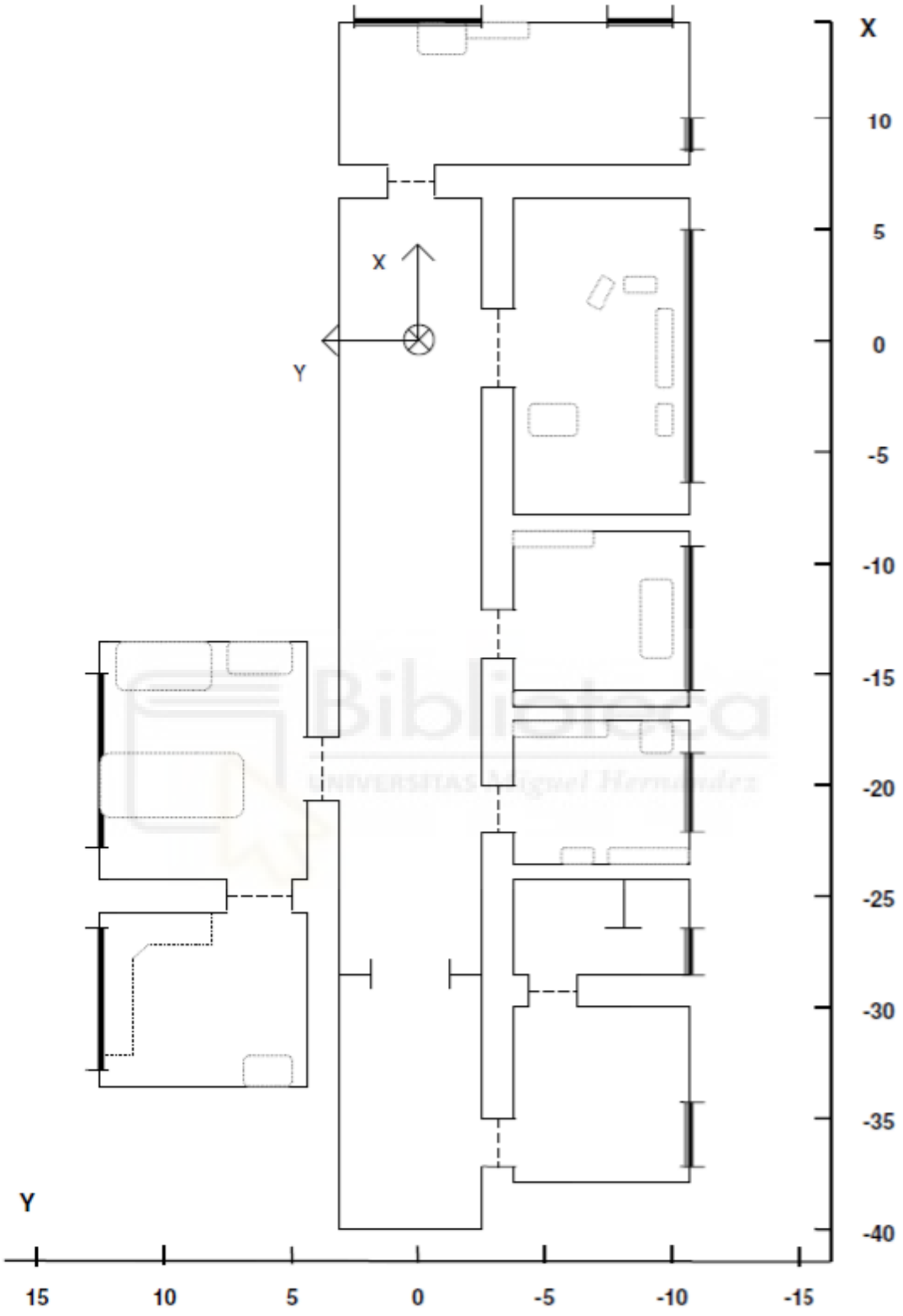


Figura 35: Sistema de coordenadas seguido en el mapa de Friburgo

3.2 Pycharm (Python)

La principal herramienta de este trabajo es la de Pycharm, el IDE más popular para Python hasta la fecha.

Pero antes de entrar en por qué usar Pycharm y no otro IDE vamos a describir un poco qué es Python y que es una IDE.

Python es un lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de software, la ciencia de datos y el *Machine Learning* (ML). Los desarrolladores utilizan Python porque es eficiente y fácil de aprender, además de que se puede ejecutar en muchas plataformas diferentes. El software Python se puede descargar gratis, se integra bien a todos los tipos de sistemas y aumenta la velocidad del desarrollo.
Entre sus beneficios nos encontramos con la posibilidad de desarrollar códigos con menos líneas en comparación a otros lenguajes, la existencia de una biblioteca que contiene códigos reutilizables para casi cualquier tarea o la opción de trasladarse de diferentes sistemas operativos como Windows o Linux [26].

Un *Integrated Development environment (IDE)* o Entorno de Desarrollo Integrado (EDI), en cambio, es un conjunto de herramientas necesarias para desarrollar software. Incluye un editor y un compilador.

El uso de un *IDE* simplifica enormemente la programación y el proceso de desarrollo. La herramienta interpreta lo que el desarrollador escribe y sugiere palabras clave relevantes para insertar, y asigna diferentes colores a los distintos elementos del código.

Un *Integrated Development Environment* incluye un editor de texto, un editor de proyectos en el que se pueden almacenar archivos, y muchos módulos y paquetes para añadir funcionalidades fácilmente.

PyCharm tiene muchas ventajas. Su editor de código inteligente ayuda a escribir código de alta calidad. Sus diferentes códigos de colores para las palabras clave, las clases y las funciones aumentan la legibilidad y la comprensión del código. Esto también simplifica la detección de errores. También está incluida la función de autocompletar.

Las funciones de navegación de código ayudan a los desarrolladores a editar y mejorar el código sin esfuerzo, y a navegar fácilmente hacia una función, clase o archivo. Localizar un elemento, símbolo o variable en el código fuente es muy sencillo, y el modo lens permite inspeccionar y depurar todo el código fuente.

Además, es compatible con bibliotecas científicas de Python como Matplotlib, NumPy y Anaconda. Por lo tanto, este *IDE* es especialmente útil para proyectos de Data Science y *Machine Learning* [27].

3.3 Pytorch

PyTorch [28] es un marco de deep learning de código abierto basado en software que se utiliza para crear redes neuronales, combinando la biblioteca de *machine learning* (ML) de Torch con una *API (Application Programming Interface)* de alto nivel basada en Python. Su flexibilidad y facilidad de uso, entre otros beneficios, lo han convertido en el marco de ML líder para las comunidades académicas y de investigación.

PyTorch admite una amplia variedad de arquitecturas de red neuronal, desde algoritmos de regresión lineal simple hasta redes neuronales convolucionales complejas y modelos de transformadores generativos utilizados para tareas como visión por ordenador y procesamiento del lenguaje natural (*NLP*). Basado en el lenguaje de programación Python y ofreciendo amplias bibliotecas de modelos preconfigurados (e incluso pre entrenados), PyTorch permite a los científicos de datos crear y ejecutar sofisticadas redes de deep learning, al tiempo que minimiza el tiempo y la mano de obra dedicados al código y la estructura matemática.

PyTorch también permite a los científicos de datos ejecutar y probar partes del código en tiempo real, en lugar de esperar a que se implemente todo el código, lo que, para grandes modelos de deep learning, puede llevar mucho tiempo. Esto hace que PyTorch sea una excelente plataforma para la creación rápida de prototipos y también agiliza enormemente el proceso de depuración.

¿Cómo funciona PyTorch?

La estructura matemática y de programación de PyTorch simplifica y agiliza los flujos de trabajo de aprendizaje automático, sin limitar la complejidad o el rendimiento de las redes neuronales profundas.

Tensores

En cualquier algoritmo de aprendizaje automático, incluso en los que se aplican a información aparentemente no numérica, como sonidos o imágenes, los datos deben representarse numéricamente. En PyTorch, esto se logra a través de tensores, que sirven como unidades fundamentales de datos utilizados para el cálculo en la plataforma.

En el contexto del aprendizaje automático, un tensor es una matriz multidimensional de números que funciona como un dispositivo de contabilidad matemática. Lingüísticamente hablando, "tensor" funciona como un término genérico que incluye algunas entidades matemáticas más conocidas:

- Un escalar es un tensor de dimensión cero, que contiene un solo número.
- Un vector es un tensor unidimensional que contiene varios escalares del mismo tipo. Una tupla es un tensor unidimensional que contiene diferentes tipos de datos.
- Una matriz es un tensor bidimensional que contiene varios vectores del mismo tipo.

- Los sensores con tres o más dimensiones, como los tensores tridimensionales utilizados para representar imágenes RGB en algoritmos de visión informática, se denominan colectivamente tensores N-dimensionales.

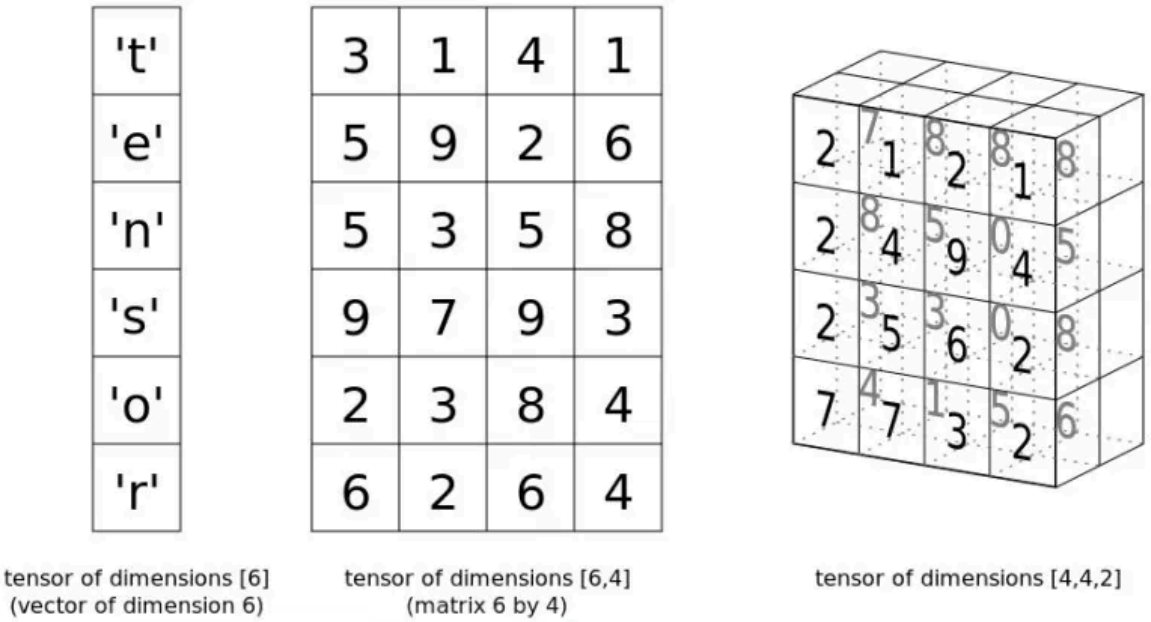


Figura 36: Ejemplos de tensores [29]

Los tensores de PyTorch funcionan de manera similar a los ndarrays que se usan en NumPy, pero a diferencia de los ndarrays, que solo se pueden ejecutar en unidades centrales de procesamiento (CPU), los tensores también se pueden ejecutar en unidades de procesamiento de gráficos (GPU). Las GPU permiten un cálculo mucho más rápido que las CPU, lo que supone una gran ventaja dado que los volúmenes masivos de datos y el procesamiento paralelo son típicos del deep learning.

Además de codificar las entradas y salidas de un modelo, los tensores PyTorch también codifican los parámetros del modelo: los pesos, sesgos y gradientes que se "aprenden" en el aprendizaje automático. Esta propiedad de los tensores permite la diferenciación automática, que es una de las características más importantes de PyTorch.

Módulos

PyTorch utiliza módulos como bloques de construcción de modelos de aprendizaje profundo, lo que permite la construcción rápida y sencilla de redes neuronales sin el tedioso trabajo de codificar manualmente cada algoritmo.

Los módulos pueden, y a menudo contienen, otros módulos anidados. Además de permitir la creación de redes neuronales multicapa más elaboradas, esto también permite guardar fácilmente estos complejos modelos de deep learning como un único módulo con nombre y transferirlos entre diferentes máquinas, CPU o GPU. Los modelos PyTorch incluso se pueden ejecutar en entornos que no sean Python, como C++, utilizando Torchscript, lo que ayuda a cerrar la brecha entre los prototipos de investigación y la implementación de producción.

En términos generales, hay tres clases principales de módulos que se utilizan para crear y optimizar modelos de deep learning en PyTorch:

- `nn módulos` se implementan como capas de una red neuronal. El paquete torch.nn contiene una gran biblioteca de módulos que realizan operaciones comunes como convoluciones, agrupación y regresión. Por ejemplo, `torch.nn.Linear(n,m)` llama a un algoritmo de regresión lineal con n entradas y m salidas (cuyas entradas y parámetros iniciales se establecen en las siguientes líneas de código).

- El módulo `autograd` proporciona una forma sencilla de calcular automáticamente los gradientes, utilizado para optimizar los parámetros del modelo a través del descenso del gradiente, para cualquier función operada dentro de una red neuronal. Añadir cualquier tensor con `requires_grad=True` indica a autograd que cada operación en ese tensor debe ser rastreada, lo que permite la diferenciación automática.

- Los módulos `optim` aplican algoritmos de optimización a esos gradientes. Torch.optim proporciona módulos para varios métodos de optimización, como el descenso estocástico del gradiente (*SGD*) o la propagación cuadrática media (*RMSprop*), para adaptarse a necesidades de optimización específicas.

Diferenciación automática

Un método muy utilizado para entrenar redes neuronales, especialmente en el aprendizaje supervisado, es la retropropagación. En primer lugar, en una pasada hacia adelante, un modelo recibe algunas entradas (x) y predice algunas salidas (y); trabajando hacia atrás desde esa salida, se utiliza una función de pérdida para medir el error de las predicciones del modelo en diferentes valores de x. Al diferenciar esa función de pérdida para encontrar su derivada, se puede utilizar el descenso del gradiente para ajustar los pesos en la red neuronal, una capa a la vez.

El módulo de *autograd* de PyTorch potencia su técnica de diferenciación automática mediante una fórmula de cálculo llamada regla de la cadena, que calcula derivadas complejas dividiéndolas en derivadas más simples y combinándolas posteriormente. Autograd calcula y registra automáticamente los gradientes de todas las operaciones ejecutadas en un gráfico computacional, lo que reduce considerablemente el trabajo preliminar de la retropropagación.

Al ejecutar un modelo que ya se ha entrenado, el autograd se convierte en un uso innecesario de los recursos computacionales. Agregar cualquier operación de tensor con `requires_grad=False` indicará a PyTorch que deje de rastrear los gradientes.

Conjuntos de datos y cargadores de datos

Trabajar con los grandes conjuntos de datos necesarios para entrenar modelos de deep learning puede ser complejo y exigente computacionalmente. PyTorch proporciona dos primitivas de datos, datasets y dataloaders, para facilitar la carga de datos y hacer que el código sea más fácil de leer.

- `torch.utils.data.Dataset` almacena muestras de datos y sus etiquetas correspondientes
- `torch.utils.data.Dataloader` encapsula un iterable (un objeto sobre el que se puede operar) alrededor del conjunto de datos para permitir un fácil acceso a las muestras

4. EXPERIMENTOS Y RESULTADOS

Como hemos descrito en el objetivo del TFG, vamos a tratar de generar imágenes que cambien de un dominio a otro con la idea de aumentar una base de datos. En nuestro caso, el cambio de dominio son cambios de iluminación, de modo que partiendo de una imagen soleada consigamos que esa misma imagen simula que ha sido tomada en un entorno de noche o nublado, y viceversa. Para verificar que los resultados son buenos, a parte de la comparación visual, vamos a sacar una métrica de error (*Cycle consistenc loss)* que muestre que la imagen original (x) de la que se parte la transformación y la reconstrucción de la generada a partir de esta original (F(G(x)) sean prácticamente la misma imagen, por lo que el error debería ser cercano a cero.

4.1 Preparación de los datos

Para nuestro trabajo, dentro de la *COLD Database* mencionada anteriormente, nos centraremos en el Laboratorio de Sistemas Inteligentes Autónomos de la Universidad de Friburgo (Alemania).

Para esta base de datos del laboratorio de Friburgo se utilizó la plataforma robótica *ActivMedia Pioneer-3* [Fig.37] para la adquisición de datos, proporcionando una solución robusta para tareas de navegación autónoma y recolección de datos de sensores. Este robot, equipado con un sistema de tracción diferencial, permitió la captura de información precisa mediante sensores avanzados. En este caso, se adquirieron tres tipos de datos principales: **imágenes regulares y omnidireccionales** [Fig.37] sincronizadas, exploraciones láser de alcance y datos de odometría. Las imágenes regulares fueron tomadas con una cámara convencional, mientras que las omnidireccionales, capturadas con una cámara especializada, ofrecían una vista completa de 360 grados del entorno. Ambas imágenes estaban sincronizadas para obtener una representación integral del entorno en cada momento. Por su parte, las exploraciones láser, obtenidas mediante un sensor LiDAR, proporcionaron mediciones precisas de la distancia a los objetos circundantes, lo que resultó esencial para el mapeo 2D y la navegación segura. La odometría, basada en el sistema de ruedas del robot, complementó la información, permitiendo un seguimiento exacto de la trayectoria recorrida.

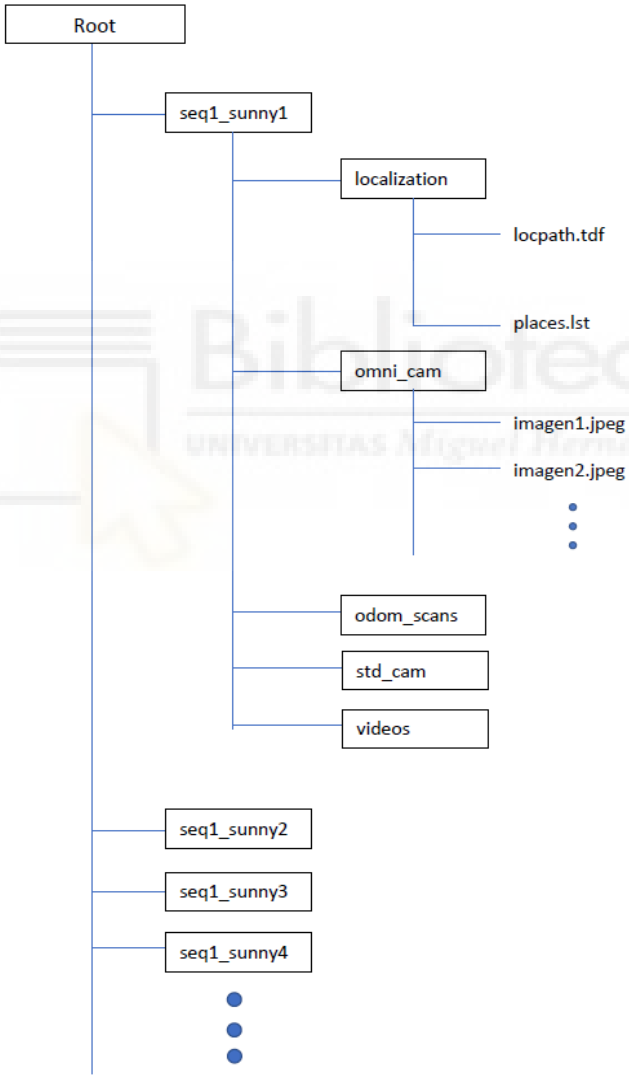
El entorno donde se realizaron las adquisiciones correspondía a un entorno de oficina dividido en 14 habitaciones, que fueron clasificadas en 8 categorías según su función (por ejemplo, oficinas, salas de reuniones, pasillos, etc.) [Fig.38]. Durante el proceso, se recogieron **26 secuencias** en total, cubriendo diversas rutas dentro de estas habitaciones. La adquisición de datos se llevó a cabo en dos partes distintas del mismo entorno de oficina, denominadas **Parte A** y **Parte B**.

En cada parte, se diseñaron dos recorridos diferenciados. El **recorrido estándar** cubría salas comunes en la mayoría de los entornos de oficina, tales como oficinas, pasillos y salas de conferencias, proporcionando una base sólida para el análisis en escenarios laborales típicos. El **recorrido extendido**, en cambio, incluyó habitaciones adicionales específicas del

entorno particular de la oficina, lo que enriqueció el conjunto de datos con información más diversa y menos común. Esta distinción permitió que los datos capturados en las dos partes del entorno (A y B) ofrecieran una representación completa de las áreas más típicas, así como de las zonas específicas de esta oficina.

Este enfoque estructurado de adquisición de datos, que combinaba información visual, láser y de odometría, junto con las diferentes rutas diseñadas para cubrir habitaciones estándar y específicas, generó un conjunto de datos altamente variado y útil para diversas áreas de investigación, como la localización y mapeo simultáneos (SLAM), la clasificación de escenas y la navegación autónoma.

Con el siguiente esquema podremos entender mejor la estructura de dicha base de datos:



Para el proceso de **entrenamiento y validación** de nuestro modelo, hemos decidido utilizar específicamente las imágenes omnidireccionales capturadas durante la secuencia 3 del camino extendido perteneciente a la parte A del entorno de oficina. Esta secuencia fue seleccionada por su representatividad y riqueza en detalles, ya que el recorrido ampliado de la parte A incluye no sólo las salas comunes que se encuentran típicamente en la mayoría

de los entornos de oficina, sino también espacios específicos de este entorno particular. De este modo, las imágenes adquiridas en esta secuencia ofrecen una variedad significativa de escenarios y características visuales que son ideales para entrenar el modelo. Estas imágenes permiten al modelo aprender a interpretar diferentes tipos de habitaciones y condiciones de iluminación, garantizando un proceso de entrenamiento más robusto.

Durante el entrenamiento, hemos seguido una estrategia de división en la que las imágenes de la secuencia 3 se separan en subconjuntos, con un porcentaje destinado al entrenamiento propiamente dicho y otro porcentaje reservado para la validación. Como mencionamos anteriormente, este proceso de selección está automatizado mediante un código en Python, que se encarga de distribuir las imágenes de manera adecuada, asegurando que el conjunto de entrenamiento y el de validación sean representativos y no tengan solapamiento.

Por otro lado, para **testear el entrenamiento**, es decir, para evaluar de forma objetiva el rendimiento del modelo una vez entrenado, hemos optado por utilizar las imágenes omnidireccionales correspondientes a la secuencia 2 del mismo camino extendido de la parte A. Esta secuencia, aunque sigue el mismo recorrido ampliado, es distinta de la secuencia 3 y no se ha utilizado durante la fase de entrenamiento o validación. De esta forma, nos aseguramos de que el modelo se someta a una prueba rigurosa, donde debe demostrar su capacidad para generalizar a nuevos datos que no ha visto antes.

La elección de la secuencia 2 para la fase de prueba tiene varias ventajas. Al estar capturada en el mismo camino ampliado pero en un momento o bajo condiciones distintas, ofrece una evaluación precisa de cómo el modelo puede desempeñarse en escenarios similares pero no idénticos a los que vio durante el entrenamiento. Además, al utilizar una secuencia diferente, evitamos el riesgo de sobreajuste (*overfitting*), donde el modelo podría memorizar las imágenes vistas en el entrenamiento en lugar de aprender a generalizar.

En resumen, la **secuencia 3 del camino extendido en la parte A** ha sido elegida para entrenar y validar el modelo debido a su diversidad de escenas y su capacidad de exponer al modelo a un amplio rango de situaciones. Mientras tanto, la **secuencia 2 del mismo camino** ha sido seleccionada para el testeo, lo que nos permite realizar una evaluación final del rendimiento del modelo en condiciones que, aunque similares, son completamente nuevas para el sistema. Esto garantiza que el modelo no solo se ajuste a los datos vistos en el entrenamiento, sino que también sea capaz de interpretar correctamente nuevas imágenes en escenarios reales o no observados previamente [30].

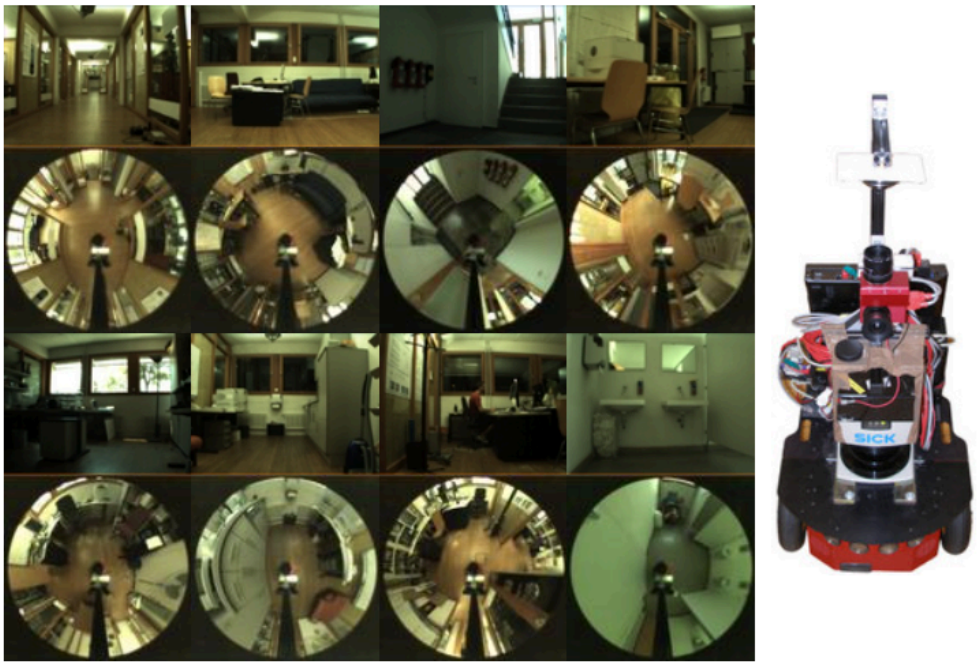


Figura 37: imágenes regulares y omnidireccionales del laboratorio de Friburgo junto al robot utilizado

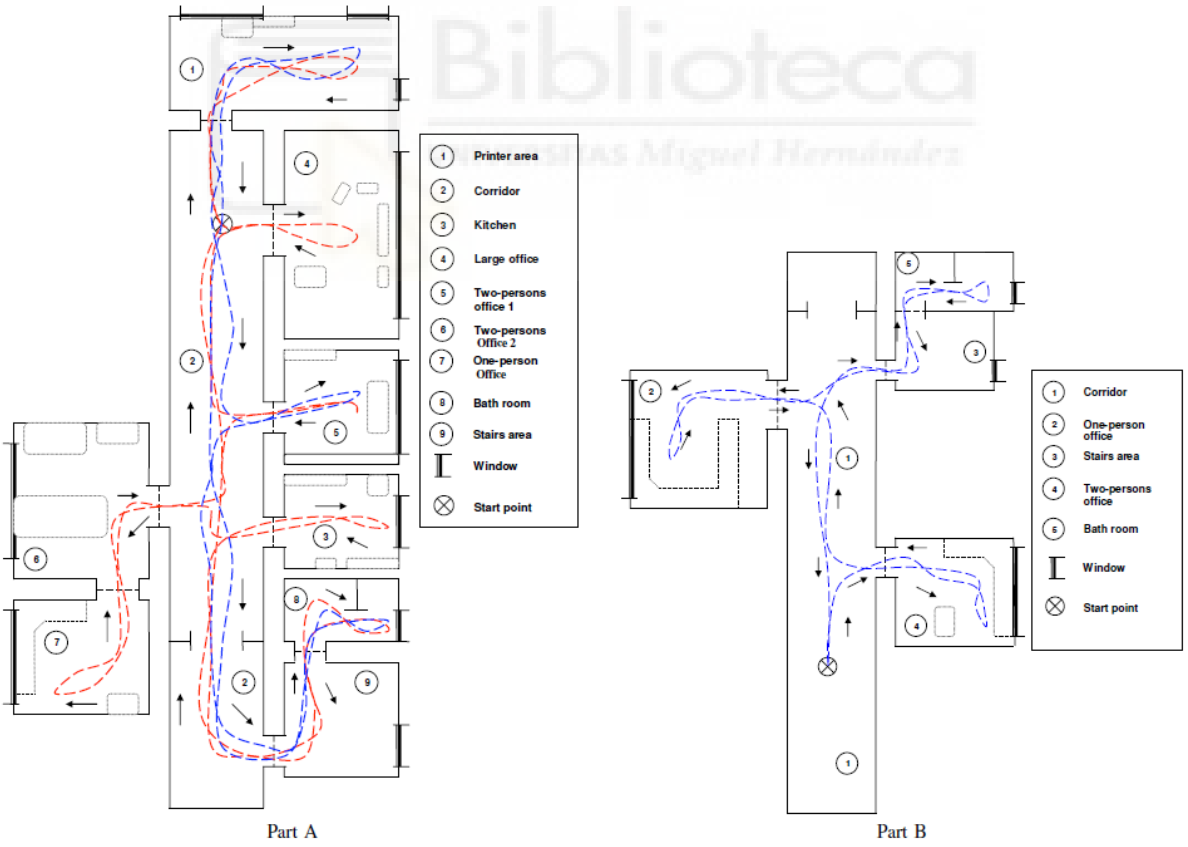


Figura 38: Mapa de las dos partes del laboratorio de Friburgo con los caminos seguidos por el robot durante la adquisición de datos. El camino estándar está representado por las rayas azules y el camino extendido está representado por por las rayas rojas. Las flechas indican la dirección que ha seguido el robot.

Conversión de omnidireccional a panorámica

Para lograr nuestro objetivo, es fundamental realizar una **conversión de las imágenes omnidireccionales a imágenes panorámicas** antes de llevar a cabo el proceso de entrenamiento de los modelos. Esta transformación es crucial, ya que las imágenes omnidireccionales capturan un campo de visión esférico completo, lo que las hace útiles para la captura de todo el entorno en una sola toma, pero no son directamente compatibles con muchos de los modelos de procesamiento de imágenes convencionales, que generalmente trabajan mejor con representaciones panorámicas. Por lo tanto, es necesario convertir estas imágenes omnidireccionales en un formato panorámico que pueda ser interpretado de manera efectiva por las redes neuronales y otros modelos de aprendizaje profundo.

El proceso de conversión se lleva a cabo **antes del entrenamiento** y la base de datos resultante, compuesta por las imágenes panorámicas, será la que se utilice como entradas en los diferentes modelos de entrenamiento. Esta etapa de **preprocesamiento** es esencial, ya que permite transformar las imágenes en un formato que maximiza su compatibilidad y aprovechamiento dentro de las arquitecturas de aprendizaje profundo, facilitando así la extracción de características y el rendimiento de los modelos.

La conversión se realiza mediante un código externo a los modelos de entrenamiento, pero también desarrollado en Python, lo que asegura una integración fluida dentro de todo el flujo de trabajo. Python se eligió por su flexibilidad y su amplio ecosistema de bibliotecas especializadas para la manipulación y procesamiento de imágenes, tales como **OpenCV** y **NumPy**, que permiten gestionar de manera eficiente este tipo de transformaciones geométricas. Este código externo se encarga de proyectar las imágenes esféricas de las cámaras omnidireccionales en un formato panorámico (por ejemplo, una proyección equirectangular), garantizando que los detalles visuales importantes se preserven durante el proceso de conversión.

Una vez que las imágenes han sido transformadas a este formato panorámico, estas imágenes convertidas forman la base de datos definitiva que será utilizada para entrenar los diferentes modelos. Esto asegura que el entrenamiento sea coherente y que los modelos puedan aprovechar la información visual de la manera más efectiva posible. De esta manera, el proceso de conversión de imágenes omnidireccionales a panorámicas no sólo es un paso técnico adicional, sino un componente clave para garantizar la eficacia del entrenamiento y los resultados en las tareas de procesamiento visual.

Figura 39: ejemplo de imagen omnidireccional

Figura 40: ejemplo de la misma imagen pero transformada en panorámica

División de imágenes en *train, validation* y *test*

Una vez realizada la conversión de las imágenes omnidireccionales a panorámicas, pasamos a la siguiente fase del proceso, que consiste en la **separación y organización de las imágenes** para su uso en las diferentes etapas del entrenamiento del modelo. Para este propósito, implementamos una estrategia automatizada que nos permite dividir eficientemente las imágenes en subconjuntos para **entrenamiento** y **validación**.

En concreto, aplicamos un esquema donde de cada 5 imágenes disponibles, seleccionamos una imagen para la fase de validación, mientras que las otras 4 imágenes restantes se utilizan para el entrenamiento del modelo. Este proceso de selección es crucial, ya que garantiza que el modelo se entrene de forma adecuada y, a la vez, tenga una base sólida para validación que permita monitorizar su rendimiento y ajustar los hiperparámetros. Esta separación se realiza de manera automática a través de un código en Python, lo que asegura que el proceso sea eficiente y repetible, evitando errores humanos y facilitando el manejo de grandes volúmenes de datos.

Para el conjunto de imágenes de **prueba** (test), no es necesario aplicar una separación adicional. Simplemente realizamos la conversión de las imágenes omnidireccionales a panorámicas, como en el caso de los conjuntos de entrenamiento y validación, y estas

imágenes se reservan exclusivamente para evaluar el rendimiento final del modelo. Este conjunto de prueba es crucial, ya que permite evaluar de manera objetiva el desempeño del modelo sobre datos nunca vistos, proporcionando una estimación realista de su capacidad para generalizar a nuevos ejemplos.

En cuanto a la organización final de las imágenes, al tratarse de imágenes capturadas bajo diferentes condiciones de iluminación (como día soleado, nublado o de noche), creamos tres carpetas diferentes para cada conjunto de imágenes, de modo que se separen según el tipo de luz presente en el entorno. Así, para cada condición de iluminación —*sunny* (soleado), *cloudy* (nublado) y *night* (noche)—, se generan tres carpetas independientes que contienen:

1. Una carpeta con las imágenes de entrenamiento.
2. Una carpeta con las imágenes de validación.
3. Una carpeta con las imágenes de prueba (test).

De esta forma, el proceso de entrenamiento del modelo se adapta a las diferentes condiciones de luz, permitiendo que el modelo aprenda de forma robusta a reconocer patrones y objetos en distintas situaciones lumínicas. Este enfoque asegura que el modelo no solo se entrene y valide en un solo tipo de entorno, sino que sea capaz de generalizar mejor cuando se enfrente a condiciones de iluminación diversas.

Finalmente, el número total de imágenes utilizadas en cada categoría y su distribución se refleja en una tabla resumen, donde se especifica cuántas imágenes se utilizarán para el entrenamiento, la validación y la prueba en cada una de las condiciones de iluminación (soleado, nublado y noche). Esto proporciona una visión clara del volumen de datos manejado en cada etapa, lo que es esencial para asegurar una correcta planificación y monitoreo del proceso de entrenamiento.

	Train	***Validation***	***Test***
Sunny	1788	443	2114
Cloudy	2227	551	2595
Night	2320	576	2707

Tabla 1: número total de imágenes por iluminación y método

4.2 Implementación del entrenamiento de las redes Cyclegan con Python

Ahora nos vamos a centrar en ver cómo ha sido la implementación del *paper* que plantea las redes Cyclegan [16]. Será mediante Python y se basa principalmente en la librería `pytorch`.

Todos los códigos empleados para creación de la red, entrenamiento y test que vamos a comentar en los siguientes apartados y que se verán de manera superficial, se pueden encontrar en el siguiente enlace de *Github*: https://github.com/CarlosGarlo/TFG_Cyclegan.git

La implementación del entrenamiento y validación se dividirá en 4 partes. En primer lugar, definiremos la arquitectura del discriminador y, en segundo lugar, la arquitectura del generador. A continuación, crearemos el cargador del *dataset* y un preprocesamiento del conjunto de datos. Por último, crearemos una función de entrenamiento para entrenar ambas partes [20]:

Modelo discriminador

En este apartado desarrollaremos el modelo genérico encargado de discriminar. Para ello lo implementaremos como se indica en la teoría del *paper* que hemos comentado previamente [16].

Como en todos nuestros códigos, lo primero que tenemos que hacer es importar las distintas librerías y funciones que nos faciliten la creación del código, como por ejemplo: torch, torch.nn para la creación de redes neuronales.

En el discriminador, utilizaremos el bloque *Convolution-InstanceNorm-Leaky ReLU* como se menciona en el artículo [Fig.41].

Sea Ck una capa *Convolution-InstanceNorm-Leaky ReLU* de 4 × 4 con k filtros y stride 2. La arquitectura del discriminador es C64-C128-C256-C512. Después de la última capa, aplicamos una convolución para producir una salida unidimensional. Utilizamos ReLUs permeables (*leaky Relu*) con una pendiente de 0,2.

Tenga en cuenta que alimentamos la salida del modelo en una función sigmoidea para hacer una etiqueta 1/0. Esta clase discriminadora se utilizará para crear 2 objetos, ambos representando discriminador para ambas condiciones de iluminación falsas.
Todo esto se ve de manera más clara en la **figura 23** del **capítulo 2.4.1**.

```python
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size: 4,
                stride,
                padding: 1,
                bias=True,
                padding_mode="reflect",
            ),
            nn.InstanceNorm2d(out_channels),
            nn.LeakyReLU( negative_slope: 0.2, inplace=True),
        )

    def forward(self, x):
        return self.conv(x)
```

Figura 41: Código del bloque que crea una capa convolucional del discriminador

Modelo generador

En primer lugar, crearemos el bloque convolucional que será usado en el generador. En esta parte, podemos inicializar 2 tipos de bloques convolucionales. El primero es un bloque de muestreo descendente y el segundo es un bloque de muestreo ascendente. El *upsampling* utiliza *ConvTranspose2d* seguido de *InstanceNorm2d* y *ReLU*, como se describe en la arquitectura mencionada en el artículo y como se muestra en la **figura 20** del **capítulo 2.4.1**.

Después, también necesitamos un bloque residual como parte del generador. En un bloque residual, el uso de dos instancias de *ConvBlock*, una con una función de activación y la otra no, es una elección de diseño que promueve el aprendizaje de información residual.

El propósito de un bloque residual es aprender el mapeo residual entre la entrada y la salida del bloque. El primer *ConvBlock* de la secuencia que incluye una función de activación, ayuda a capturar y extraer características importantes de la entrada. La función de activación introduce no linealidad, lo que permite a la red modelar relaciones complejas entre la entrada y la salida.

Estos bloques descritos podemos verlos implementados en python en la **figura 42**.

El segundo *ConvBlock* no incluye una función de activación. Se centra principalmente en ajustar las dimensiones (por ejemplo, el número de canales) de las características extraídas por el primer *ConvBlock*. La ausencia de una función de activación en el segundo *ConvBlock* permite que el bloque aprenda la información residual. Al añadir directamente la salida del segundo *ConvBlock* a la entrada original, el bloque aprende a capturar las características residuales o los cambios necesarios para alcanzar la salida deseada.

Esta conexión de salto, conseguida mediante la operación de suma, ayuda a gradientes durante el entrenamiento y alivia el problema del gradiente. También facilita el flujo de información desde las capas capas anteriores a las posteriores, lo que permite a la red aprender con mayor eficacia.

Por último, usamos ambos bloques para crear el generador. Esta clase generadora será usada para crear 2 generadores, uno por cada cambio de iluminación correspondiente. Como se ha mencionado anteriormente, el generador se crea utilizando tres partes: el bloque de *convs* de muestreo descendente, los bloques residuales y los bloques de *conv* de muestreo ascendente. En esta implementación, utilizamos 6 bloques residuales y empleamos la arquitectura mencionada en el artículo.

```python
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down=True, use_act=True, **kwargs):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, padding_mode="reflect", **kwargs)
            if down
            else nn.ConvTranspose2d(in_channels, out_channels, **kwargs),
            nn.InstanceNorm2d(out_channels),
            nn.ReLU(inplace=True) if use_act else nn.Identity(),
        )

    def forward(self, x):
        return self.conv(x)


1 usage
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.block = nn.Sequential(
            ConvBlock(channels, channels, kernel_size=3, padding=1),
            ConvBlock(channels, channels, use_act=False, kernel_size=3, padding=1),
        )

    def forward(self, x):
        return x + self.block(x)
```

Figura 42: Creación de bloques convolucional y residual del generador

Dataset

En esta parte de la red nos encargamos de cargar las distintas imágenes de entrenamiento y validación para que puedan ser procesadas y utilizadas en la función *train* y *val* del código *train* principal.

Realmente tenemos dos datasets, uno por cada cambio de iluminación.

Lo primero que hacemos es indicar las rutas de las imágenes de entrada. Una vez partimos de aquí, nos centraremos en hacer un directorio con todas las imágenes que hemos cargado del *loader* de entrada procedente de la función *`main()`* con la función `os.listdir`. A su vez vemos la cantidad de imágenes que contiene cada directorio el cual será distinto en un dataset y en otro, lo cual ya hemos descrito con anterioridad que no es algo que importe porque nuestra red trabaja con entradas sin emparejar. De modo que nos basta con comprobar cual de los dos tiene un dataset mayor y será el que se devuelve en una función posterior (*max length*).

Ahora ya nos podemos centrar en la función *get item* la cual cogerá el índice de la imagen del directorio para saber cual es la imagen que se va a usar en el entrenamiento (una por cada cambio de iluminación) y evitar que nos dé error gracias a que sacamos el resto de la división entre el índice y el número de imágenes totales de ese directorio. Lo siguiente será crear una ruta para esas imágenes y finalizamos transformandolas en np.arrays (matrices de *numpy*) para poder usarlas en nuestras redes generadoras y discriminadoras, ya que será lo que se devuelva a la función *train.*

```
class SunnyNightDataset(Dataset):
    def __init__(self, root_night, root_sunny, transform=None):
        self.root_night = root_night
        self.root_sunny = root_sunny
        self.transform = transform

        self.night_images = os.listdir(root_night)
        self.sunny_images = os.listdir(root_sunny)
        self.length_dataset = max(len(self.night_images), len(self.sunny_images))  # 1000, 1500
        self.night_len = len(self.night_images)
        self.sunny_len = len(self.sunny_images)

    def __len__(self):
        return self.length_dataset

    def __getitem__(self, index):
        night_img_name = self.night_images[index % self.night_len] #renombrar y devolverla a train
        sunny_img_name = self.sunny_images[index % self.sunny_len]

        night_path = os.path.join(self.root_night, night_img_name)
        sunny_path = os.path.join(self.root_sunny, sunny_img_name)

        night_img = np.array(Image.open(night_path).convert("RGB"))
        sunny_img = np.array(Image.open(sunny_path).convert("RGB"))

        if self.transform:
            augmentations = self.transform(image=night_img, image0=sunny_img)
            night_img = augmentations["image"]
            sunny_img = augmentations["image0"]

        return night_img, sunny_img, night_img_name, sunny_img_name
```

Figura 43: Código correspondiente al dataset del modelo *Sunny-Night*

Train

Ahora nos centraremos en la implementación del entrenamiento, donde usaremos ambas clases del generador y discriminador para crear tanto un generador como un discriminador de cada cambio de iluminación.

En la primera sección, crearemos discriminadores para cada clase. Aquí, la pérdida se determina utilizando el error cuadrático medio (ECM) entre la salida del discriminador y la etiqueta real (1 ó 0) dependiendo de si se considera real o falsa. Observe también que la pérdida total en el discriminador se calcula tomando la media de las pérdidas de los dos discriminadores.

```
# Train Discriminators S and N
with torch.cuda.amp.autocast():
    fake_sunny = gen_S(night)
    D_S_real = disc_S(sunny)
    D_S_fake = disc_S(fake_sunny.detach())
    S_reals += D_S_real.mean().item()
    S_fakes += D_S_fake.mean().item()
    D_S_real_loss = mse(D_S_real, torch.ones_like(D_S_real))
    D_S_fake_loss = mse(D_S_fake, torch.zeros_like(D_S_fake))
    D_S_loss = D_S_real_loss + D_S_fake_loss

    fake_night = gen_N(sunny)
    D_N_real = disc_N(night)
    D_N_fake = disc_N(fake_night.detach())
    D_N_real_loss = mse(D_N_real, torch.ones_like(D_N_real))
    D_N_fake_loss = mse(D_N_fake, torch.zeros_like(D_N_fake))
    D_N_loss = D_N_real_loss + D_N_fake_loss

    # put it togethor
    D_loss = (D_S_loss + D_N_loss) / 2
    D_loss_array = D_loss.cpu().detach().numpy()
```

Figura 44: Código de entrenamiento para discriminadores *Sunny* y *Night*.

Lo siguiente será definir los generadores, donde estableceremos los dos tipos de pérdidas mencionados anteriormente: *adversarial loss* y *cycle consistency loss.* Estas pérdidas se suman combinándolas y asignando pesos a cada una de ellas:

- El *adversarial loss*, al igual que la pérdida del discriminador, se calcula con el ECM entre la salida del discriminador y la etiqueta de que es una imagen real para engañarla.

- Para el *cycle consistency loss* aplicaremos la norma L1 entre la reconstrucción de la imagen generada y la imagen original de la que partimos para generar dicha imagen.

  Es decir, hacemos uso de L1 entre X y $\widehat{X}$, de modo que idealmente este valor tiene que ser lo más cercano a cero ya que deberían ser la misma imagen.

```
# adversarial loss for both generators
D_S_fake = disc_S(fake_sunny)
D_N_fake = disc_N(fake_night)
loss_G_S = mse(D_S_fake, torch.ones_like(D_S_fake))
loss_G_N = mse(D_N_fake, torch.ones_like(D_N_fake))

# cycle loss
cycle_night = gen_N(fake_sunny)
cycle_sunny = gen_S(fake_night)
cycle_night_loss = l1(night, cycle_night)
cycle_night_loss_array = cycle_night_loss.cpu().detach().numpy()
cycle_sunny_loss = l1(sunny, cycle_sunny)
cycle_sunny_loss_array = cycle_sunny_loss.cpu().detach().numpy()
```

Figura 45: Código correspondiente de sacar las pérdidas adversarias y cíclicas de los generadores

Al final tanto del discriminador como del generador, definiremos la parte de propagación hacia atrás y actualizaremos los pesos:

```
opt_disc.zero_grad()
d_scaler.scale(D_loss).backward()
d_scaler.step(opt_disc)
d_scaler.update()
```

Figura 46: *Backpropagation* y actualización de pesos de los discriminadores.

Para ver los resultados por épocas, hemos decidido que cada 10 épocas y cada 200 imágenes de entrenamiento, guardemos la imagen generada por el entrenamiento para visualizar la calidad de la transformación. Posteriormente, veremos los resultados obtenidos.

Por último, cabe decir que llamaremos a esta función *train* y alimentaremos el dataset dentro de la red a través de la función principal *main* que será la que inicie el proceso de entrenamiento.

También me gustaría mencionar los siguientes detalles que son fundamentales para el correcto funcionamiento del entrenamiento:

- Los optimizadores Adam (`opt_disc` y `opt_gen`) se utilizan tanto para las redes generadoras como para las discriminadoras. A los optimizadores se les pasan los parámetros de las redes correspondientes, la tasa de aprendizaje (*LEARNING_RATE*) y los valores betas.

- Los objetos `GradScaler` (*`g_scaler`* y *`d_scaler`*) se crean utilizando `torch.cuda.amp.GradScaler()` para permitir el entrenamiento de precisión

mixta con escalado automático de gradientes. Estos escaladores se utilizan durante el entrenamiento para manejar los cálculos y ajustes de gradiente.

- El bucle de entrenamiento comienza, iterando sobre el número especificado de épocas (*NUM_EPOCHS*). Dentro de cada época, se llama a la función `train_fn`() para realizar el proceso de entrenamiento. Las redes discriminadoras y generadoras, junto con otros argumentos necesarios, se pasan a esta función.

Dentro de la función principal, para nuestro trabajo, hemos decidido añadir una función de validación, la cual funciona exactamente como la de entrenamiento, salvo que aquí no se actualizan los pesos del entrenamiento. Por ello, cada final de época, llamamos a la función de validación para ver que tal se está comportando la función de entrenamiento. Aquí también guardaremos imágenes cada 10 épocas y cada 200 iteraciones, para visualizar cuánto de bien funciona.

```
disc_S.eval()
disc_N.eval()
gen_N.eval()
gen_S.eval()
```

```
disc_S.train(True)
disc_N.train(True)
gen_N.train(True)
gen_S.train(True)
```

Figura 47: marcamos al inicio que en la función de validación solo vamos a evaluar y al final que volvemos a entrenar para la siguiente iteración del entrenamiento

Además, para nosotros es importante visualizar los *cycle consistency loss* y el error del discriminador a través de las épocas. Es por ello que hemos creado una lista que indica la media de estos valores por épocas y posteriormente la visualizamos en una gráfica que nos genera el paquete `matplotlib`. Además, esta función de pérdida marcará el fin del entrenamiento como vemos en la siguiente gráfica:

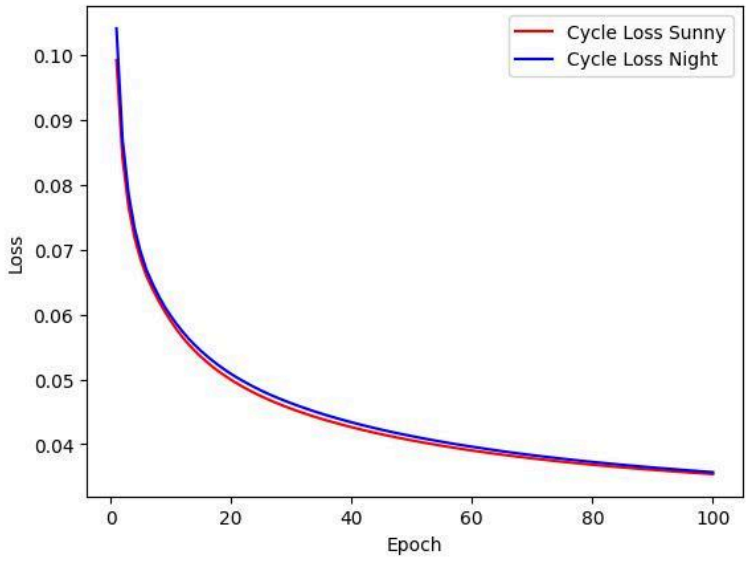


Figura 48: Cycle Consistency Loss del entrenamiento del modelo Sunny-Night

Para finalizar nuestro entrenamiento, hemos marcado el *cycle consistency loss* como parámetro que indica el correcto funcionamiento del entrenamiento y esto lo sabemos

porque la curva de entrenamiento y de la validación es muy similar y marcan la misma tendencia (convergen en el mismo punto). Por ello, cada época comprobamos si el valor indicado es aún menor al anterior, para así cargar en los modelos los *checkpoints* correspondientes al valor del *cycle consistency loss* (son los generadores y discriminadores creados junto a los pesos marcados por los gradientes que se han ido actualizando según mejoraba el entreno) y de esta manera asegurarnos que el modelo generado por el entrenamiento es el que mejor funciona. Tendremos un discriminador y generador por tipo de iluminación y por modelo entrenado (12 en total) y se guardarán en archivos .tar:

Fig 49: *checkpoints* de soleado y noche del modelo *Sunny-Night*

Config

Aquí nos centraremos en primer lugar en indicar que el computador use una GPU si la tiene, y si no tendrá que tirar de la CPU. En esta sección también nos encargamos de indicar las distintas rutas de donde se encuentran las diferentes carpetas de entrenamiento, validación… Así como también las rutas donde se guardaran los modelos generados con sus respectivos *checkpoints* (en .tar). También servirá para determinar los distintos hiperparámetros de la red (*batch size, lambda cycle, learning rate,* número de épocas… los cuales utilizaremos los mencionados en el *paper*, excepto el número de épocas que hemos decidido que sean 100). Además, también lo usamos para decidir si queremos cargar algún modelo de entrenamiento ya creado o si queremos guardar los modelos que generen los entrenamientos. Cabe destacar que las funciones encargadas de guardar y cargar los checkpoints, las tenemos en otro código externo llamado `utils.py`.

Por último, también preparamos un transformador para preprocesar la imagen, donde se le indicará el tamaño de las imágenes (en nuestro caso 512x128 por ser panorámicas) y el cual servirá para el *data augmentation* gracias a la librería `albumentations`.

```
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

TRAIN_DIR = "data/train"
VAL_DIR = "data/val"
TEST_DIR = "data/test"

MODEL_DIR_SN = "model_saved/SunnyNight"
MODEL_DIR_SC = "model_saved/SunnyCloudy"
MODEL_DIR_NC = "model_saved/NightCloudy"

BATCH_SIZE = 1
LEARNING_RATE = 1e-5
LAMBDA_IDENTITY = 0.0
LAMBDA_CYCLE = 10
NUM_WORKERS = 4
NUM_EPOCHS = 200

LOAD_MODEL = True
SAVE_MODEL = False

CHECKPOINT_GEN_S_SN = config.MODEL_DIR_SN + "/genS_sn.pth.tar"
CHECKPOINT_GEN_N_SN = config.MODEL_DIR_SN + "/genN_sn.pth.tar"
CHECKPOINT_DISCRIM_S_SN = config.MODEL_DIR_SN + "/discrimS_sn.pth.tar"
CHECKPOINT_DISCRIM_N_SN = config.MODEL_DIR_SN + "/discrimN_sn.pth.tar"
```

Figura 50: Parte del código config

4.3 Descripción de pérdidas y parámetros de entrenamiento de la red

En este apartado de la memoria vamos a expandirnos más sobre algunos términos que son parte fundamental del funcionamiento correcto del entrenamiento como son las funciones de pérdida y los hiperparámetros.

4.3.1 *Mean square error* (Error cuadrático medio)

Es una de las funciones de pérdida de regresión más comunes [31]. En el error cuadrático medio, también conocido como pérdida L2, calculamos el error elevando al cuadrado la diferencia entre el valor predicho y el valor real y promediando el conjunto de datos.

Representa la distancia al cuadrado entre los valores reales y predichos. Realizamos al cuadrado para evitar la cancelación de términos negativos y es útil elevar al cuadrado el error porque da un mayor peso a los valores atípicos, lo que da como resultado un gradiente suave para errores pequeños. Los algoritmos de optimización aprovechan esta gran penalización de error porque es útil para encontrar valores de parámetros óptimos.

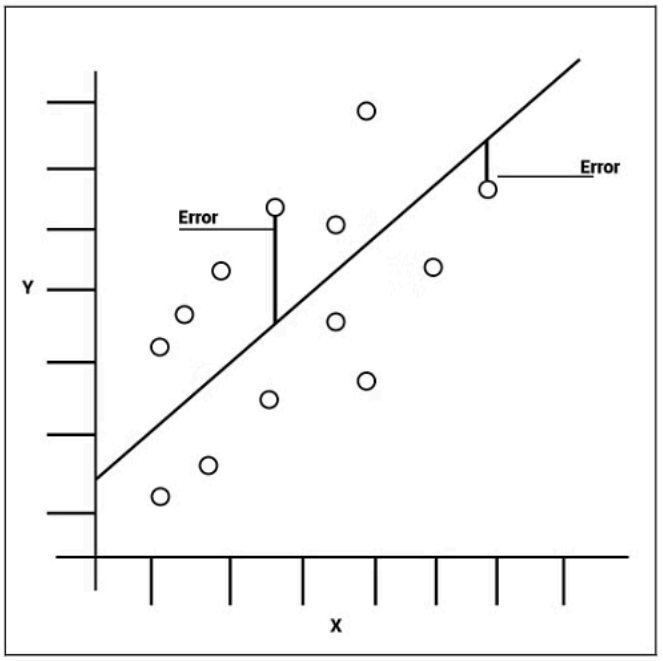


Figura 51: el error cuadrático medio mide la proximidad de una línea de regresión a un conjunto de puntos dados

$ECM = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$, siendo $\widehat{Y}_i$ un vector de n predicciones e Y el vector de los verdaderos valores.

Consideraciones:

- MSE nunca será negativo ya que los errores se elevan al cuadrado.
- El valor del error varía de cero a infinito.
- MSE aumenta exponencialmente con un aumento en el error.
- Un buen modelo tendrá un valor MSE más cercano a cero.

Ventajas del *MSE*:

- Los valores de *MSE* se expresan en ecuaciones cuadráticas. Por lo tanto, cuando lo graficamos, obtenemos un descenso de gradiente con solo un mínimo global.
- Para errores pequeños, converge a los mínimos de manera eficiente. No hay mínimos locales.
- *MSE* penaliza el modelo por tener grandes errores al elevarlos al cuadrado.
- Es particularmente útil para eliminar valores atípicos con grandes errores del modelo al ponerles más peso.

Cuándo utilizar el MSE

Saber cuándo utilizar el *MSE* es crucial en el desarrollo de modelos de machine learning [31]. El *MSE* es una función de pérdida estándar utilizada en la mayoría de las tareas de regresión, ya que dirige el modelo a optimizar para minimizar las diferencias al cuadrado entre los valores predichos y los valores objetivo.

El *MSE* se recomienda para escenarios de ML en los que favorece el proceso de aprendizaje penalizar significativamente la presencia de valores atípicos. Sin embargo, estas características del *MSE* no siempre son adecuadas para escenarios y casos de uso en los que los valores atípicos se deben al ruido de los datos, en contraposición a las señales positivas.

4.3.2 Pérdida L1 / Error medio absoluto

El Error medio absoluto (*MAE*), también conocido como Pérdida L1 [31], es una función de pérdida utilizada en tareas de regresión que calcula las diferencias absolutas medias entre los valores predichos de un modelo de machine learning y los valores reales del objetivo. A diferencia del Error cuadrático medio (ECM), el *MAE* no eleva al cuadrado las diferencias, tratando todos los errores con el mismo peso, independientemente de su magnitud.

$EMA = \frac{1}{n}\sum_{i=1}^{n} |Y_i - \widehat{Y}_i|^2$, siendo $\widehat{Y}_i$ un vector de n predicciones e Y el vector de los verdaderos valores.

Para comprenderlo mejor tomemos el siguiente ejemplo: consideremos que estamos pronosticando los precios de vehículos usados. El precio real de un vehículo usado es de 10 mil €. Después de aplicar un modelo de Machine Learning predijimos que el precio es de 12 mil €, ahora podemos observar que la diferencia es de 2 mil € entre ambos precios de venta. Esta diferencia se llama error absoluto.

Ventajas de MAE

- Es una métrica de evaluación fácil de calcular.
- Todos los errores se ponderan en la misma escala ya que se toman valores absolutos.
- Es útil si los datos de entrenamiento tienen valores atípicos, ya que MAE no penaliza los errores elevados causados por los valores atípicos.
- Proporciona una medida uniforme de qué tan bien está funcionando el modelo.

Cuándo utilizar el *MAE*

El *MAE* mide la diferencia absoluta media entre los valores previstos y los reales [31]. A diferencia del *MSE*, el *MAE* no eleva al cuadrado las diferencias, lo que lo hace menos sensible a los valores atípicos. Comparado con el Error cuadrático medio (ECM), el Error medio absoluto (*EAM*) es intrínsecamente menos sensible a los valores atípicos, porque asigna el mismo peso a todos los errores, independientemente de su magnitud.

Esto significa que, aunque un valor atípico puede sesgar significativamente el MSE al contribuir con un error desproporcionadamente grande cuando se eleva al cuadrado, su impacto en el *MAE* es mucho más contenido. La influencia de un valor atípico en la métrica de error global es mínima cuando se utiliza *MAE* como función de pérdida. En cambio, el *MSE* amplifica el efecto de los valores atípicos debido a la elevación al cuadrado de los términos de error, afectando más sustancialmente a la estimación del error del modelo.

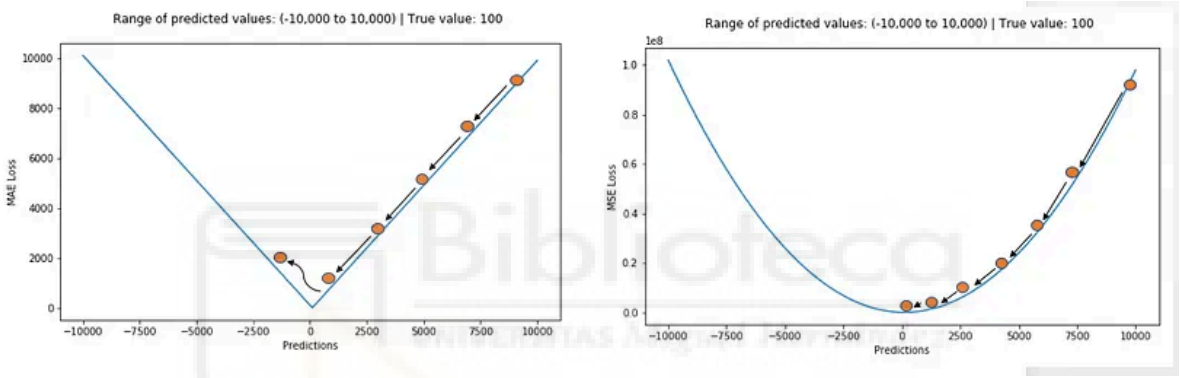


Figura 52: pérdida *MAE* vs *MSE* [32]

4.3.3. Hiperparámetros de entrenamiento de la red

Los hiperparámetros en una red neuronal son las variables que no se aprenden directamente del proceso de entrenamiento del modelo [33]. Se establecen antes del proceso del entrenamiento de la red y afectan en la forma en que se entrena el modelo.

- *Batch size*: Establece cuántas muestras se van a utilizar para calcular el gradiente y actualizar los pesos de la red en cada paso de entrenamiento. En nuestro caso será de 1.

- *Learning rate*: Establece la velocidad de ajuste de los pesos de la red durante el entrenamiento. Para nosotros, el valor será de 0.00001.

- *Epochs*: Establece cuántas veces se va a iterar sobre el conjunto de datos de entrenamiento. Nuestro valor ha ido alternando entre las distintas pruebas realizadas pero finalmente hemos optado que 100 épocas son suficientes.

- *Lambda cycle*: controla la importancia relativa entre los dos objetivos a la hora de sacar el error total del generador, el cual será de 10.

4.4 Resultados entrenamiento y validación

Como hemos mencionado en apartados anteriores, el objetivo principal es desarrollar tres modelos distintos de entrenamiento de redes CycleGAN, cuyo propósito será transformar imágenes entre diferentes condiciones de iluminación. En otras palabras, buscamos que a partir de imágenes con condiciones de iluminación soleadas, se puedan generar tanto imágenes nocturnas como nubladas, y aplicar este mismo enfoque a las otras dos combinaciones de iluminación. De esta manera, tendremos modelos que pueden transformar de manera bidireccional entre estas tres condiciones: soleado, nocturno y nublado. A continuación, detallamos cada uno de los tres modelos:

- **Modelo Sunny-Night:** Este primer modelo tiene la tarea de transformar imágenes soleadas en nocturnas y, de forma inversa, imágenes nocturnas en soleadas. Aquí, el generador de la red aprenderá a capturar las características de una escena soleada, como el brillo intenso, las sombras marcadas y los colores cálidos, y las transformará en un ambiente nocturno, donde predominen los tonos oscuros, la iluminación artificial y las sombras suaves. Por otro lado, el modelo también podrá hacer el proceso inverso, es decir, tomar una imagen nocturna y reconstruir la escena tal y como se vería durante el día con condiciones soleadas. Esta bidireccionalidad es crucial para preservar las características clave de los objetos y el contexto, independientemente de la iluminación.

- **Modelo Sunny-Cloudy:** En este modelo, el objetivo es transformar imágenes con iluminación soleada en imágenes con un ambiente nublado, y viceversa. Las imágenes soleadas presentan características específicas, como sombras definidas y una paleta de colores más vibrante. El generador en este caso deberá aprender a transformar estos rasgos en aquellos propios de un día nublado, donde las sombras se vuelven mucho más suaves o inexistentes, y los colores pierden saturación, adquiriendo tonalidades más grises o deslavadas. Al igual que el modelo anterior, este también debe ser capaz de hacer el proceso inverso, es decir, a partir de una imagen nublada generar una versión con iluminación soleada. Esto implica entender las diferencias sutiles en la iluminación difusa y directa, y cómo afecta los objetos y el entorno.

- **Modelo Night-Cloudy:** El tercer modelo se encarga de realizar la transformación entre imágenes nocturnas e imágenes nubladas. En este caso, las imágenes nocturnas están caracterizadas por la presencia de luces artificiales, altos contrastes entre las zonas iluminadas y las sombras, y una atmósfera general más oscura. En cambio, las imágenes nubladas suelen ser más uniformes en cuanto a iluminación, con colores apagados y sombras tenues o ausentes. Este modelo tiene el desafío de transformar una escena nocturna, con sus características luces y sombras, en una escena diurna nublada, donde la iluminación natural, aunque tenue, predomina sobre la artificial. De igual manera, el modelo deberá ser capaz de realizar la transformación inversa, generando una imagen nocturna a partir de una imagen diurna nublada.

Cada uno de estos modelos de CycleGAN está compuesto por dos generadores y dos discriminadores, lo que permite realizar la transformación bidireccional entre los distintos tipos de iluminación. El uso de dos generadores es esencial para realizar la transformación en ambas direcciones (por ejemplo, de soleado a nocturno y viceversa), mientras que los dos discriminadores permiten distinguir si las imágenes generadas pertenecen al dominio original o al generado, fomentando que las imágenes transformadas sean realistas y coherentes con las condiciones de iluminación deseadas.

Además, como se ha mencionado previamente, cada tipo de iluminación tiene su propia carpeta de imágenes de referencia. A pesar de que se trata del mismo recorrido y la misma secuencia en los tres entornos, estas imágenes no son exactamente iguales debido a la presencia de objetos o personas que pueden aparecer en diferentes posiciones en cada conjunto. Esto introduce una variabilidad adicional en los datos de entrenamiento, lo cual es positivo, ya que obliga a los modelos a ser más robustos y adaptarse a diferentes contextos y elementos en la escena, sin perder la coherencia en la transformación de iluminación.

Cada entrenamiento tendrá una duración de 100 épocas por modelo, ya que vimos que los resultados no variaron mucho en comparación a las 200 épocas, por tanto ese tiempo de computación que se puede ahorrar nos beneficia.

Por último destacar que cada modelo tiene sus propios pesos y errores asociados a las pérdidas adversarias y cíclicas, ya que no es lo mismo generar imágenes soleadas a partir de imágenes nocturnas que a partir de imágenes nubladas. Por ello también mostraremos las curvas asociadas a la *Cycle consistency loss* del entrenamiento y validación de cada modelo que se han generado al final del entrenamiento.

4.4.1 Modelo *Sunny-Night*

En primer lugar, vamos a mostrar el progreso del entrenamiento en el modelo que transforma imágenes soleadas en nocturnas y al contrario.

Imágenes soleadas a nocturnas

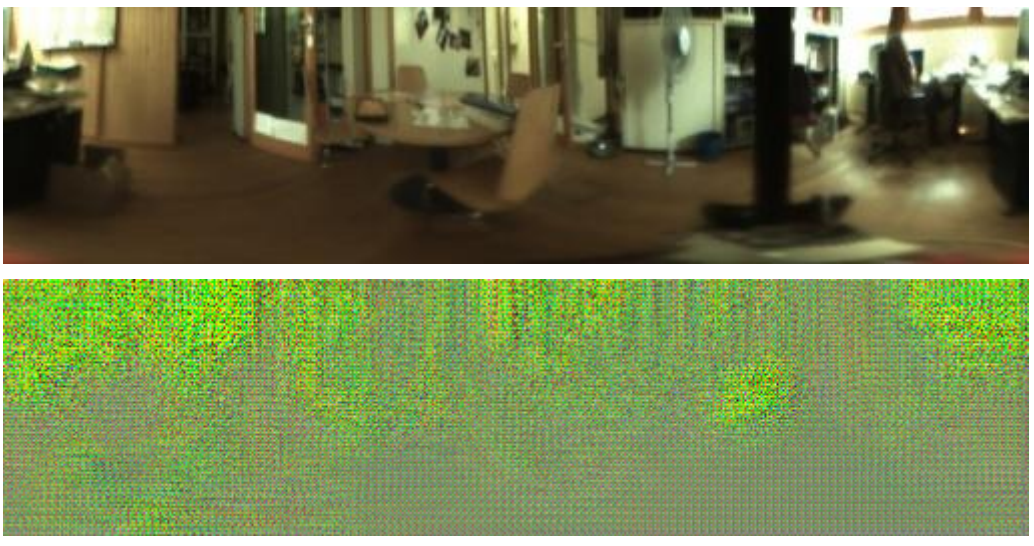


Figura 53: original vs generada en la primera época del entrenamiento

Figura 54: original vs generada en la época número 10

Figura 55: original vs generada en la época número 30

Figura 56: original vs generada en la época número 50

Figura 57: original vs generada en la época número 70

Figura 58: original vs generada en la época número 100

Imágenes nocturnas a soleadas

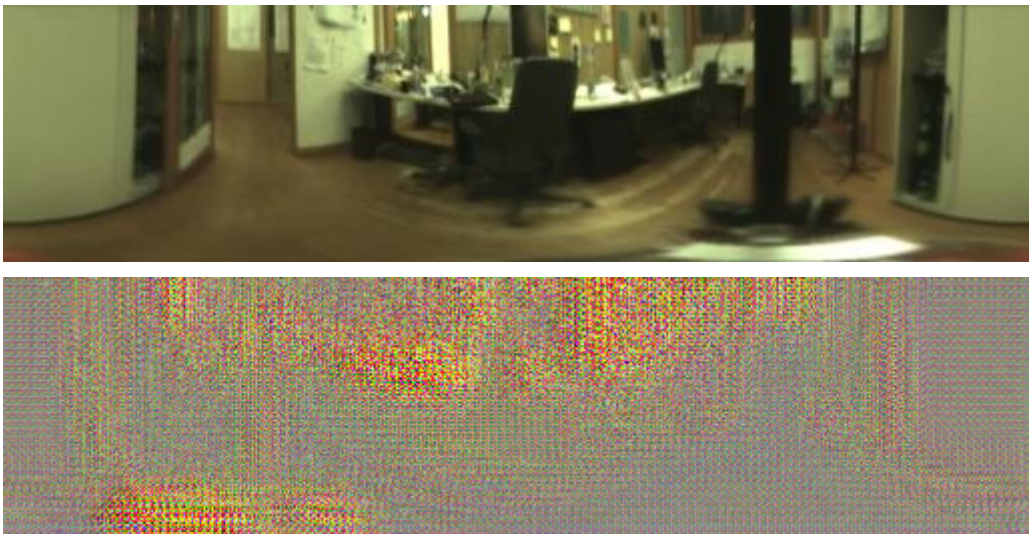


Figura 59: original vs generada en la primera época

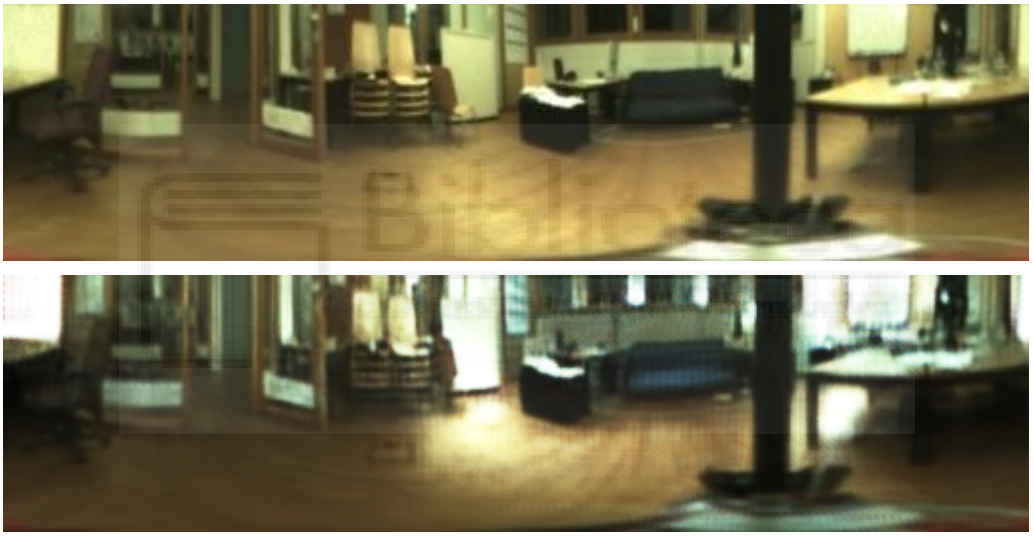


Figura 60: original vs generada en la época número 10

Figura 61: original vs generada en la época número 30

Figura 62: original vs generada en la época número 50

Figura 63: original vs generada en la época número 70

Figura 64: original vs generada en la época número 100

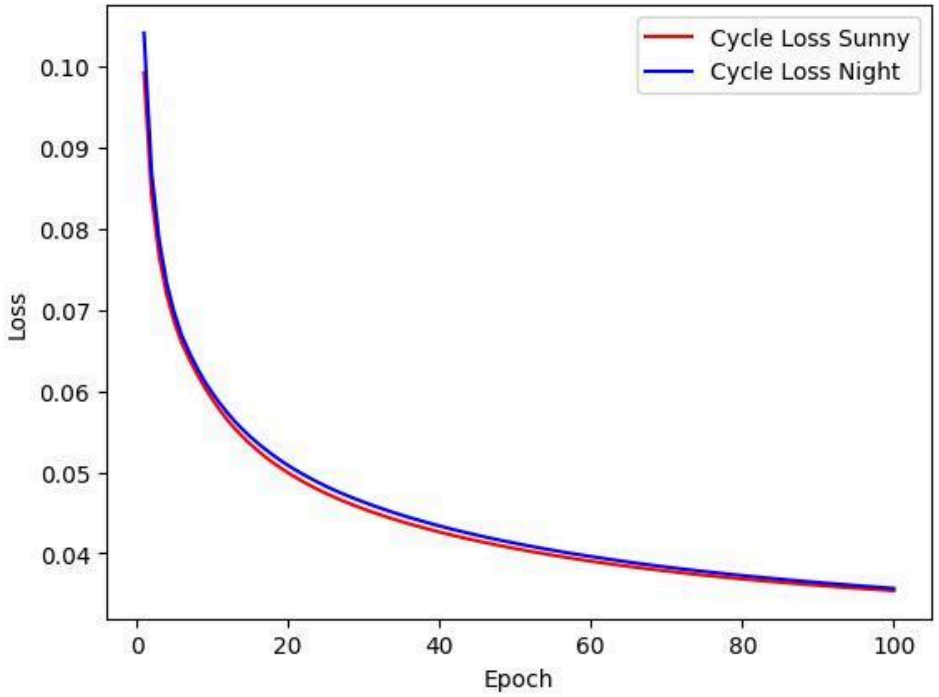


Figura 65: curva del *Cycle consistency loss* del entrenamiento de los generadores *Sunny* y *Night*

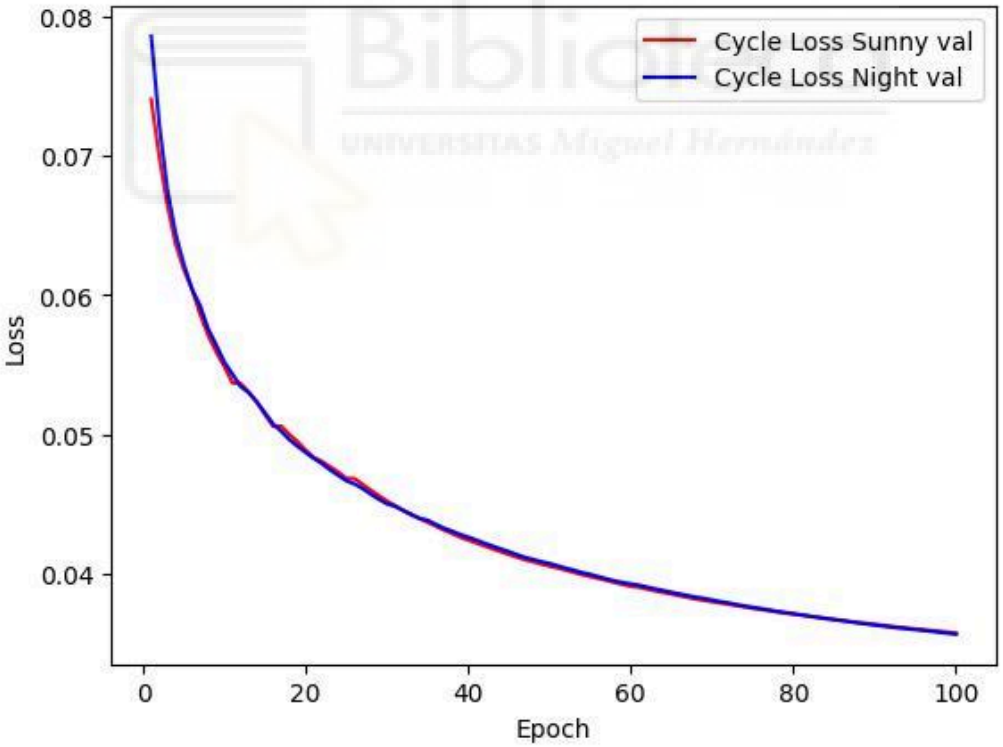


Figura 66: curva del *Cycle consistency loss* de la validación de los generadores *Sunny* y *Night*

A lo largo de los distintos modelos, veremos las gráficas de las figuras 65 y 66, donde vemos la curva descendente de la métrica utilizada *Cycle consistency loss*. Como podemos observar, converge a un valor del 3-4%, lo cual como veremos más adelante de manera visual en los resultados del test significa un valor más que válido y por tanto veremos que no es necesario entrenar más épocas la red y así ahorrarnos tiempo de cómputo. Además

la curva de validación (que se realiza al final de cada época) y la de entrenamiento es prácticamente lo misma, lo que nos garantiza que no hay *Overfitting* y que podemos dar por hecho el entrenamiento, que por muchas épocas más que hagamos no vamos a conseguir una mejoría en los resultados.

También hay que destacar que estas curvas reflejan claramente la evolución que hemos visto en las imágenes anteriores. Se ve que conforme pasan las épocas, las imágenes generadas son de más calidad y se asemejan más a las originales, por lo que vemos como se va cumpliendo el objetivo.

4.4.2 Modelo *Sunny-Cloudy*

Imágenes soleadas a nubladas

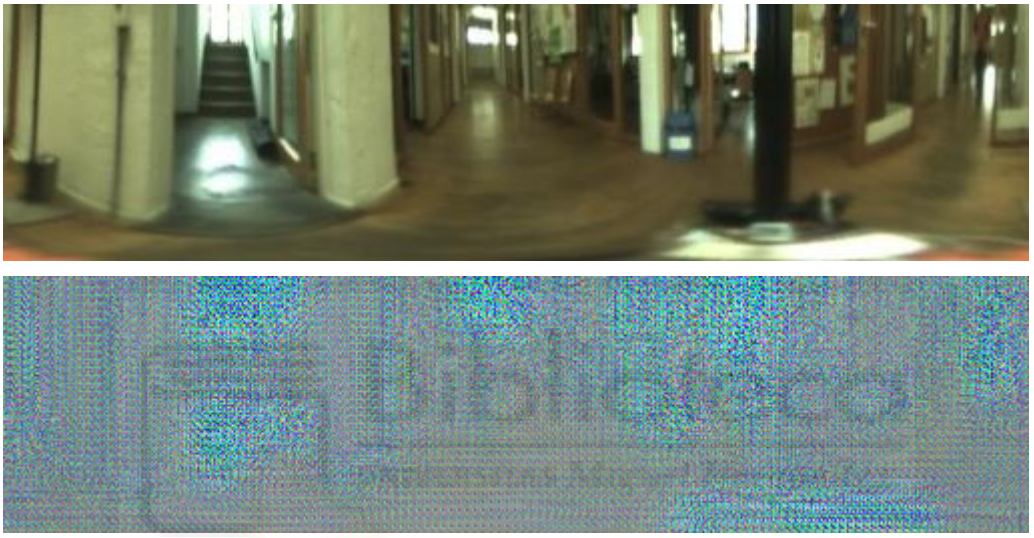


Figura 67: original vs generada en la primera época

Figura 68: original vs generada en la época número 10

Figura 69: original vs generada en la época número 30

Figura 70: original vs generada en la época número 50

Figura 71: original vs generada en la época número 70

Figura 72: original vs generada en la época número 100

Imágenes nubladas a soleadas

Figura 73: original vs generada en la primera época

Figura 74: original vs generada en la época número 10

Figura 75: original vs generada en la época número 30

Figura 76: original vs generada en la época número 50

Figura 77: original vs generada en la época número 70

Figura 78: original vs generada en la época número 100

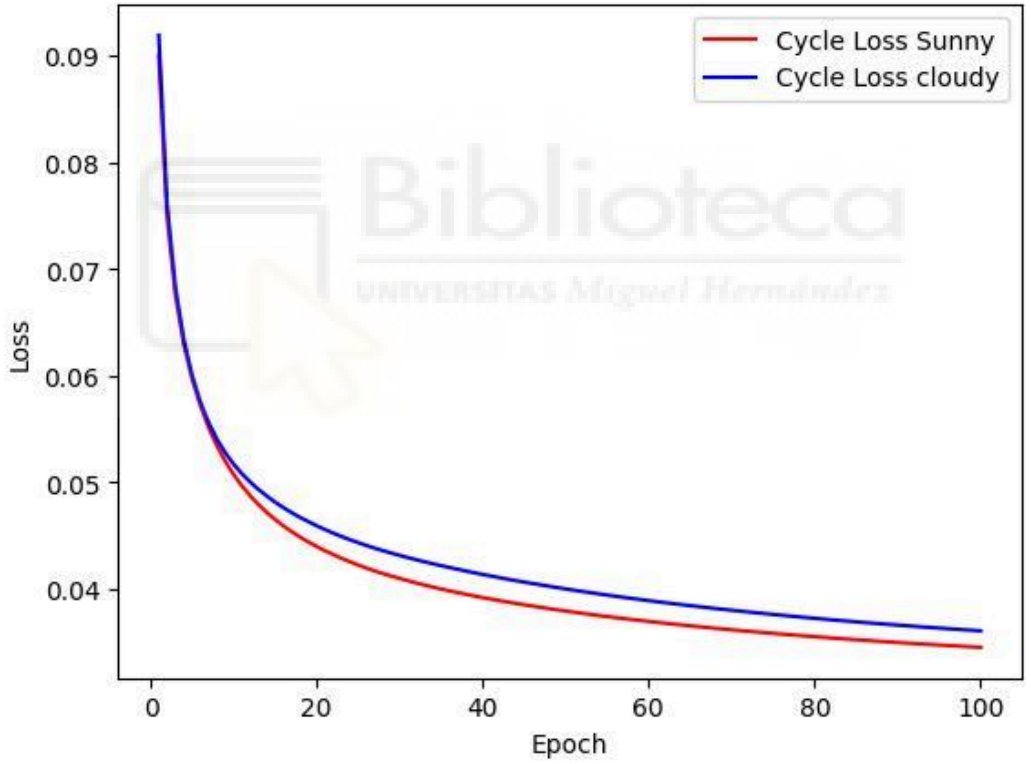


Figura 79: curva del *Cycle consistency loss* del entrenamiento de los generadores *Sunny* y *Cloudy*

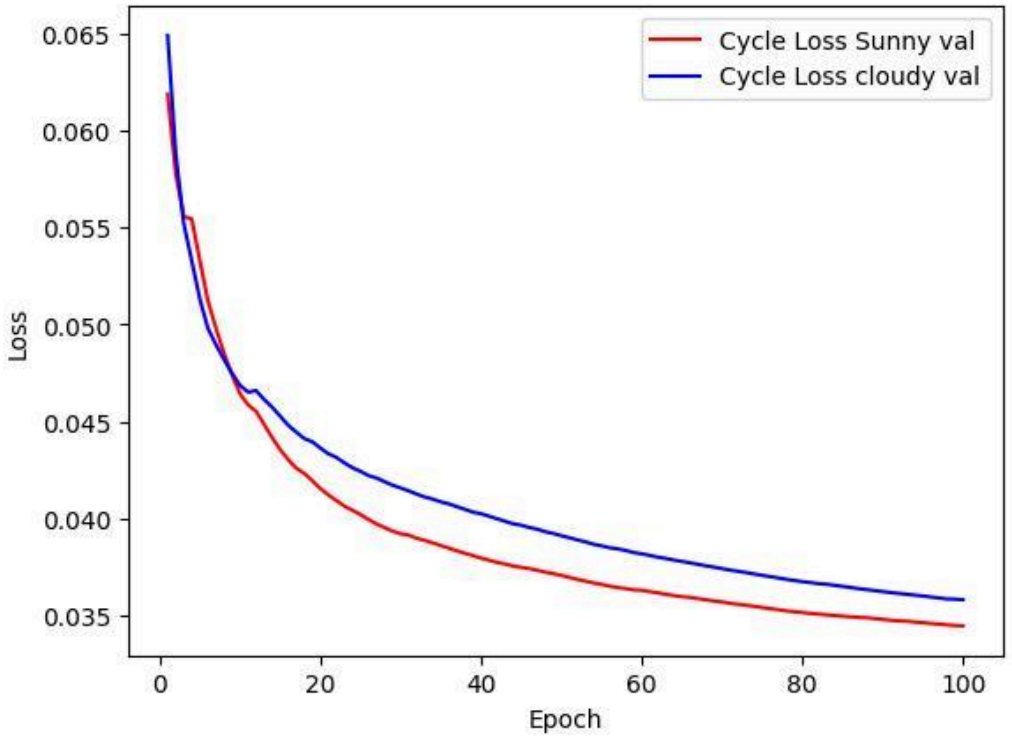


Figura 80: curva del *Cycle consistency loss* de validación de los generadores *Sunny* y *Cloudy*

En estas gráficas encontramos el mismo patrón que en el modelo anterior, es decir, el error está cerca del 3-4%. Por tanto, al igual que el anterior modelo, damos por finalizado el entrenamiento sabiendo que esto nos garantiza un resultado más que correcto.

También nos apoyamos en las imágenes que hemos visto para corroborar que estos resultados tienen sentido y nos sirven para cumplir con el objetivo del trabajo.

4.4.3 Modelo *Night-Cloudy*

Imágenes nocturnas a nubladas

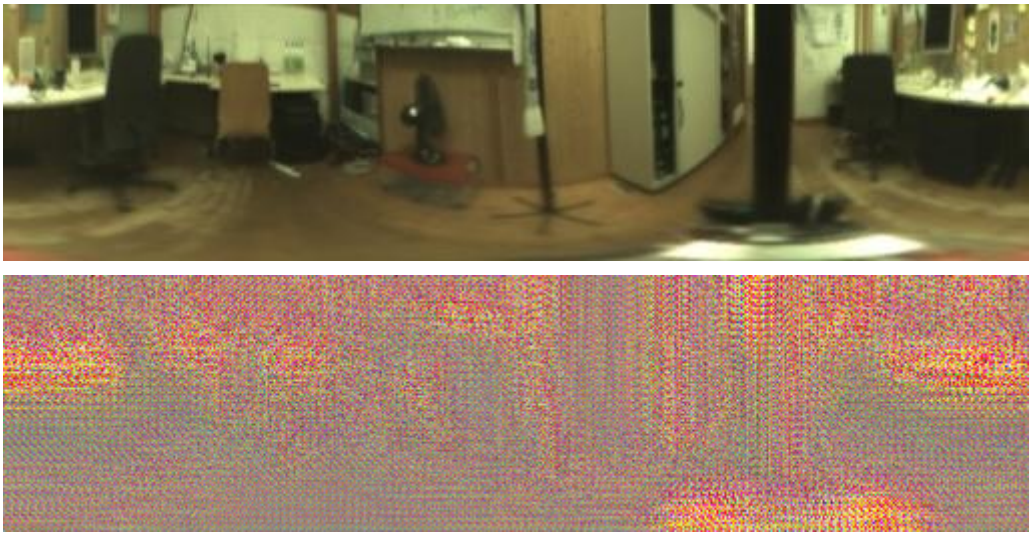


Figura 81: original vs generada en la primera época

Figura 82: original vs generada en la época número 10

Figura 83: original vs generada en la época número 30

Figura 84: original vs generada en la época número 50

Figura 85: original vs generada en la época número 70

Figura 86: original vs generada en la época número 100

Imágenes nubladas a nocturnas

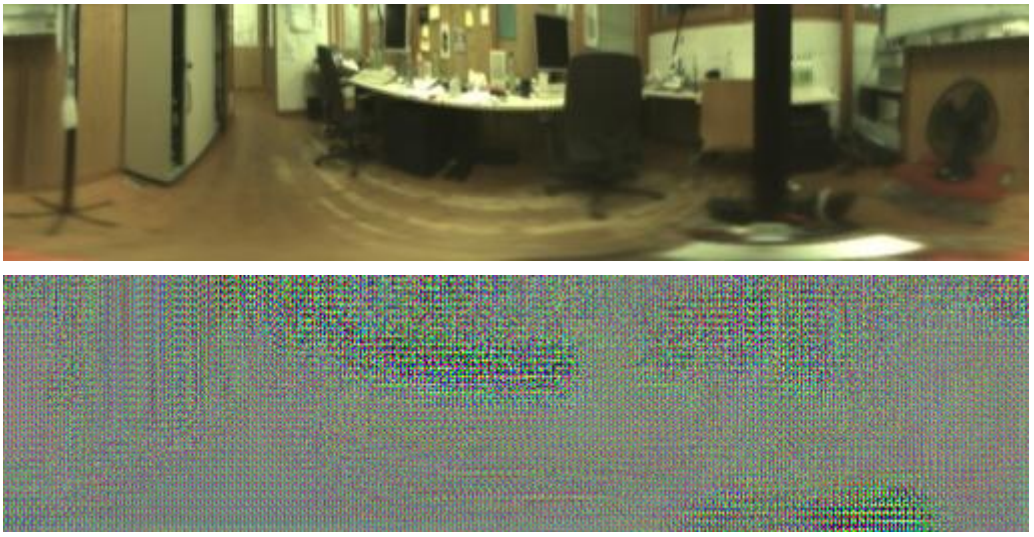


Figura 87: original vs generada en la primera época

Figura 88: original vs generada en la época número 10

Figura 89: original vs generada en la época número 30

Figura 90: original vs generada en la época número 50

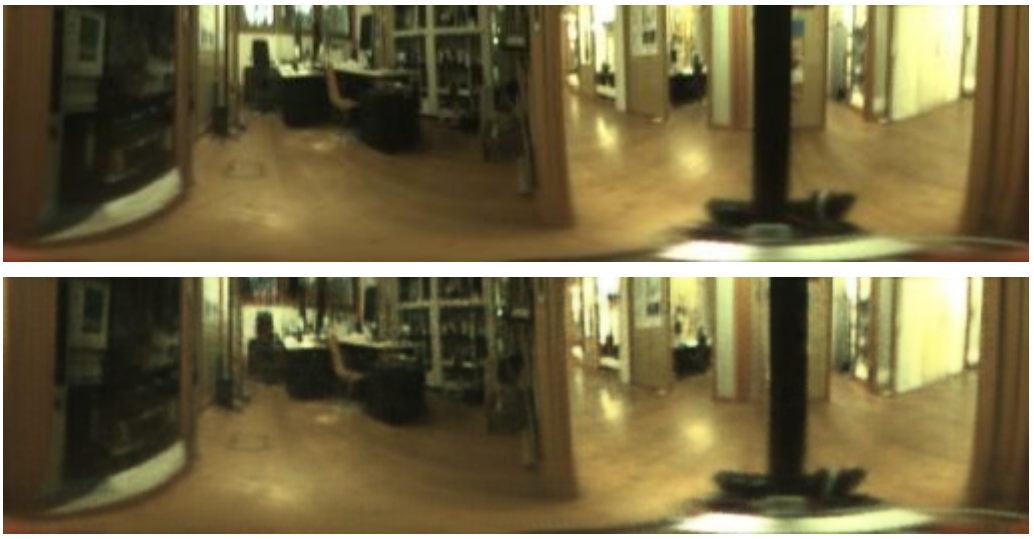


Figura 91: original vs generada en la época número 70

Figura 92: original vs generada en la época número 100

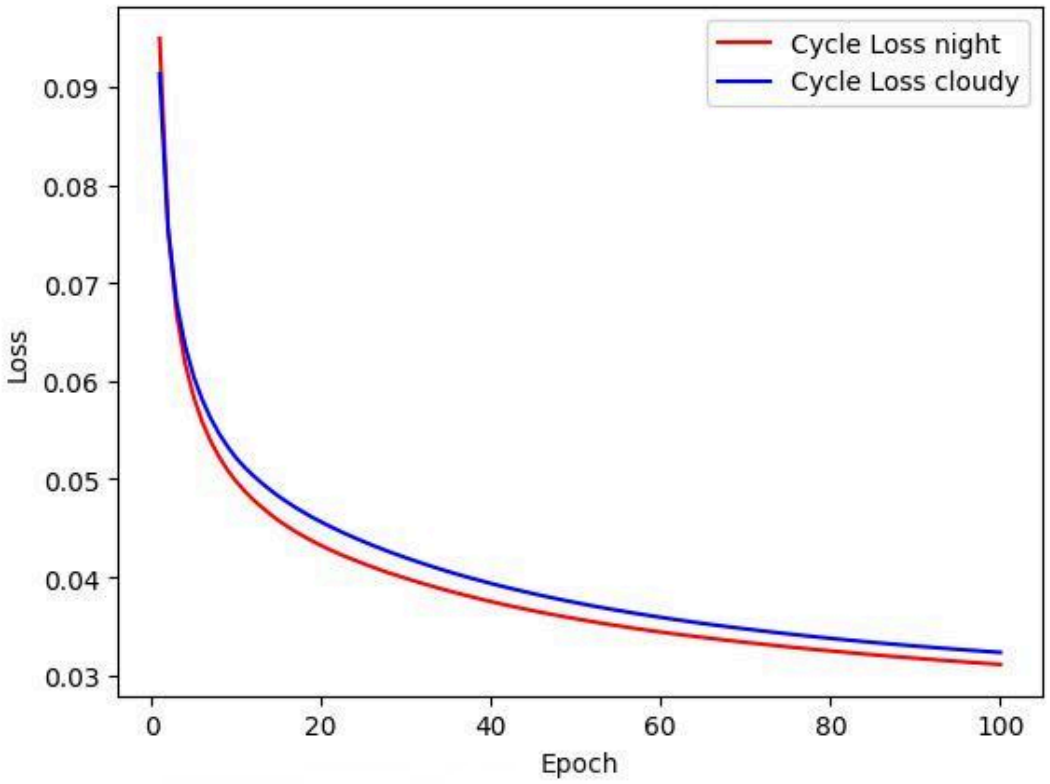


Figura 93: curva del *Cycle consistency loss* del entrenamiento de los generadores *Night* y *Cloudy*

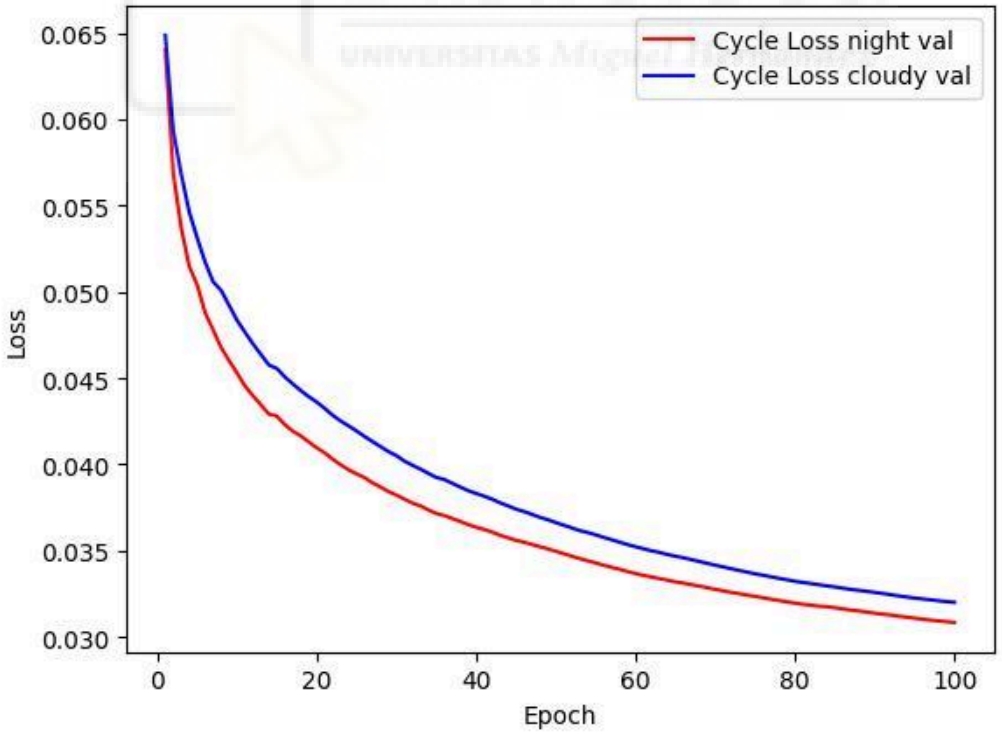


Figura 94: curva del *Cycle consistency loss* de validación de los generadores *Night* y Cloudy

Mismos resultados que en los otros dos modelos, de modo que garantizamos que con 100 épocas podemos entrenar los 3 distintos modelos que podemos estar seguros de los resultados que obtendremos.

4.5 Resultados test

Por último, el código test se encargará de probar los modelos generados y para ello introduciremos otras imágenes distintas y ver que tan bien las genera. La intención de esto es comprobar que no hay *overfitting*. Aquí justificamos también por qué nos quedamos con 100 épocas en vez de 200 y el cycle consistency loss a partir de imágenes reconstruidas y comparándolas con las originales.

4.5.1 Modelo *Sunny-Night*

Transformación de imágenes soleadas a nocturnas

Figura 95: imagen original soleada vs generada nocturna

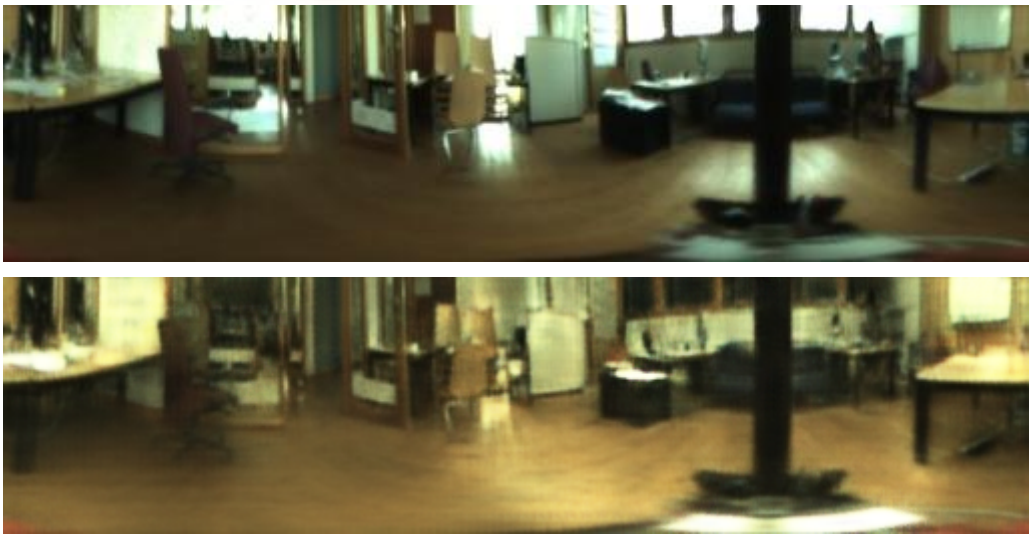


Figura 96: imagen original soleada vs generada nocturna

Figura 97: imagen original soleada vs generada nocturna

Figura 98: imagen original soleada vs generada nocturna

Figura 99: imagen original soleada vs generada nocturna

Figura 100: imagen original soleada vs generada nocturna

Como vemos los resultados son bastante buenos y válidos. Como son imágenes soleadas y nocturnas, el contraste es alto y podemos diferenciar claramente entre las dos imágenes. Para ello podemos fijarnos en los reflejos del suelo o en las ventanas y ver cómo se oscurece, donde en las primeras cuatro parejas de imágenes al haber mejor iluminación en los pasillos y distintas salas, el trabajo es aún mejor que en las dos últimas parejas. Sobre todo en el baño (figura 100), la iluminación es bastante pobre y a la red le cuesta más diferenciar, ya que, por ejemplo, la entrada del medio recibe bastante luz para ser de noche.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Sunny: 0.037 → 3.7%

Figura 101: imagen original soleada utilizada para generar imágenes nocturnas (x)

Figura 102: imagen soleada sintética a partir de la nocturna generada por la soleada original $(x \rightarrow G(x) \rightarrow F(G(x)) \simeq x)$

Con las figuras 101 y 102 vemos claramente lo que significa el *Cycle consistency loss*. El error que obtenemos del 3.7% marca la diferencia entre la imagen soleada original utilizada para generar imágenes nocturnas y la imagen soleada sintética que se genera de la reconstrucción de dicha imagen nocturna sintética. Como podemos observar, las imágenes se parecen mucho, salvo una pequeña bajada de calidad, el nivel de detalles es prácticamente el mismo. Donde mejor se ve esto es en los reflejos del suelo provocados por el sol y la luz que entra por las ventanas. No debemos olvidar que la imagen de abajo antes era nocturna y al hacer soleada otra vez es normal que se pierdan matices de iluminación y color.

Transformación de imágenes nocturnas a soleadas

Figura 103: imagen original nocturna vs generada soleada

Figura 104: imagen original nocturna vs generada soleada

Figura 105: imagen original nocturna vs generada soleada

Figura 106: imagen original nocturna vs generada soleada

Figura 107: imagen original nocturna vs generada soleada

Figura 108: imagen original nocturna vs generada soleada

Aquí como es de esperar, el efecto es el contrario. La idea es aclarar las imágenes para que parezcan diurnas y al igual que en el caso anterior, las imágenes se crean con mejor calidad cuando los factores de iluminación y contraste son mayores (como en las 5 primeras parejas). Sin embargo, en el último par, la imagen no es tan buena al ser un contexto peor de iluminación como es el baño y el pasillo sin ventanas ni nada de iluminación. Aún así parece que la idea la tiene y solo habría que pulir detalles.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Night: 0.032 → 3.2%

Figura 109: imagen original nocturna utilizada para generar imágenes soleadas (y)

Figura 110: imagen nocturna sintética a partir de la soleada generada por la nocturna original $(y \rightarrow F(y) \rightarrow G(F(y)) \simeq y)$

Esta vez la pérdida de consistencia de ciclo es algo inferior y como podemos ver en las figuras 109 y 110 tiene todo el sentido, ya que son imágenes prácticamente idénticas. Sin embargo, hay claras diferencias como en las ventanas de la izquierda, que a pesar de simular que no tienen nada de luz, no las oscurece tanto como su imagen original. Los sofás del final de la sala que se encuentran en el medio de la imagen tampoco son tan negros como el original pero si son de tonos oscuros.

4.5.2 Modelo *Sunny-Cloudy*

Transformación de imágenes soleadas a nubladas

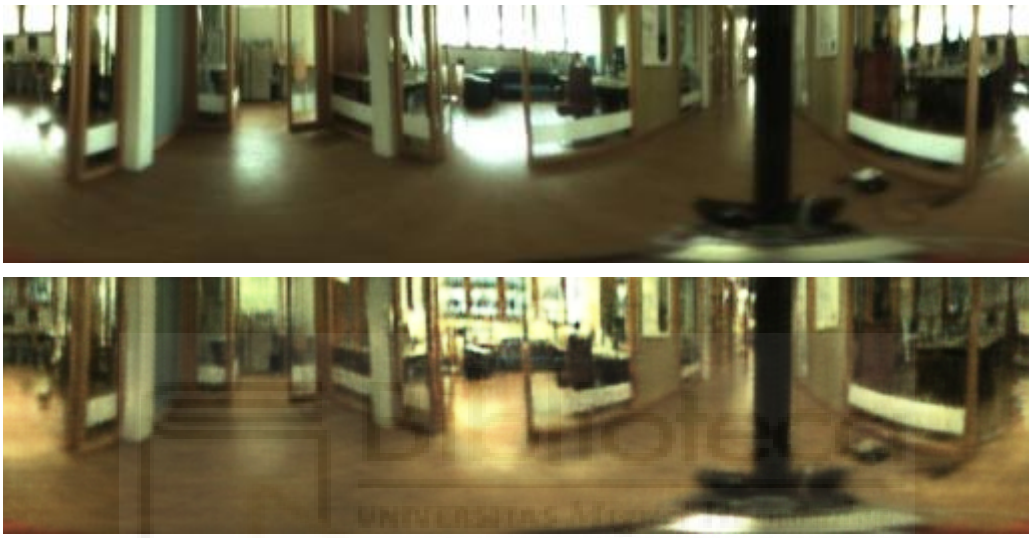


Figura 111: imagen original soleada vs generada nublada

Figura 112: imagen original soleada vs generada nublada

Figura 113: imagen original soleada vs generada nublada

Figura 114: imagen original soleada vs generada nublada

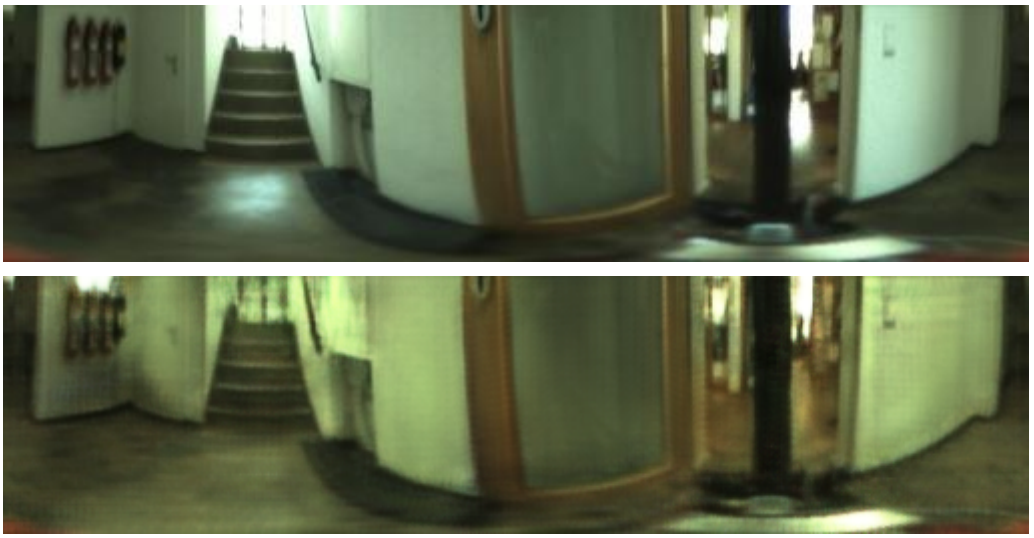


Figura 115: imagen original soleada vs generada nublada

Figura 116: imagen original soleada vs generada nublada

En este caso la tarea es algo más complicada porque no hay tanto contraste como puede haber en una imagen nocturna y diurna. Aun así, vemos a lo largo de los ejemplos que la red se comporta bastante bien quitando iluminación a las escenas sin llegar a oscurecer del todo, cómo se vería en un día nublado. Todo esto lo podemos ver en los reflejos de la luz y en las ventanas, así como en el efecto de la imagen entera que se ve menos clara pero a pesar de ello podemos identificar que sigue siendo de día.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Sunny: 0.039 → 3.9%

Figura 117: imagen original soleada utilizada para generar imágenes nocturnas (x)

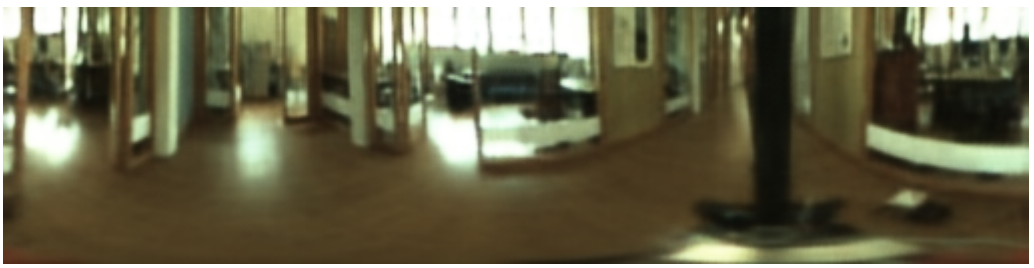


Figura 118: imagen soleada sintética a partir de la nublada generada por la soleada original $(x \rightarrow G(x) \rightarrow F(G(x)) \simeq x)$

En este modelo el generador *sunny* hace un excelente trabajo a la hora de recuperar la imagen original. A pesar del error del 3.9%, no hay apenas diferencias entre las figuras 117

y 118. Si no fuera porque la imagen sintética pierde algo de calidad, no seríamos capaces de identificar las diferencias entre una y otra.

Transformación de imágenes nubladas a soleadas

Figura 119: imagen original nublada vs generada soleada

Figura 120: imagen original nublada vs generada soleada

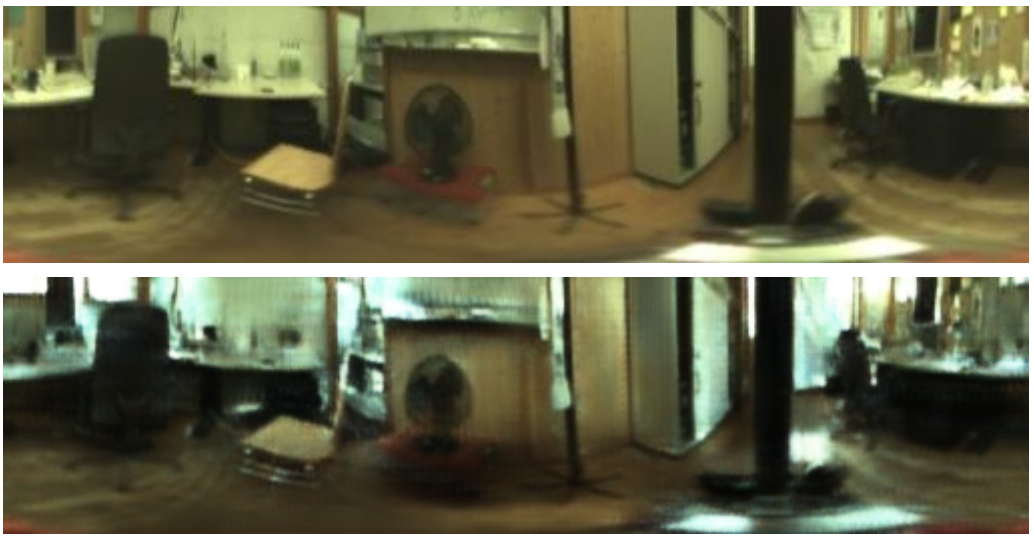


Figura 121: imagen original nublada vs generada soleada

Figura 122: imagen original nublada vs generada soleada

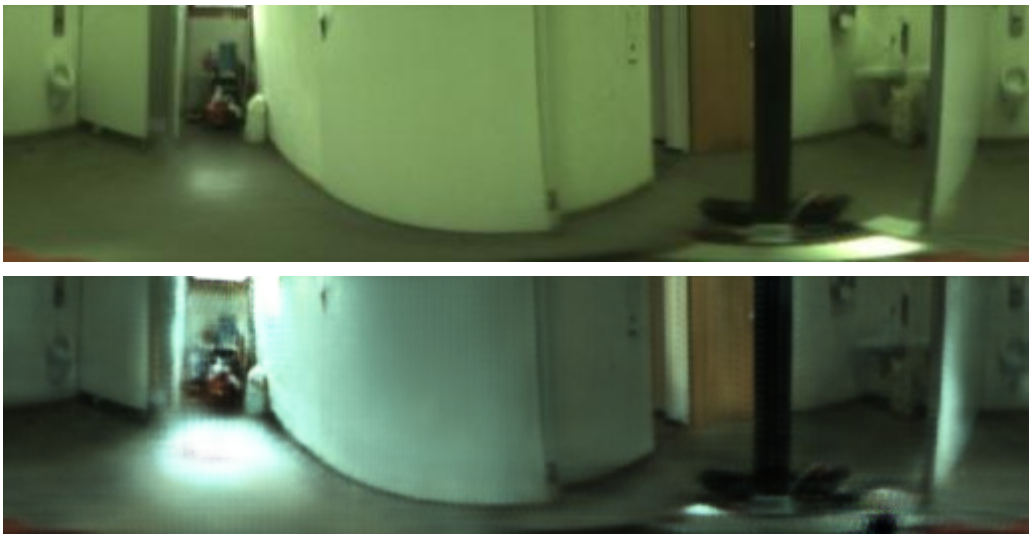


Figura 123: imagen original nublada vs generada soleada

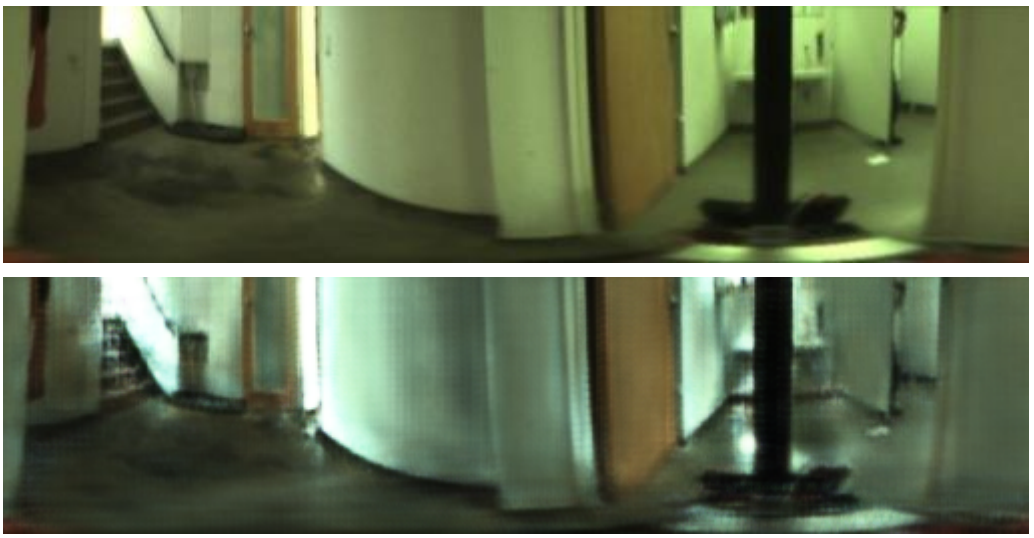


Figura 124: imagen original nublada vs generada soleada

Al igual que en el otro generador, la red trabaja muy bien a la hora de crear imágenes soleadas, donde vemos como hay una mayor iluminación en las escenas y se aumentan los reflejos de la luz tanto en el suelo, como la propia luz que sale por las ventanas. Al igual que en el modelo anterior, la red sufre cuando las condiciones son peores como en la figura 124 y es que como hemos visto y seguiremos viendo, se trata de una zona compleja para que la red trabaje con eficacia.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Cloudy: 0.043 → 4.3%

Figura 125: imagen original nublada utilizada para generar imágenes soleadas (y)

Figura 126: imagen nublada sintética a partir de la soleada generada por la nublada original $(y \rightarrow F(y) \rightarrow G(F(y)) \simeq y)$

A pesar de que el valor de la métrica sea algo mayor que en otros ejemplos, la realidad es que no podemos apreciar apenas diferencias entre las figuras 125 y 126, concluyendo en

que la red realiza un excelente trabajo al recuperar la imagen original. Posiblemente las mayores diferencias se vean en las ventanas donde la iluminación no es exactamente igual pero tanto en los reflejos del suelo como en los detalles de los muebles, el propio robot, etc es prácticamente la misma escena.

4.5.3 Modelo *Night-Cloudy*

Transformación de imágenes nocturnas a nubladas

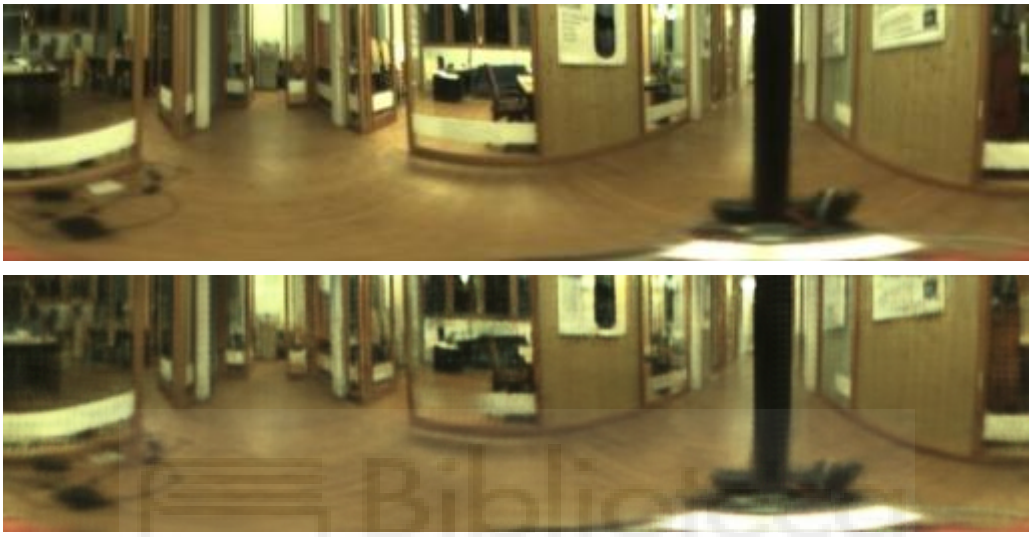


Figura 127: imagen original nocturna vs generada nublada

Figura 128: imagen original nocturna vs generada nublada

Figura 129: imagen original nocturna vs generada nublada

Figura 130: imagen original nocturna vs generada nublada

Figura 131: imagen original nocturna vs generada nublada

Figura 132: imagen original nocturna vs generada nublada

En este modelo llegamos a los peores resultados de todo el trabajo, las transformaciones nocturnas-nubladas y viceversa son claramente las más difíciles de lograr y es que al final la iluminación y las escenas son muy similares, por tanto no hay mucho margen de lograr una gran diferenciación entre ellas. Al final son imágenes que se basan en la poca iluminación de las ventanas y de la escena en general. Aun así nos encontramos la figura 131 que consigue una diferenciación basada en iluminar algo más la escena, llegando a concluir que la imagen de arriba es nocturna y la de abajo diurna pero en un día con menos luz como son los días nublados.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Night: 0.093 → 9.3%

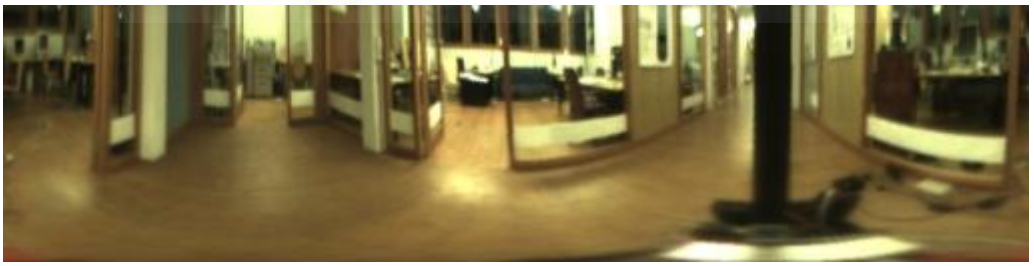


Figura 133: imagen original nocturna utilizada para generar imágenes nubladas (x)

Figura 134: imagen nocturna sintética a partir de la nublada generada por la nocturna original $(x \rightarrow G(x) \rightarrow F(G(x)) \simeq x)$

Con este error logrado en la reconstrucción de la imagen original justificamos la mayor dificultad que encuentra la red a la hora de trabajar con este modelo. Este error que se

obtiene en la fase de testeo de la red nos confirma que la red no consigue diferenciar tanto un contexto de iluminación y otro como si se logra en los dos modelos anteriores. A pesar de ello no deja de ser un valor promedio de todas las imágenes y es por eso que a nivel visual vemos que las imágenes 133 y 134 son muy parecidas y que el trabajo de la red sigue siendo válido.

Transformación de imágenes nubladas a nocturnas

Figura 135: imagen original nublada vs generada nocturna

Figura 136: imagen original nublada vs generada nocturna

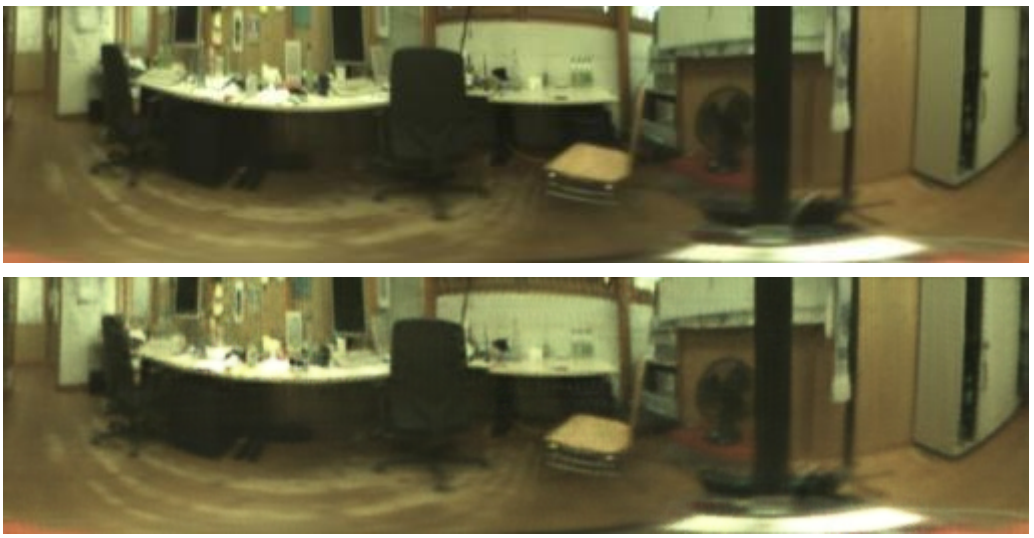


Figura 137: imagen original nublada vs generada nocturna

Figura 138: imagen original nublada vs generada nocturna

Figura 139: imagen original nublada vs generada nocturna

Figura 140: imagen original nublada vs generada nocturna

Aquí por norma general nos encontramos con la misma problemática de no diferenciar contextos lumínicos. Aunque tenemos los ejemplos 136 y 140 que es donde encontramos una mayor diferencias gracias a las ventanas. Si nos fijamos podemos ver como en la imagen nublada hay algo de luz y en las sintética nocturna las oscurece, lo que permite justificar que son imágenes distintas.

Valores conseguidos del *Cycle consistency loss*:

Cycle consistency loss Cloudy: 0.0106 → 10.6%

Figura 141: imagen original nublada utilizada para generar imágenes nocturnas (y)

Figura 142: imagen nublada sintética a partir de la nocturna generada por la nublada original $(y \rightarrow F(y) \rightarrow G(F(y)) \simeq y)$

Aquí obtenemos el mayor error de pérdida de consistencia cíclica y su valor se ve reflejado en la reconstrucción de la imagen por la que hemos optado, donde nos encontramos la mayor diferenciación en unas imágenes que deberían ser lo más parecidas posibles. El

mayor defecto los encontramos en las ventanas donde se ve que no entra la misma luz en una figura y en otra.

5. CONCLUSIONES Y TRABAJOS FUTUROS

Como hemos descrito a lo largo de este trabajo, nuestro objetivo principal era el de aumentar una base de datos que permita al robot recopilar más información y estar preparado ante distintos contextos de un mismo entorno. Por ese motivo, nuestro foco siempre ha sido el de comprobar de manera visual que las imágenes generadas por nuestra red *Cyclegan* eran válidas para cumplir dicho objetivo. Y es que a partir de las imágenes procedentes de la base de datos *Cold* teníamos que entrenar las dos redes que compiten dentro de la arquitectura *Cyclegan* con la idea de que la imagen sintética creada por el generador tenía que ser lo suficientemente buena para que le discriminador la tomara por real, y a su vez el discriminador tiene que ser lo suficientemente potente para que no cualquier imagen le sirva. Para ello comparamos imágenes originales con generadas y podemos decir con seguridad que son bastante parecidas.

Para que el trabajo sea lo mejor posible, hemos tomado distintas imágenes tanto para el entrenamiento como para la validación como para el testeo final. De este modo, nos aseguramos de evitar sobreentrenamiento y de que optimizamos los tiempos de computación para que el modelo generado sea el mejor.

En el trabajo se han presentado los experimentos realizados, donde se muestra tanto una evaluación cuantitativa como cualitativa. Como se ha podido apreciar, los entrenamientos han ido mejorando su rendimiento conforme han pasado las épocas, pero como se indican en las gráficas de la métrica utilizada, se llega a un punto en el que por más épocas que entrenemos no se disminuye más la pérdida de consistencia. Por tanto, es innecesario aumentar el número de épocas y gastar tiempo de computación.

Una vez visualizados los resultados de la función test, fijándonos en las ventanas y en los reflejos del suelo, llegamos a la conclusión que claramente los modelos *Sunny-Night* y *Sunny-Cloudy* son los que mejores han trabajado a diferencia del modelo *Night-Cloudy*, en el cual no se aprecian muchos cambios de iluminación de una imagen generada del dominio X a otra del dominio Y. Esto no es casualidad, y es que el factor del nivel de detalle de las distintas iluminaciones es crucial para el mejor funcionamiento, ya que hay mayor contraste en una imagen soleada con otra nublada o nocturna, a una imagen nublada y nocturna las cuales son bastante similares.

Aún así los resultados son bastante buenos y se cumple el objetivo de aumentar considerablemente la base de datos, ya que a pesar de que en esta memoria solo se muestran ejemplos de habitaciones sueltas para cada modelo, la red consigue generar imágenes con cambio de dominio para todas las imágenes que le metemos como entrada. Bien es cierto que no todas las imágenes son buenas y no consiguen el resultado esperado, pero en su gran mayoría si es así.

A continuación, podemos ver una comparativa directa entre imágenes originales procedentes de la base de datos *Cold* con imágenes sintéticas creadas por nuestra red:

Figura 143: imágenes soleada, nublada y nocturna originales

Figura 144: imágenes soleada, nublada y nocturna generadas

Visualmente se aprecia que los matices de colores, iluminación e incluso calidad de imagen son bastante parecidas. Por lo que podemos decir con total seguridad que el trabajo de la red *Cyclegan* cumple con su propósito.

Sin embargo, a pesar de que es una red muy potente y cumple con su función, los modelos entrenados solo se pueden aplicar para imágenes similares a las entrenadas, es decir que sean en entornos de oficina. Si intentamos meter cualquier imagen panorámica para que nos haga el mismo cambio de iluminación pero en un entorno distinto, no se consigue un resultado muy óptimo:

Figura 145: imagen panorámica soleada original

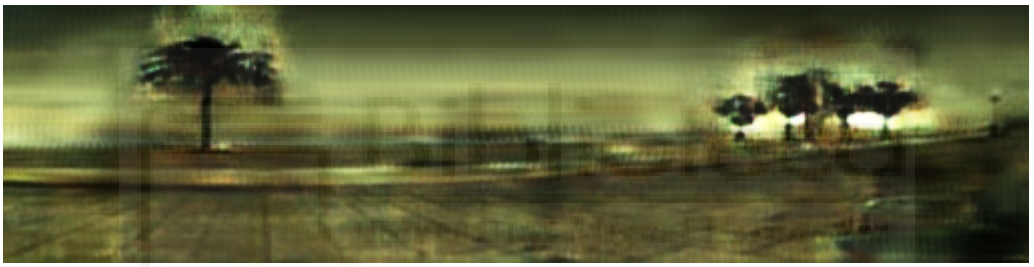


Figura 146: imagen nocturna sintética generada a partir de la soleada original

A nivel cuantitativo también se nota el mal funcionamiento, ya que el *Cycle consistency loss* obtenido es de un 0.474, es decir, de un 47.4% que comparado con el 3.7% conseguido en las pruebas, significa un aumento bastante considerable.

En términos de trabajo futuro, existen varias direcciones claras en las que se puede expandir esta investigación para mejorar tanto la capacidad del modelo como su aplicación práctica. A continuación, se detallan algunas de estas posibles líneas de desarrollo:

Una posible línea sería el desarrollo de modelos más potentes y sofisticados que puedan realizar cambios de iluminación selectivos y específicos en cualquier tipo de imagen. Por ejemplo, en un escenario como el de la fotografía panorámica de la playa, se buscaría que la red pueda diferenciar automáticamente entre diferentes elementos de la escena, tales como el cielo, el mar y la tierra. Una vez identificados estos elementos, el modelo debería ser capaz de modificar exclusivamente la iluminación de uno de ellos (por ejemplo, oscurecer solo el cielo para simular el atardecer), sin alterar los demás.

Para lograr esto, sería necesario entrenar la red con una gran cantidad de datos variados que incluyan múltiples condiciones de iluminación, diferentes tipos de paisajes y elementos visuales. Además, esto implicaría disponer de un equipo de hardware potente capaz de soportar el entrenamiento intensivo que requiere manejar grandes volúmenes de datos y

realizar ajustes finos en el modelo. Esta capacidad de cambiar de manera precisa y localizada la iluminación en las imágenes podría ser una herramienta valiosa en aplicaciones de fotografía y edición automática de imágenes.

Otra posible línea de trabajo sería integrar imágenes emparejadas en el entrenamiento de la red, es decir, pares de imágenes que representan la misma escena bajo diferentes condiciones de iluminación. La idea detrás de esto es explorar si el uso de estos pares puede ayudar a mejorar los resultados, especialmente en términos de calidad y precisión de la manipulación de la iluminación.

La métrica de pérdida de consistencia de ciclo seguiría siendo un factor crucial en esta etapa. Hasta ahora, se ha logrado un valor bajo en esta métrica, lo cual indica que el modelo tiene un buen desempeño al transformar imágenes de ida y vuelta sin perder información significativa. Sin embargo, reducir aún más este valor es un desafío considerable. El objetivo sería acercarse lo más posible a una equivalencia perfecta entre la imagen original y su versión sintetizada, lo que implica que ambas deben ser prácticamente indistinguibles en cuanto a calidad y detalles.

Finalmente, es esencial evaluar si las imágenes sintéticas generadas por este modelo tienen un valor práctico en otros contextos de aprendizaje automático, particularmente en tareas relacionadas con *SLAM* (*Simultaneous Localization and Mapping*). Si las imágenes sintéticas pueden ser utilizadas eficazmente como datos de entrenamiento para redes de SLAM, esto significaría que el modelo no solo es capaz de generar imágenes visualmente coherentes, sino que estas también contienen características relevantes y útiles para mejorar el desempeño de sistemas robóticos.

El objetivo aquí sería probar que estas imágenes no solo son visualmente atractivas, sino también funcionales, contribuyendo a la creación de modelos más robustos y eficientes. Si se demuestra que los datos sintéticos ayudan a los robots a adaptarse mejor a diferentes condiciones de iluminación y entornos visuales, esto representaría un avance significativo en aplicaciones de robótica autónoma, aumentando su precisión y velocidad en la navegación.

Estos desarrollos no solo contribuirían a la mejora de las capacidades actuales del modelo, sino que también abrirían nuevas oportunidades para su aplicación en distintos campos, desde la edición de imágenes, lo cual supone un aumento de las bases de datos, hasta la robótica avanzada.

6. BIBLIOGRAFÍA

[1] https://es.mathworks.com/discovery/slam.html
[2] https://es.mathworks.com/help/nav/ug/implement-simultaneous-localization-and-mapping-with-lidar-scans.html
[3] https://new.abb.com/news/es/detail/113774/prsrl-abb-presenta-su-innovador-robot-movil-con-tecnologia-visual-slam-y-amr-studio-suite
[4] Jordi Delatorre, “Redes Generativas Adversarias (GAN). Fundamentos teóricos y aplicaciones”, 18 febrero de 2023.
[5] https://datascientest.com/es/machine-learning-definicion-funcionamiento-usos
[6] https://www.codificandobits.com/blog/que-es-una-red-neuronal/
[7] https://www.codificandobits.com/blog/redes-convolucionales-introduccion/
[8] https://www.codificandobits.com/blog/redes-adversarias-explicacion-y-tutorial-python/#qu%C3%A9-son-las-redes-adversarias-una-idea-general
[9] https://inteligenciaartificial360.com/fundamentos-ia/redes-adversarias-generativas-gan-fundamentos-y-aplicaciones/
[10] https://techxplore.com/news/2018-12-nvidia-face-making-approach-genuinely-gan-tastic.html#google_vignette
[11] Mehedi Mirza and Simon Osindero, Conditional generative adversarial nets, CoRR,abs/1411.1748, 2014.
[12] https://learnopencv.com/conditional-gan-cgan-in-pytorch-and-tensorflow/
[13] https://keepcoding.io/blog/que-son-redes-generativas-antagonicas/#Aplicaciones_de_las_redes_generativas_antagonicas
[14] https://paperswithcode.com/method/instance-normalization
[15] https://www.researchgate.net/figure/DCGAN-Deep-Convolutional-Generative-Adversarial-Network-generator-used-for-LSUN_fig1_340884113
[16] Jun-Yan Zhu, Taesung Park, “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, arXiv:1703.10593, 2017.
[17] https://paperswithcode.com/method/instance-normalization
[18] https://paperswithcode.com/method/relu
[19] https://paperswithcode.com/method/patchgan
[20] https://medium.com/@chilldenaya/cyclegan-introduction-pytorch-implementation-5b53913741ca
[21] https://paperswithcode.com/method/leaky-relu
[22] https://jacar.es/la-funcion-sigmoide-una-herramienta-clave-en-redes-neuronales/
[23] https://aws.amazon.com/es/what-is/data-augmentation/

[24] https://www.linkedin.com/advice/0/what-best-practices-using-generative-adversarial-1awbf?lang=es&originalSubdomain=es
[25] A. Pronobis and B. Caputo. COLD: COsy Localization Database. *The International Journal of Robotics Research (IJRR)*, 28(5), May 2009.
[26] https://aws.amazon.com/es/what-is/python/
[27] https://datascientest.com/es/pycharm#:~:text=%C2%BFQu%C3%A9%20es%20PyCharm%3F,%2C%20Facebook%2C%20Amazon%20y%20Pinterest.
[28] https://www.ibm.com/es-es/topics/pytorch
[29] https://aadhil-imam.medium.com/introduction-pytorch-tensors-tutorial-01-9dca5b4a1590
[30] https://www.cas.kth.se/COLD/downloads.html
[31] https://www.growupcr.com/post/metricas-precision
[32] https://www.comet.com/site/blog/5-regression-loss-functions-all-machine-learners-should-know/
[33] https://blog.javierheras.website/hiperparametros-de-una-red-neuronal/