ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso académico 2022-2023

Trabajo Fin de Grado

Aprendizaje Automático para la Generación de Texturas de Videojuegos utilizando VAEs

Autor: Miguel Herrero Montiel

Tutor: Iván Ramírez Díaz

Cotutor: Diego Hortelano Haro

Agradecimientos

Quiero agradecer sobre todo a mi familia por ser ese apoyo incondicional durante toda mi vida. Por darme la oportunidad de estudiar y apoyarme a lo largo del camino

A mis amigos porque sin ellos no sería quién soy, por ayudarme a desconectar y por darme fuerzas en todo momento.

A mi hermano por ser ese ejemplo a seguir desde que tengo uso de razón y por saber cuidarme como nadie sabe.

A mi novia por motivarme a ser mejor cada día, brindándome felicidad aún cuando todo rema en contra.

A mis tutores por haberme guiado en este complicado viaje de adentrarse en el mundo de la Inteligencia Artificial

Muchas Gracias a todos

Madrid, 25 de junio de 2023

Miguel Herrero Montiel

Resumen

El propósito de este trabajo es entrenar un modelo de inteligencia artificial, específicamente un Variational Autoencoder (VAE), para generar texturas de videojuegos. Se ha adquirido un conocimiento desde cero sobre el funcionamiento de las redes neuronales y del modelo generativo VAE. Se ha elegido Python como lenguaje de programación para aplicar los conocimientos adquiridos, y se ha elegido Minecraft como el videojuego para la generación de texturas debido a la familiaridad con el mismo y a la amplia variedad de texturas y skins disponibles. Tras la generación de dichas texturas se han creado paquetes con las imágenes generadas y se han incorporado al juego para poder ver los resultados de nuestro trabajo.

Palabras clave: Aprendizaje automático, redes neuronales, red convolucional, autoencoder, VAE, MNIST, CIFAR10, Minecraft, texturas, skins.

Abstract

The objective of this project is to train an artificial intelligence model, specifically a Variational Autoencoder (VAE), for generating textures in video games. Extensive knowledge has been acquired on the workings of neural networks and the generative model VAE from scratch. Python has been selected as the programming language to implement this knowledge, and Minecraft has been chosen as the target video game for texture generation due to its familiarity and diverse range of textures and skins available. Following the texture generation process, packages containing the generated images have been created and integrated into the game to observe the outcomes of our efforts.

Keywords: Machine learning, neural networks, convolutional network, autoencoder, VAE, MNIST, CIFAR10, Minecraft, textures, skins.

Índice de Contenidos

1. Introducción **1**

1.1. Objetivos 3

2. Aprendizaje Automático **5**

2.1. El aprendizaje supervisado 5

2.2. El aprendizaje no supervisado 6

2.3. El aprendizaje por refuerzo 7

3. Redes neuronales **8**

3.1. Redes neuronales monocapa 9

3.2. Redes neuronales multicapa 10

3.3. Redes neuronales Recurrentes 11

3.4. Redes neuronales Convolucionales 12

3.4.1. Capa de Entrada 14

3.4.2. Capa convolucional 14

3.4.3. Capa de pooling 16

3.5. Entrenamiento Redes Neuronales 16

4. Modelos Discriminativos vs Modelos Generativos **19**

5. Variational Autoencoder **22**

5.1. Autoencoder 22

5.2. VAE 23

6. Tecnología y creación del código **25**

6.1. Tecnología 25

6.2. Proceso 26

6.3. Pruebas iniciales 27

6.3.1. Número de épocas 27

6.3.2. Tasa de aprendizaje 27

6.3.3. Función de pérdida 28
6.3.4. Capas de la red 28
6.3.5. Arquitecturas más complejas 28

7. Aplicación **31**
7.1. Elección de Videojuego 31
7.2. Bloques 32
7.2.1. Conjunto de datos 33
7.2.2. Ajustes 33
7.2.3. Pruebas 34
7.3. Aspectos 34
7.3.1. Conjunto de datos 36
7.3.2. Ajustes 36
7.3.3. Pruebas 36
7.4. Generacion con VAE convolucional 37
7.4.1. Bloques 37
7.4.2. Skins 38

8. Resultados **39**
8.1. Bloques 39
8.1.1. Incorporación al juego 39
8.2. Skins 42
8.2.1. Incorporación al juego 43
8.3. Código del proyecto 44

9. Conclusiones **45**

Bibliografía **47**

Índice de Figuras

1.1. Reconstrucción de caras por un Autoencoder [19] 3

3.1. Neurona biológica [32] 8
3.2. Relación neurona [3] 8
3.3. Red neuronal monocapa[3] 9
3.4. Funciones de activación [5] 10
3.5. Red neuronal Multicapa[3] 10
3.6. Red neuronal Recurrente[3] 12
3.7. Representación de las capas en el tiempo[3] 12
3.8. Red neuronal Multicapa 13
3.9. Imagen demostración 13
3.10. Red convolucional[20] 14
3.11. Proceso de convolución [17] 15
3.12. obtención mapa características[6] 15
3.13. Estructura red convolucional[12] 16

5.1. Estructura Autoencoder[30] 23
5.2. Estructura Autoencoder Variacional[10] 24

6.1. Resultados obtenidos en la prueba inicial utilizando el conjunto de datos MNIST 27
6.2. Resultados obtenidos al aumentar el número de épocas 28
6.3. Cambiar tasa de aprendizaje 28
6.4. Cambiar función de pérdida 29
6.5. Cambiar número de capas 29
6.6. Prueba inicial CIFAR10 29
6.7. Prueba final CIFAR10 30

7.1. Estructura de los bloques en Minecraft[18] 32
7.2. Bloques por defecto Minecraft [29] 33

7.3. Primera prueba Texturas 34
7.4. Patrones bloques 34
7.5. Estructura Skins Minecraft [28] 35
7.6. Aspecto por defecto en Minecraft [31] 35
7.7. Primeras muestras generadas 36
7.8. Ejemplo de aspectos generados 37
7.9. Primeras generaciones de bloques con VAE convolucional 37
7.10. Primeras generaciones de bloques con VAE convolucional 38

8.1. Tipos de bloque 39
8.2. Texturas originales incorporadas en el juego 40
8.3. Texturas seleccionadas para crear el paquete 40
8.4. Texturas generadas incorporadas en el juego 41
8.5. Texturas generadas incorporadas en el juego de cerca 42
8.6. vista previa de aspectos generados 43
8.7. aspecto generado incorporado en el juego 44

Índice de Tablas

4.1. Tabla comparativa Discriminativos vs Generativos 20

Capítulo 1
Introducción

En los últimos años, el campo de la inteligencia artificial (IA) ha experimentado un crecimiento exponencial. Diariamente, escuchamos en las noticias nuevos avances y hazañas que se están logrando en este campo. La IA está revolucionando el mundo de la tecnología y haciéndonos conscientes de su gran potencial. Pero, ¿qué es la IA? La IA es el campo que se encarga de crear máquinas o sistemas capaces de realizar tareas que requieren inteligencia humana. La IA trata de emular y automatizar procesos cognitivos y de toma de decisiones propios de los seres humanos, como el razonamiento, el aprendizaje, la percepción, la comprensión del lenguaje natural y la resolución de problemas.

Sus impresionantes avances estos últimos años han generado dudas y miedo ante la posibilidad de que una IA reemplace la labor humana en muchos empleos. Áreas artísticas como el diseño, la música o la escritura han sido algunos de los sectores que más se han visto afectados por esta revolución. Existe un gran dilema sobre dichos avances que no comentaremos en este trabajo. Vamos a desglosar los temas tratados en este documento.

El aprendizaje automático o *machine learning* es una rama de la IA que se centra en desarrollar algoritmos y modelos que permiten a las máquinas aprender y mejorar su rendimiento sin ser específicamente programados para ello. Cómo su propio nombre dice, aprenden automáticamente a partir de unos datos de entrada. Hay diferentes tipos de aprendizaje automático que desarrollaremos en este trabajo: supervisado, no supervisado y por refuerzo.

Uno de los modelos de aprendizaje automático más conocido son las redes neuronales. Simulan el funcionamiento de las redes neuronales biológicas del cerebro humano de manera computacional. Las redes neuronales artificiales llevan tiempo entre nosotros, pero no ha sido hasta el desarrollo de nuevas técnicas de entrenamiento y una mejora de hardware y de capacidad computacional que no han ganado la popularidad que tienen

ahora. Existen numerosos tipos, pero en este trabajo hablaremos de las redes monocapa, multicapa, recurrentes y sobre todo las convolucionales por su importancia en la visión artificial.

Dentro de esta revolución, la generación de contenido visual con la creación de imágenes y vídeos cada vez más realistas ha despertado un gran interés. Concretamente, modelos como DALL-E 2 [25] o Stable Diffusion [4] han causado un gran impacto por su capacidad de generar imágenes de altísima calidad a partir de un texto de entrada. Todo esto se engloba dentro del campo de la IA de la Visión artificial, que es la disciplina que se encarga de desarrollar sistemas o algoritmos capaces de comprender y generar contenido visual.

En este contexto, cobran importancia los modelos discriminativos y los modelos generativos. Los modelos discriminativos buscan encontrar una función que permita clasificar datos de entrada, siendo muy útiles en la clasificación de imágenes. Los generativos, por otro lado, modelan la distribución de probabilidad conjunta de las variables de entrada y salida aprendiendo a generar nuevas muestras, pudiendo crear nuevas imágenes.

En este trabajo vamos a centrarnos en un modelo generativo muy popular, el Variational Autoencoder (VAE), que ha emergido como una poderosa herramienta para la generación de imágenes. Los VAEs surgen de los autoencoders que son un tipo de red neuronal artificial entrenada de manera no supervisada, cuyo objetivo es codificar una entrada, reduciendo su tamaño para que posteriormente el modelo aprenda a reconstruirla con un decodificador. Los VAEs combinan la capacidad de los autoencoders, con técnicas de inferencia variacional que lo convierten en un poderoso modelo generativo. Una idea de su funcionamiento se puede ver reflejado en la Figura 1.1.

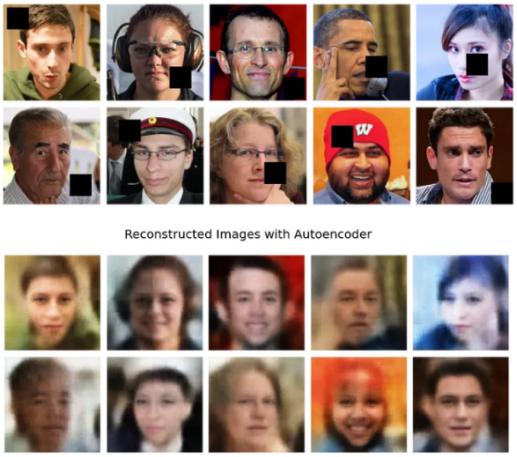


Figura 1.1: Reconstrucción de caras por un Autoencoder [19]

El estado del arte de los VAEs ha demostrado su utilidad en diferentes aplicaciones como generación de imágenes de alta calidad, mejora de la calidad de imágenes y restauración, aplicaciones en medicina y ciencias biológicas, etc. Algunos ejemplos son la generación de rostros realistas [8], diagnósticos médicos [22], detección de noticias falsas [21], etc.

1.1. Objetivos

El presente trabajo de fin de grado tiene como objetivo principal el proporcionar una explicación didáctica del funcionamiento de los Variational Autoencoders (VAEs) para que pueda ser comprendido por otras personas. Además, se busca encontrar una aplicación práctica para aprovechar las capacidades de los VAEs en un contexto específico. Para ello, se explican los fundamentos del aprendizaje automático y las redes neuronales, de una manera clara y comprensible, sin, en un primer acercamiento, profundizar en las complejidades matemáticas. Posteriormente, se usará el conocimiento adquirido para explorar modelos generativos, siendo el Variational Autoencoder (VAE) el modelo elegido para este estudio. La aplicación que daremos a dicho VAE será la generación de imágenes, en este caso, generar texturas para un videojuego.

El videojuego seleccionado en este caso es Minecraft, un popular juego de construcción y exploración, conocido por su estilo visual de bloques y skins. A través de la implementación de un VAE, se buscará generar nuevas texturas de bloques y skins para enriquecer la experiencia visual de los jugadores.

Los objetivos específicos de este trabajo son los siguientes:

- Comprender los fundamentos teóricos del aprendizaje automático y las redes neuronales, así como los de los autoencoders y los VAE en la generación de imágenes.
- Implementar un VAE específico para la generación de texturas en el contexto de Minecraft, considerando tanto los bloques como los skins del juego.
- Evaluar la calidad y la coherencia de las texturas generadas por el VAE, comparándolas con las texturas originales del juego y considerando su integración directa en el entorno de Minecraft.
- Explorar posibles trabajos futuros, como probar el modelo en otros videojuegos con texturas más realistas, comparar diferentes modelos generativos como los GANs y los Diffusion Models, y ampliar el tamaño y la diversidad de las texturas generadas.

A través de esto, se espera abrir la puerta a gente como yo, que desea introducirse en el mundo de la inteligencia artificial, proporcionando una manera clara y visual del funcionamiento de las redes neuronales y modelos generativos. Con nuestra aplicación hecha con un VAE se espera contribuir al campo de la visión artificial, explorando el potencial de estas técnicas en la generación de texturas en videojuegos.

Capítulo 2
Aprendizaje Automático

El aprendizaje automático o machine learning es una rama de la inteligencia artificial que crea sistemas que son capaces de aprender por sí solos a partir de un conjunto de datos. La idea es usar datos y algoritmos para imitar la forma en la que aprenden los seres humanos, y por consiguiente, realizar predicciones o tomar decisiones sin ser programadas explícitamente para hacerlo. Las bases del aprendizaje automático residen en la estadística, que es la ciencia que nos dota de la capacidad de extraer información de grandes cantidades de datos y usarla para diversos fines. Métodos como la regresión lineal y la estadística bayesiana siguen siendo fundamentales hoy en día en el aprendizaje automático.

El proceso de aprendizaje automático podría dividirse en varias fases:

- Selección de los datos (conjunto de datos) para nuestra tarea específica
- Elección de un algoritmo de aprendizaje.
- Creación del modelo.
- Entrenamiento de dicho modelo con datos de entrenamiento.
- Comprobar eficacia del aprendizaje con datos test.
- Posterior mejora del modelo.

2.1. El aprendizaje supervisado

Como su propio nombre dice, en este aprendizaje el modelo necesita de supervisión, se entrena con datos etiquetados, es decir, con la respuesta correcta o el resultado deseado para cada ejemplo. Así, el algoritmo de aprendizaje supervisado intenta modelar las relaciones existentes entre los datos de entrada y los resultados esperados, encontrando

una función que le permita predecir o clasificar nuevos datos basándose en lo que han aprendido previamente [11]. Existen dos tipos principales de aprendizaje supervisado:

- Clasificación: El objetivo es clasificar la pertenencia de un dato de entrada a una clase. Un ejemplo de un modelo entrenado con este tipo de aprendizaje sería un detector de señales de tráfico y decir de qué tipo es. En otros casos más simples, la clasificación solo consiste en decir si la entrada pertenece a un tipo o no [11].
- Estimación del Valor: El objetivo ahora es hacer una predicción de un valor dada una entrada. El modelo se ha entrenado dando valores n-valores y proporcionado la salida esperada Y. Una vez entrenado se busca pasar nuevos n-valores y que el modelo haga la predicción del valor Y Por ejemplo, a partir de unas variables de entrada(Zona, m^2, habitaciones, etc) predecir el valor de una casa [11].

2.2. El aprendizaje no supervisado

En este tipo el modelo carece de elementos etiquetados, no existe un valor objetivo, por lo tanto, el modelo trata de encontrar patrones, similitudes o relaciones entre los datos de entrada. Se pueden destacar los siguientes tipos:

- Clustering: El objetivo es identificar patrones y agrupar los datos de entrada en conjuntos o clusters similares sin tener etiquetas previas. Un ejemplo sería clasificar noticias o artículos en función de su contenido, juntando aquellas que tratan sobre el mismo tema, terminando por diferenciar aquellas que hablen de deportes, política, etc. [11].
- Aprendizaje por reglas de asociación: El objetivo es encontrar relaciones y patrones frecuentes en los datos de entrada. Por ejemplo, según lo que un cliente suele comprar cada vez que acude al supermercado, podemos establecer patrones de consumo y ubicar productos que se asocian en estanterías cercanas [11].
- Generativos: El objetivo es aprender a generar nuevos datos a partir de una gran cantidad de datos de entrada. Algunos ejemplos son los Variational Autoencoders (VAES) que veremos más adelante.

Y un ejemplo sería un modelo que extraiga las características más importantes de un conjunto de datos grande y reduzca la dimensionalidad de los datos. Este tipo de aprendizaje es el que usan los autoencoders que veremos más adelante.

2.3. El aprendizaje por refuerzo

Este tipo de aprendizaje se suele usar en modelos que interactúan con un entorno en el que hay que tener diversas variables en cuenta y ya no basta con etiquetar cada una ellas. Este modelo se entrenará con un sistema de recompensas en las que cada acción será recompensada o penalizada. De esta manera, el sistema aprenderá la mejor forma de realizar una tarea específica, maximizando la recompensa y minimizando el riesgo [11]. Se dispone entonces de un objetivo al que llegar, un conjunto de acciones posibles y retroalimentación sobre su desempeño. El ejemplo más común sería la IA de los videojuegos, tanto en los actuales como en el clásico ajedrez.

Capítulo 3
Redes neuronales

Las redes neuronales son un tipo de modelo de aprendizaje automático, que dependiendo de su finalidad pueden ser de diferentes tipos de aprendizaje automático. Su estructura y funcionamiento se basa en las redes neuronales biológicas del cerebro humano Figura3.1. El sistema nervioso humano está compuesto por células especializadas llamadas neuronas, las cuales se interconectan entre sí a través de estructuras llamadas axones y dendritas. La región que conecta dos neuronas es a lo que llamaríamos sinapsis y permiten la transmisión de señales eléctricas y químicas entre las neuronas.
Las redes neuronales artificiales tratan de simular el funcionamiento de las redes neuronales biológicas de manera computacional. Por tanto, una red neuronal artificial está formada por neuronas artificiales conectadas entre sí, por lo que llamaríamos pesos. Véase la relación en la Figura 3.2

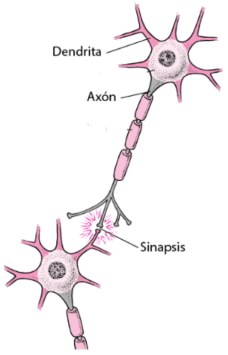


Figura 3.1: Neurona biológica [32]

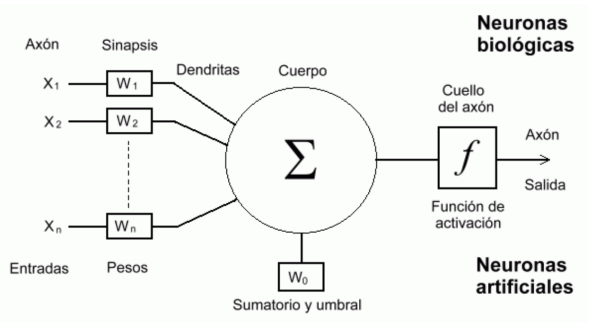


Figura 3.2: Relación neurona [3]

Una neurona artificial no deja de ser una función en la que se produce una combinación lineal entre las entradas que llegan y unos pesos que determinan la "importancia" de cada entrada en la operación [14].
Podemos distinguir diferentes tipos de redes neuronales en función de su número de capas, como se distribuyen y cómo se usan dichas capas. Distinguiendo las redes monocapa, multicapas, recurrentes y convolucionales como los tipos principales.

3.1. Redes neuronales monocapa

El tipo de red neuronal más simple es la red neuronal monocapa, que contiene una única capa de entrada y un nodo de salida. Su estructura sería la siguiente:

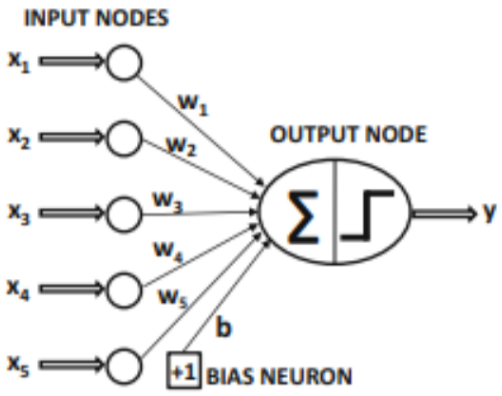


Figura 3.3: Red neuronal monocapa[3]

Donde $X = x_1, x_2, ..., x_n$ son las entradas de los datos, $W = w_1, w_2, ..., w_n$ son los pesos sinápticos que multiplicarán a las entradas para determinar la prioridad de cada una de ellas, y b es el sesgo o bias que se puede considerar como otro peso más que controla la predisposición de la neurona a generar una salida 0 o 1 independientemente de los pesos.

Todos estos datos se procesan en el nodo de salida primero aplicando un sumatorio $\sum_{i=1}^{n} x_i \cdot w_i + b = x_1 \cdot w_1 + x_2 \cdot w_2 + ... + x_n \cdot w_n + b$, que será usado por la función de activación para darnos una salida. La función de activación es una función que modifica el valor de salida del nodo. Una de sus características más importantes es su capacidad de agregar no linealidad a la red neuronal, lo cual es muy importante para las redes neuronales multicapa [3]. La elección de la función de activación dependerá del objetivo de nuestro modelo. Por ejemplo, si nuestro modelo realiza predicciones de probabilidad, podríamos usar la función sigmoide porque su rango está entre 0 y 1, y la probabilidad de cualquier suceso solo se encuentra entre 0 y 1. Véase algunas de las funciones de activación más comunes en la Figura3.4 :

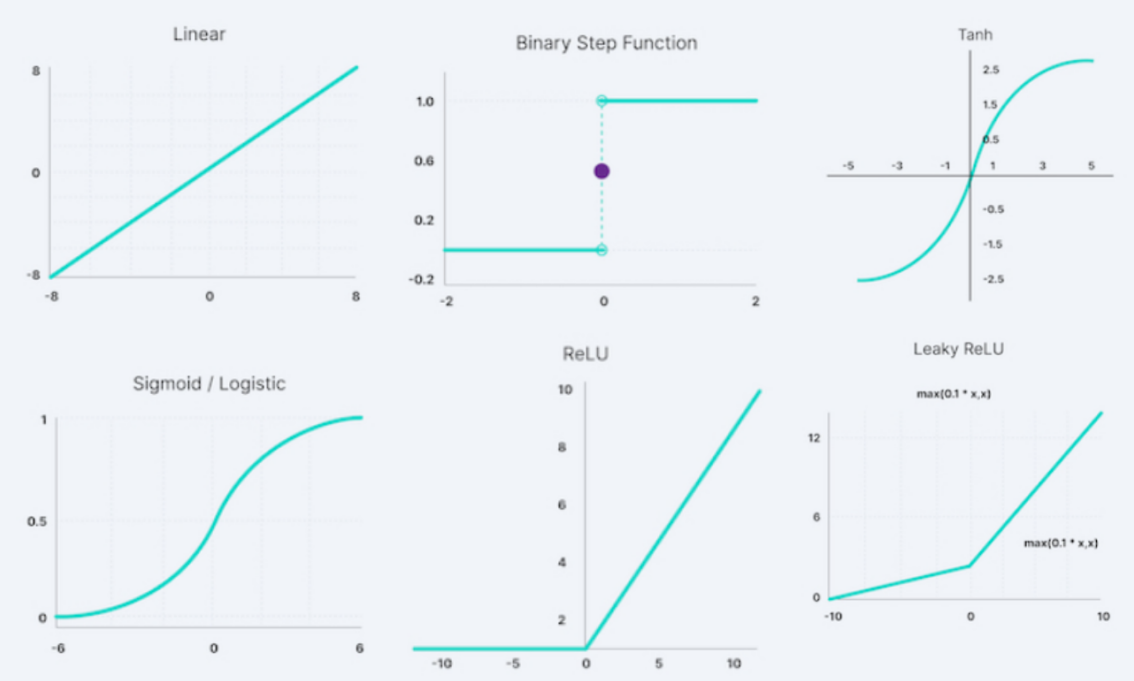


Figura 3.4: Funciones de activación [5]

Las redes neuronales monocapa nos pueden servir para resolver problemas simples, como, por ejemplo, predecir el precio de una casa en función de sus características (regresión lineal simple). Para representar problemas reales más complejos son necesarias redes neuronales multicapas.

3.2. Redes neuronales multicapa

A diferencia de las redes neuronales monocapa, que constan de una única capa de entrada que transmite directamente los datos al nodo de salida, las redes neuronales multicapa cuentan con una o más capas ocultas intermedias (hidden layers). Cada capa oculta está compuesta por un conjunto de neuronas que se conectan con las neuronas de la capa anterior y la capa siguiente. Véase la estructura en la [Figura3.5]

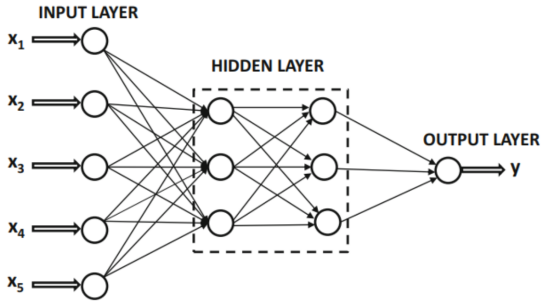


Figura 3.5: Red neuronal Multicapa[3]

Cada neurona en una capa oculta toma como entrada una combinación lineal de las salidas de las neuronas de la capa anterior, y aplica una función de activación no lineal a la suma ponderada para producir su salida. Aquí cobra importancia la capacidad de la función de activación de añadir no linealidad a las redes neuronales, puesto que si usáramos una función de activación lineal o no usáramos función de activación, la red neuronal multicapa por muchas capas que tuviera acabaría dándonos una salida lineal siendo igual que una red monocapa.

Existen redes neuronales multicapa con múltiples salidas como puede ser un autoencoder el cual veremos en profundidad más adelante.

Las redes neuronales multicapa son redes feed-forward (propagación hacia adelante), las neuronas están organizadas en capas que se conectan de manera unidireccional, una vez se llega a una capa solo tenemos acceso a la información ha llegado a esa capa y no a las anteriores. Sin embargo, existen otro tipo de redes, **las redes neuronales recurrentes (o RNN)**, que permiten crear ciclos y que la información fluya entre las neuronas en ambas direcciones.

3.3. Redes neuronales Recurrentes

Las redes neuronales clásicas presentan dificultades a la hora de procesar datos secuenciales que de manera independiente carecen de sentido, como una secuencia de palabra(texto) o de imágenes (vídeo). Las redes neuronales recurrentes resuelven este problema, utilizan para ello el concepto de recurrencia, que se refiere a la capacidad de mantener y utilizar información anterior y posterior en el procesamiento [9]. De esta manera, el flujo de información de neuronas en las RNN es mucho más complejo, ya que una sola neurona puede estar conectada con las neuronas de la siguiente capa, de la anterior y con ella misma y dependiendo del tipo de algoritmo de entrenamiento que se elija puede llegar a predecir eventos. La diferencia más importante con las redes clásicas es que las RNN no solo tienen en cuenta la entrada actual, sino también entradas pasadas o futuras, en distintos instantes de tiempo [3].

En la Figura 3.6 se puede observan las estructura de una red neuronal recurrente en un instante de tiempo y en la Figura 3.7 se puede ver como quedan las capas en diferentes instantes de tiempo para una secuencia de palabras.

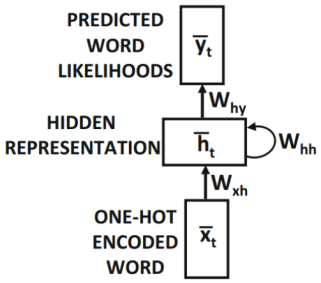


Figura 3.6: Red neuronal Recurrente[3]

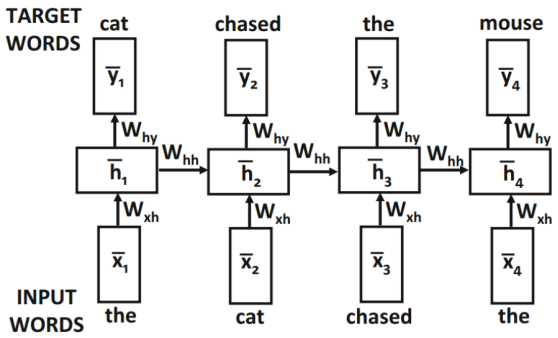


Figura 3.7: Representación de las capas en el tiempo[3]

Existen diferentes tipos de arquitecturas de RNN según el objetivo de las mismas:

- One to many: La entrada es un único dato y la salida una secuencia. Un ejemplo de uso es la generación de una descripción dada una foto como entrada
- Many to one: La entrada es una secuencia de datos y la salida un único dato. Un ejemplo sería dado un texto valoración sobre un producto, proporcionar como salida la puntuación dada a dicho producto en función del contenido de la entrada.
- Many to many: Tanto la entrada como la salida contienen secuencias de datos. Un claro ejemplo son los traductores, que generan a partir de un texto de entrada en un idioma otro texto de salida traducido.

3.4. Redes neuronales Convolucionales

Si bien las redes neuronales clásicas permiten procesar imágenes, presentan limitaciones a la hora de tener en cuenta la estructura espacial, presentando conflictos en problemas como puede ser la clasificación o detección de imágenes. Las redes convolucionales son las redes neuronales que solucionan dichos conflictos y son esenciales en el mundo de la visión por computador. A diferencia de una red clásica (una red neuronal multicapa) que también puede clasificar imágenes, la convolucional tiene en cuenta la estructura espacial de la imagen, dándonos un funcionamiento mucho más correcto y preciso.

Para explicar la utilidad de las redes convolucionales es útil este ejemplo: Si se quisiera clasificar letras escritas a mano como la de la Figura 3.8 con una red neuronal multicapa, le introduciría como entradas los valores de todos los píxeles de la imagen como si estos fueran independientes.

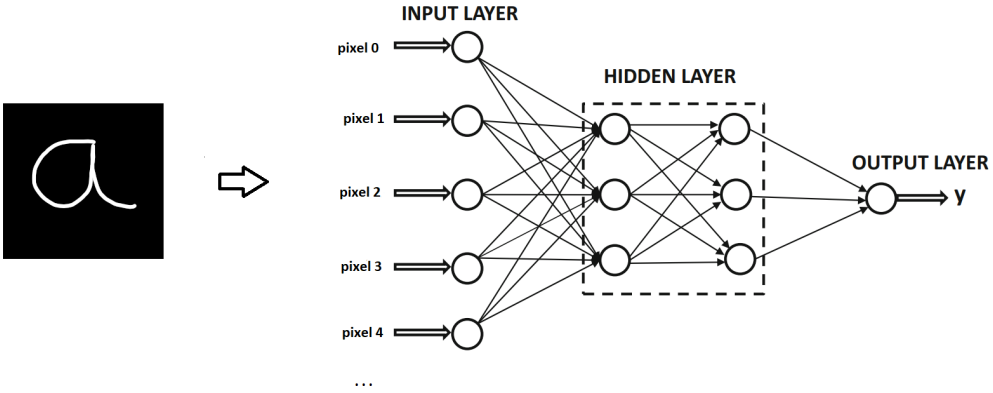


Figura 3.8: Red neuronal Multicapa

De tal manera que obtendremos buenos resultados solo cuando las letras que clasifiquemos sean muy parecidas a las imágenes con las que se ha entrenado (situadas en el mismo lugar, tamaño muy similar, etc). Sin embargo, si tenemos como entrada una letra *a* como esta:

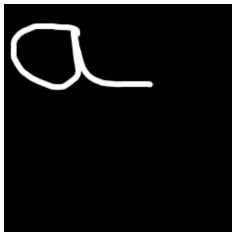


Figura 3.9: Imagen demostración

No se calificaría correctamente, puesto que iría píxel por píxel comparando y no encontrando coincidencias en los vectores, dando un resultado erróneo. Y es a raíz de estos problemas de los que surge la necesidad de nuevas redes que tengan en cuenta la dependencia de los píxeles y que sean capaces de extraer características de las imágenes para encontrar las mismas en imágenes diferentes.

Las redes convolucionales están inspiradas biológicamente en el córtex visual, que tiene pequeñas regiones de neuronas que son sensibles a regiones específicas del campo visual. Las bases de las redes convolucionales fueron motivadas por el entendimiento del córtex visual de un gato en un experimento realizado por los científicos Hubel y Wiesel donde mostraban cómo algunas neuronas se activaban solo en presencia de ciertos patrones. Descubrieron que estas pequeñas neuronas que reaccionan a regiones o formas específicas

trabajaban de manera conjunta formando una arquitectura que es la que nos permite tener percepción visual. Esta idea de pequeñas partes especializadas en tareas concretas dentro de un sistema es la base de las redes neuronales convolucionales [16].

Una red neuronal convolucional es un tipo de red multicapa que presenta diversas capas convolucionales en las que se realiza la extracción de características y otras de reducción de muestreo (pooling) alternadas entre sí. Por último, contiene una capa fully-connected que se parece a la capa de salida en una red neuronal feedforward que toma las características extraídas y las utiliza para clasificar la entrada, que suele ser generalmente una imagen [16].

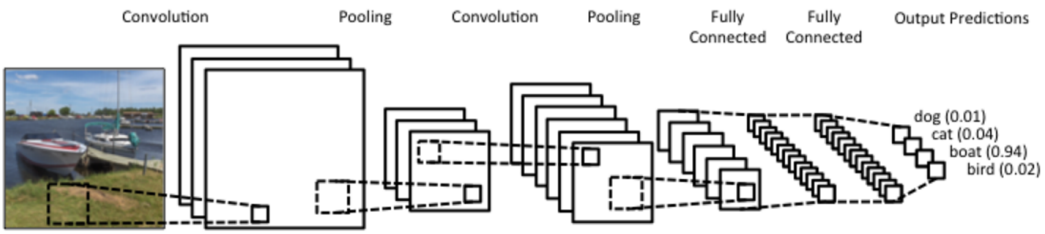


Figura 3.10: Red convolucional[20]

3.4.1. Capa de Entrada

La primera capa de la red neuronal convolucional es la entrada, que generalmente es una imagen.

3.4.2. Capa convolucional

Como su propio nombre indica, es donde se realiza una convolución a la imagen de entrada. Una convolución es una operación lineal que permite detectar patrones en la imagen de entrada. La convolución consistirá en elegir un filtro (o kernel) que iremos pasando por la imagen de entrada, realizando multiplicaciones para obtener un mapa de características. El filtro del tamaño elegido se superpone sobre una sección de la imagen de entrada y se calcula la convolución. Una vez hecho esto, almacenamos el resultado en una matriz y desplazamos el filtro una posición, realizando posteriormente el mismo procedimiento [17].

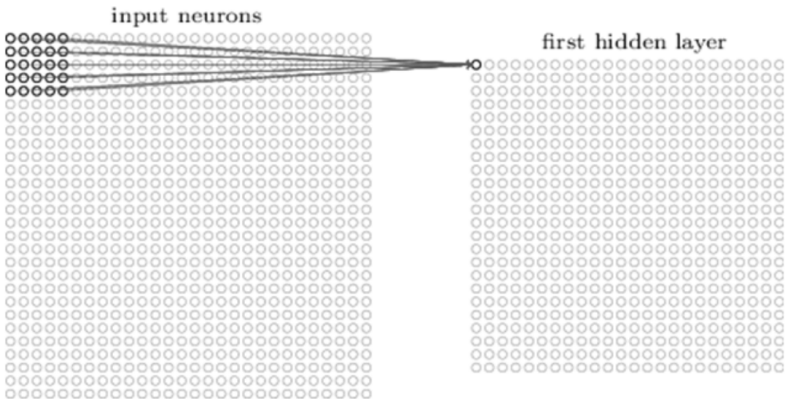


Figura 3.11: Proceso de convolución [17]

Se repite este proceso hasta que se ha recorrido completamente la imagen, y la matriz obtenida es lo que conocemos como mapa de características. Este mapa contiene las características relevantes extraídas por el filtro.

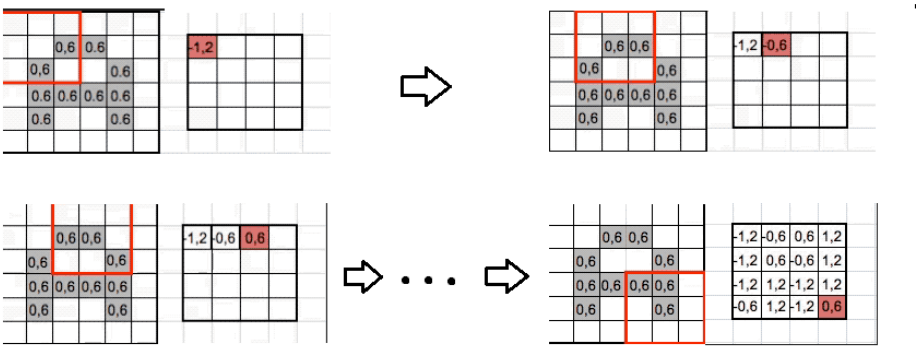


Figura 3.12: obtención mapa características[6]

Generalmente, los valores del filtro se iniciarán con valores aleatorios y será la propia red la encargada de ir ajustando dichos valores para lograr su propósito. Por último, aplicamos una función de activación, generalmente ReLu ($f(x) = \text{máx}(0, x)$), a todos los mapas obtenidos.

Es en las capas de convolución dónde se observa la analogía con las pequeñas neuronas especializadas del córtex visual, ya que cada elemento del mapa de características obtenido hace referencia a una región de la imagen original.

3.4.3. Capa de pooling

En esta capa se reduce el tamaño de los mapas obtenidos en capas anteriores, quedando las características más importantes con el objetivo de disminuir la carga computacional del sistema [17]. Así como las capas convolucionales, se aplican en una región o ventana específica. Los dos tipos más usados de pooling son:

- Max Pooling: Toma el máximo elemento dentro de la ventana de aplicación. Es el más usado.
- Average Pooling: Toma el promedio de los elementos dentro de la ventana de aplicación.

Todo este proceso de capas se lleva a cabo de forma iterativa, donde los mapas obtenidos en las capas iniciales se convierten en las entradas de las capas posteriores. Este proceso continúa hasta llegar a las capas fully-connected, que utiliza las características extraídas para realizar la clasificación. Adoptando la red neuronal en conjunto, una estructura en forma de embudo.

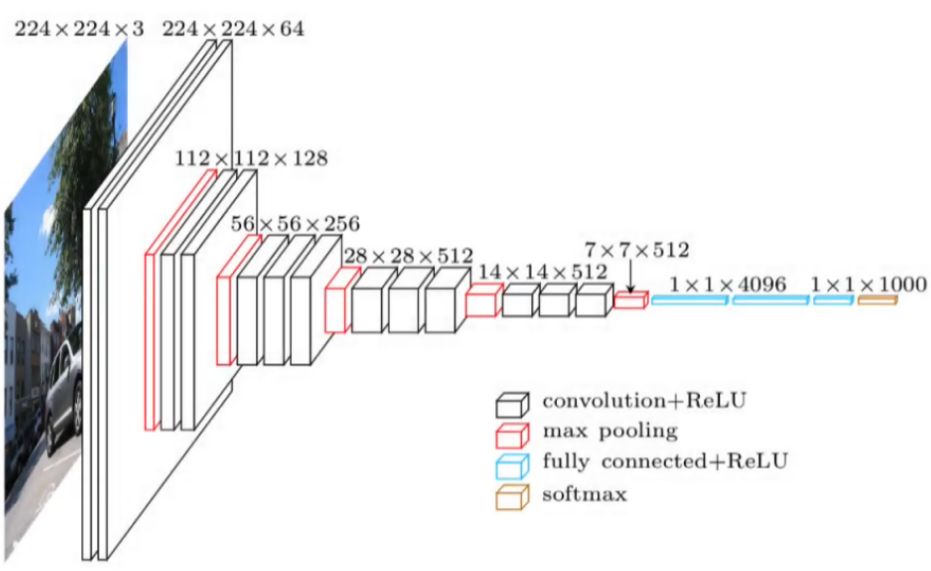


Figura 3.13: Estructura red convolucional[12]

3.5. Entrenamiento Redes Neuronales

Una Red neuronal está formada por diversas capas de neuronas conectadas entre sí. La importancia o prioridad de cada conexión entre neuronas viene determinada por los pesos. Por tanto, el valor de cada neurona en la que convergen diferentes entradas se

puede definir como una función cuyo valor se calcula mediante la combinación lineal de pesos y entradas: $f(x_1, x_2, \ldots, x_n) = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n + b$

Además, se aplica una función no lineal para obtener la salida (función de activación), evitando la linealidad. La elección de esta viene determinada por el objetivo de nuestra red. Una de las más usadas es la $ReLU$: $ReLU(x) = max(x, 0)$ [7].

Estos pesos son los valores que se ajustan durante el entrenamiento para conseguir obtener los resultados esperados en nuestra red neuronal. Para entrenar una red neuronal, se comienza inicializando de manera aleatoria los pesos, lo cual nos dará malos resultados al principio. Sin embargo, se busca ir ajustando estos valores iniciales midiendo el error entre la salida obtenida y la salida deseada con una función de error o coste [7]. El objetivo es minimizar el error que hay con los valores escogidos para ir mejorando progresivamente los resultados, pero, ¿cómo ajustamos los valores de los pesos?

Conseguimos realizar dichos ajustes con el algoritmo de backpropagation (propagación hacia atrás), que calculan los gradientes de los pesos, propagando de manera recursiva el error desde la capa de salida a la de entrada. Estos valores serán utilizados por un algoritmo de optimización, el descenso del gradiente, que será el encargado de ajustar los pesos de cada neurona [15]. Los gradientes nos indican qué dirección tomar a la hora de ajustar nuestros pesos, pero no cuánto debemos ajustarlos [13]. El encargado de esto es la tasa de aprendizaje (learning rate) que determinará en qué medida se ajustan los pesos y cuyo valor tiene gran importancia a la hora de un correcto funcionamiento de la red neuronal que podría dar problemas de sobreajuste.

Por tanto, un el entrenamiento de una red neuronal se podría resumir en los siguientes pasos:

1. Inicialización aleatoria de los pesos de la red.
2. Se introduce la entrada de datos y se avanza por la red neuronal propagándose hasta conseguir una salida.
3. Cálculo de la función de error.
4. Backpropagation para saber cómo ajustar los pesos.
5. Ajustar los pesos con el descenso del gradiente.

6. Repetir el proceso hasta que consigamos un error mínimo y la red tenga el funcionamiento esperado.

Capítulo 4
Modelos Discriminativos vs Modelos Generativos

En Machine Learning, existen varias formas de categorizar un modelo de aprendizaje automático: modelos discriminativos, modelos generativos, modelos paramétricos, modelos no paramétricos, modelos basados en árboles, etc. Dichos modelos pueden ser implementados utilizando redes neuronales. En este trabajo nos centraremos en los modelos discriminativos y los modelos generativos debido a su importancia en la Visión Artificial. Ambos enfoques tienen diferentes objetivos y métodos de modelización, y cada uno tiene sus propias ventajas y desventajas.

Los modelos discriminativos son aquellos que se utilizan para modelar la relación directa entre las variables de entrada (características) y la variable de salida (clase o etiqueta). Su objetivo es encontrar una función de decisión que permita clasificar nuevos datos de entrada en una o varias categorías. Los modelos discriminativos son útiles cuando se desea hacer predicciones precisas y rápidas sobre nuevas instancias de datos. Estos modelos separan las clases en el conjunto de datos mediante el uso de probabilidad condicional ($p(y|x)$). Ejemplos de modelos discriminativos incluyen las redes neuronales, los árboles de decisión y los modelos de regresión logística. Estos modelos se centran en establecer relaciones directas entre las variables de entrada y la variable de salida, sin preocuparse tanto por capturar la distribución conjunta de los datos

Por otro lado, los modelos generativos se utilizan para modelar la distribución de probabilidad conjunta de las variables de entrada y salida. Es decir, su objetivo es aprender cómo se generan los datos de entrada y cómo se relacionan con la variable de salida. Una vez capturada la distribución de probabilidad de los datos de entrada, podemos generar nuevos datos de entrada que sean similares a los datos de entrenamiento (generando nuevas imágenes, por ejemplo), modelar la distribución subyacente de los datos, etc. Ejemplos de modelos generativos incluyen los modelos LDA, los modelos ocultos de Markov y las redes

neuronales generativas adversarias (GAN). Estos modelos se enfocan en aprender cómo se generan los datos y cómo se relacionan con la variable de salida, lo que los hace útiles en tareas de síntesis de datos, detección de anomalías y modelización de distribuciones de datos.

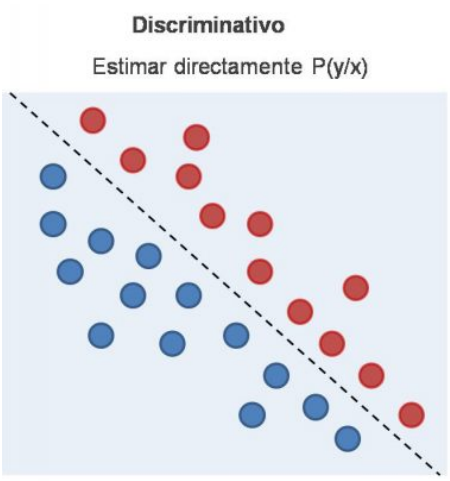
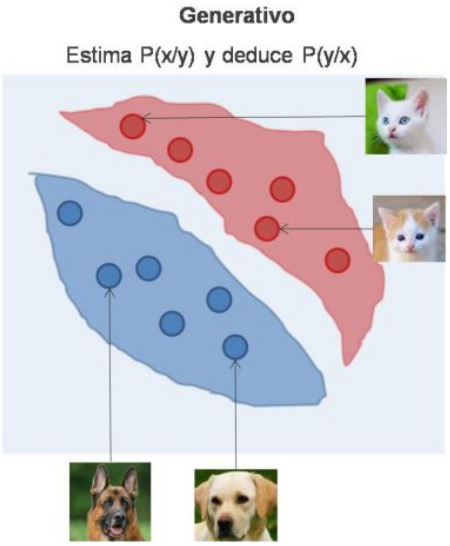
	Discriminativo	**Generativo**
Objetivo	Modelan el límite de decisión para clasificar conjuntos de datos, estableciendo la relación directa entre los datos de entrada y de salida.	Obtener la distribución de probabilidad conjunta de las variables de entrada y salida para usarla en diferentes tareas como generar nuevas imágenes, detectar anomalías, etc.
Aprendizaje	Útiles para tareas de aprendizaje automático supervisado. Aprenden la probabilidad condicional - $p(y\|x)$.	Útiles para tareas de aprendizaje automático no supervisado. Aprenden la distribución de probabilidad conjunta - $p(x, y)$.
Ilustración		
Ejemplos	Árboles de decisión, Regresión logística, etc.	Modelos LDA, redes bayesianas, GANs, etc.

Tabla 4.1: Tabla comparativa Discriminativos vs Generativos

Los modelos generativos tienen una gran utilidad si son aplicados a imágenes, ya que permiten generar nuevas imágenes del mismo tipo que las imágenes con las que ha sido entrenado el modelo. Por ejemplo, podríamos entrenar un modelo con rostros humanos para que sea capaz de generar nuevos rostros de personas que no existen. Este gran potencial de crear nuevas imágenes es muy usado en visión artificial. Entre los modelos generativos más utilizados en visión artificial se encuentran los autoencoders

variacionales (VAE), las redes generativas adversarias (GAN) y los modelos de difusión (diffusion models).

Capítulo 5
Variational Autoencoder

Los VAE (Variational autoencoder) son modelos generativos que son un tipo de de autoencoder, y para entender su funcionamiento debemos entender primero qué es un autoencoder.

5.1. Autoencoder

Un autoencoder es un tipo de red neuronal artificial entrenada de manera no supervisada, cuyo objetivo es generar, a partir de una entrada de datos, una salida estrechamente relacionada. Se codifica una entrada de manera que se obtenga una versión comprimida de la misma para después decodificarla buscando que la reconstrucción sea lo más parecida a la imagen original. Para ello, la red neuronal se entrena con una gran cantidad de datos de los que aprende a extraer las características más importantes de estos datos (extracción de características) consiguiendo una representación de los datos de entrada más compacta, reduciendo así su dimensionalidad. Una vez tenemos la entrada codificada, el siguiente paso es que la red neuronal aprenda a reconstruir, a partir del dato de entrada codificado, el mismo dato de entrada original.

El autoencoder, por tanto, se compone de 3 partes fundamentales:

- Encoder: Parte de la red neuronal donde la entrada comienza a reducir su dimensionalidad progresivamente extrayendo sus características más importantes.
- Espacio latente: es la representación comprimida de la entrada y que, por tanto, contiene menor cantidad de datos.
- Decoder: Parte de la red neuronal que reconstruye la entrada a partir de los datos comprimidos del espacio latente.

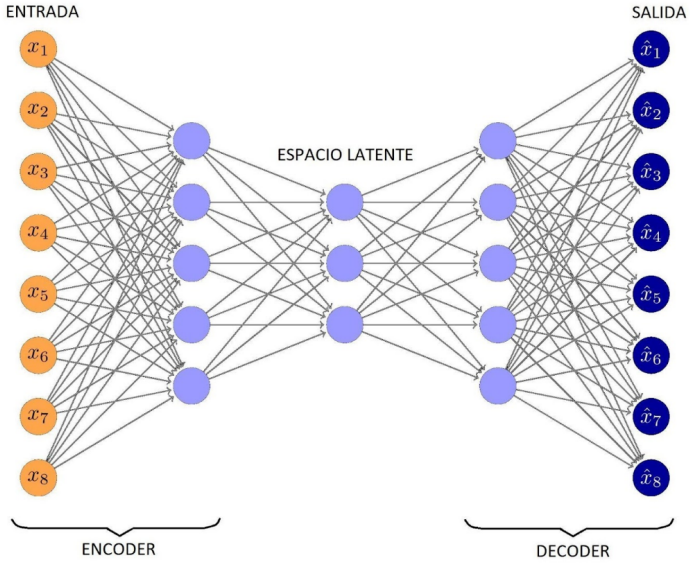


Figura 5.1: Estructura Autoencoder[30]

Para lograr su funcionamiento, el Autoencoder se entrena ajustando cada uno de los pesos de las entradas de todas las neuronas y la función error se calcula como la diferencia entre el dato original (entrada) y el dato reconstruido (salida) buscando minimizar dicha función como en el resto de arquitecturas de redes neuronales artificiales. Existen diferentes tipos de autoencoders (Denoising autoencoders, Sparse autoencoder, Variational autoencoder, etc) que pueden tener diferentes aplicaciones como: compresión de información, detección de fraudes, eliminación de ruido de imágenes o generación de imágenes. El tipo de autoencoder en el que nos vamos a centrar es el Autoencoder Variacional(VAE).

5.2. VAE

El Variational Autoencoder (VAE) es un modelo que se basa en la arquitectura de un autoencoder convencional, pero se diferencia en que tiene un espacio latente distinto que le permite generar nuevas muestras, convirtiéndolo en un modelo generativo. En los autoencoders tradicionales, la entrada se comprime para obtener un vector latente simplificado que se utiliza posteriormente para la reconstrucción a través del decoder. En los VAEs se realiza un proceso similar, pero ahora el objetivo es codificar la entrada no en un vector sino en un espacio de probabilidad. Asumimos que los datos de entrada se generan a partir de una distribución de probabilidad existente previa e intentamos que el modelo aprenda a derivar datos de esa distribución de probabilidad. Dicha distribución de probabilidad es a partir de la cual se obtiene el vector que se pasa al decoder [24].

Para lograr esto se calculan dos vectores, el vector media $\mu(x)$ y la desviación típica $\sigma(x)$, reemplazando la última capa de del encoder por dos capas, dando cada una de ella como salida $\mu(x)$ y $\sigma(x)$ respectivamente. Además de minimizar el error en las reconstrucciones cómo veníamos haciendo, también actualizamos la función de error para penalizar la divergencia KL. Esto nos permite controlar que los vectores latentes se encuentren en una ubicación más centralizada y uniforme, lo que a su vez nos permite decodificarlos de manera más eficiente [24, 35].

Para generar ahora un vector para el decoder bastaría con muestrear de la distribución definida por el encoder. Sin embargo, este proceso genera un cuello de botella que nos impide usar backpropagation [35]. Para arreglar este problema, se aplica lo que conocido vulgarmente como 'El truco de la reparametrización que consiste en muestrear de otra distribución, generalmente la normal, y usar $\mu(x)$ y $\sigma(x)$ para transformar la muestra.

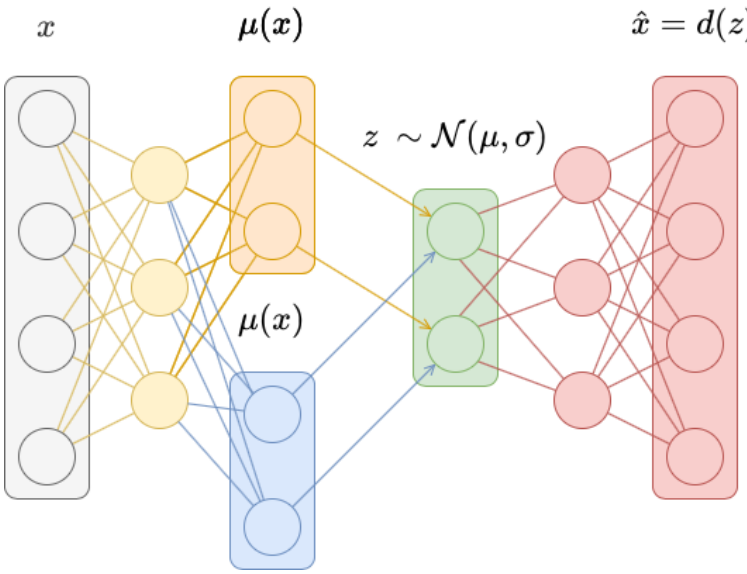


Figura 5.2: Estructura Autoencoder Variacional[10]

Capítulo 6
Tecnología y creación del código

En este capítulo, nos sumergiremos en el aspecto práctico de nuestro trabajo, luego de haber establecido el marco teórico en capítulos anteriores. Nos enfocaremos en las tecnologías necesarias para la implementación exitosa de nuestro VAE y exploraremos en detalle el proceso de desarrollo del VAE en código.

6.1. Tecnología

El entorno de desarrollo elegido ha sido PyCharm para escribir, editar y ejecutar el código, siendo Python el lenguaje escogido. Se hace uso de las siguientes librerías:

- Pytorch: Es la biblioteca principal del proyecto usada para implementar y entrenar redes neuronales. PyTorch es una biblioteca de aprendizaje automático de código abierto que proporciona una forma flexible y eficiente de construir y entrenar modelos de aprendizaje profundo.
- Torchvision:es una biblioteca de Python para tareas de visión por computador, compatible con PyTorch. Proporciona herramientas para cargar y procesar conjuntos de datos de imágenes, transformaciones de imágenes y modelos pre-entrenados para tareas de clasificación y análisis de imágenes.

Para el entrenamiento de un modelo generador de imágenes es importante contar con una buena tarjeta gráfica. El equipo utilizado para entrenar los modelos de este trabajo cuenta con las siguientes especificaciones:

- Tarjeta gráfica: Gigabyte GeForce RTX 3070 Ti GAMING OC 8GB GDDR6X.
- Procesador: Intel Core i7-12700 4.9 GHz.
- Fuente de alimentación: Corsair RMx Series RM750x 750W 80 Plus Gold Modular.
- Tarjeta RAM: Team Group T-Force Delta RGB DDR4 3600MHz PC4-28800 16GB 2x8GB CL18 Negro.

- Unidad de almacenamiento: Samsung 970 EVO Plus 1TB SSD NVMe M.2.
- Placa Base: MSI PRO B660M-A WIFI DDR4.

6.2. Proceso

Antes de entrar a la aplicación, mi objetivo ha sido comprender el funcionamiento de un autoencoder en código para la posterior implementación del VAE. Para ello he usado como referencia un código de un autoencoder básico [2].

Tras hacer pruebas y comprender cómo funcionan las redes en pytorch y el autoencoder, he procedido a la ampliación del código para transformarlo en VAE. Para ello se han llevado a cabo las siguientes acciones:

- Cambiar el espacio latente: En los autoencoder tradicionales, el espacio latente es un vector que representa una versión codificada del dato de entrada. En un VAE, se transforma en una distribución de probabilidad a partir de la cual se puede muestrear posteriormente. Se añaden dos capas llamadas 'mu' (media) y 'log var' (logaritmo de varianzas), que son parámetros estadísticos utilizados para modelar la distribución latente de los datos.
- Separamos los métodos encode y decode: En el método encode, se pasa a través de las capas del codificador y se obtienen las capas 'mu' y 'log var'. En el método decode, se atraviesan las capas del decodificador para generar las muestras a partir de la distribución latente.
- Añadimos el método reparameterize: En lugar de muestrear directamente de la distribución latente, aplicamos el "truco"de la parametrización. Este consiste en tomar una muestra aleatoria de una distribución conocida, como la distribución normal estándar, y luego aplicar transformaciones a esa muestra para obtener una muestra de la distribución latente. La transformación se realiza mediante la fórmula: $z = \mu + \sigma \cdot \epsilon$, donde μ es la media, σ es la desviación estándar, ϵ es una muestra de la distribución normal y z el vector que pasaremos al decoder para la reconstrucción. En el código, se realizan operaciones con el logaritmo de las varianzas para obtener la desviación estándar, se toma una muestra aleatoria de la distribución normal y se realiza la operación. El objetivo de la parametrización es permitir el uso del algoritmo de retropropagación (backpropagation) para el entrenamiento del modelo.

- El método forward ahora llama a los métodos mencionados anteriormente para atravesar toda la red neuronal y generar las muestras.
- Se modifica la función de pérdida añadiendo la divergencia de Kullback-Leibler (KLD) para regularizar la distribución latente del modelo.

6.3. Pruebas iniciales

Antes de comenzar con la aplicación, se procedió a realizar una serie de pruebas para ajustar los valores de nuestro VAE, así como para comprender su funcionamiento. Comencé realizando pruebas con el conjunto de datos [33], un conjunto de datos de gran tamaño y fácil acceso formado por imágenes 28x28 en blanco y negro de dígitos escritos a mano. Tras modificar el código del autoenconder para obtener el del VAE, las pruebas iniciales se realizan sin cambiar ningún parámetro ni realizar ajustes a la red.

Figura 6.1: Resultados obtenidos en la prueba inicial utilizando el conjunto de datos MNIST

Ante los malos resultados existen diferentes valores que podemos cambiar para ajustar la red a nuestro gusto.

6.3.1. Número de épocas

Las épocas determinan el número de veces que se itera sobre el conjunto de datos de entrenamiento, aumentar el número puede darnos mejores resultados, pero también podemos sobreajustar la red.

6.3.2. Tasa de aprendizaje

La Tasa de aprendizaje es un valor que determina el grado de ajuste que se aplica a los pesos durante el entrenamiento. Tener una tasa demasiado alta puede provocar ajustes demasiado grandes y dificultar el entrenamiento de la misma manera que una tasa demasiado baja puede ralentizar mucho el entrenamiento.

6.3.3. Función de pérdida

La función de pérdida desempeña un papel importante en el funcionamiento del VAE, ya que mide el error entre los datos originales y los de las reconstrucciones además de regular la distribución latente del modelo con KLD. Algunas de las más comunes son el error cuadrático medio (MSE) o la pérdida de la entropía cruzada binaria (BCE).

6.3.4. Capas de la red

En la red se pueden cambiar el número de capas así como ajustar los tamaños de los datos al pasar a través de cada una de ella. Incrementar estos valores puede mejorar el desempeño, sin embargo, es importante tener precaución, ya se puede sobreajustar la red y obtener resultados aún peores.

6.3.5. Arquitecturas más complejas

En lugar de usar capas fast-forward exclusivamente, se pueden implementar capas convolucionales, transformando el VAE en una red neuronal convolucional.

Tras conocer los valores que podemos ajustar, procedemos a experimentar con ellos: Aumentamos el número de épocas de 20 a 40 y observamos que los cambios son mínimos.

Figura 6.2: Resultados obtenidos al aumentar el número de épocas

Probamos a cambiar la tasa de aprendizaje a 0.001 y obtenemos resultados un poco peores, por lo que decidimos dejarla con la tasa actual(1e-3).

Figura 6.3: Cambiar tasa de aprendizaje

Cambiamos la función de pérdida de la BCE a la MSE y obtenemos mejores resultados, se ven los números más claros.

Figura 6.4: Cambiar función de pérdida

Por último decidimos cambiar el número de capas, añadiendo una capa extra tanto al encoder como al decoder:

Figura 6.5: Cambiar número de capas

Los resultados con el conjunto de datos MNIST ya son considerablemente buenos y hemos aprendido cómo varían los resultados en función de los diversos ajustes que se pueden aplicar. El conjunto de datos MNIST está formado por imágenes en blanco y negro, por lo que solo usa un canal de color. Sin embargo, las texturas que se planean usar son en color, por lo que antes de pasar a las pruebas para la generación texturas es conveniente hacer alguna prueba con un conjunto de datos en color que use 3 canales, para así tener el código adaptado. Aparte del MNIST, un conjunto de datos muy conocido es el CIFAR10, un conjunto de 60000 imágenes de 32x32 en color que presenta 10 tipos de imágenes (automóviles, pajaros, caballos, etc.).

Adaptamos el código cambiando los canales de los colores y ajustando los tamaños de las imágenes procesadas. Procedemos a realizar la primera prueba con CIFAR10:

Figura 6.6: Prueba inicial CIFAR10

Obtenemos unos resultados con contrastes inusuales. A pesar de ello, se puede distinguir alguna imágen como puede ser el caballo en la primera prueba. Para evitar el contraste anterior, antes de mostrar las imágenes hacemos un np.clip(x, 0,1) para que los valores menores que 0 sean 0 y los que se pasen de 1 sean 1.La función np.clip(x, 0, 1) se utiliza para limitar los valores de una matriz o vector x dentro de un rango específico. En este caso, el rango establecido es de 0 a 1.

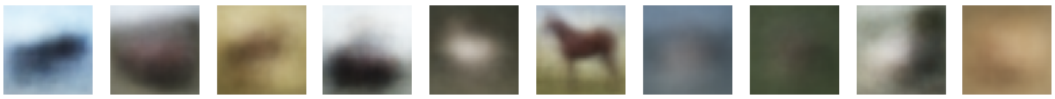


Figura 6.7: Prueba final CIFAR10

Los resultados han mejorado considerablemente, aunque no es posible diferenciar el tipo de imágen que estamos viendo, salvo excepciones. Sin embargo, en lugar de comenzar a adaptar la red neuronal para conseguir buenos resultados con CIFAR10, opté por ir directamente a la generación de texturas de videojuegos, ya que los ajustes hechos en CIFAR10 pueden no ser útiles para la generación de imágenes en aplicación. Además, estas pruebas ya han sido útiles para adaptar el código y hacer las primeras pruebas a color.

Capítulo 7
Aplicación

Con base en los conocimientos adquiridos hasta ahora, seleccionamos un videojuego para nuestra generación de texturas. Detallaremos las características de las texturas utilizadas en el juego, incluyendo su formato y los ajustes requeridos para optimizar los valores.

7.1. Elección de Videojuego

El objetivo era encontrar un videojuego que a priori tuviera texturas no muy complejas para obtener buenos resultados, así como facilidad para incorporar las texturas generadas al videojuego. El videojuego elegido es Minecraft, debido a la gran variedad de texturas de bloques y objetos que ofrece, no solo con las texturas originales del juego, sino por los paquetes de texturas que la comunidad crea.

Minecraft es un videojuego de mundo abierto desarrollado por Mojang Studios. Se caracteriza por su estilo de gráficos pixelados y su enfoque en la construcción y exploración. Ofrece una amplia variedad de biomas para explorar, donde los jugadores pueden recolectar recursos, construir estructuras y enfrentarse a diversos desafíos. Su popularidad radica en sus mecánicas y en sus opciones multijugador, permitiendo a los jugadores reunirse con otros para explorar y disfrutar de las mecánicas del juego.

Minecraft tiene una comunidad con la que estoy familiarizado y encuentro gran utilidad en la generación de texturas. Continuamente usuarios o diseñadores crean diferentes paquetes de texturas cambiando el aspecto por defecto del videojuego para que otros usuarios le den uso. Por tanto, la generación de este tipo de texturas podría ser usada por los usuarios como herramienta para crear nuevos paquetes y basar el diseño de los mismos en las generaciones obtenidas

7.2. Bloques

El universo de Minecraft está formado por bloques o cubos con los que se forman el entorno y las estructuras. Es precisamente en estos bloques donde se incorporan las texturas del videojuego. Por ello, debemos entender su estructura.

Los bloques de Minecraft son cubos que necesitan de 6 caras para conformar la textura de un bloque completo. Véase la Figura 7.1

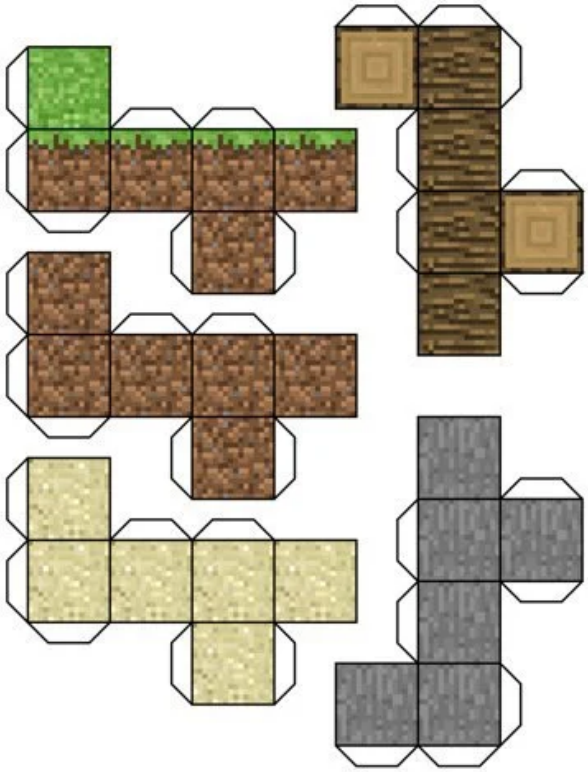


Figura 7.1: Estructura de los bloques en Minecraft[18]

Como podemos observar, hay bloques cuyas caras son diferentes y bloques con todas ellas iguales. Las texturas de cada bloque se incorporan en el juego como imágenes 16x16, generalmente en formato PNG. Los bloques con diferentes caras se incorporan como imágenes independientes con nombres reconocidos por el juego, por ejemplo, el bloque de piedra 'Stone_slab' se conforma de varias imágenes 'Stone_slab_side' 'Stone_slab_top' 'Stone_slab_bottom'. Debido a que las imágenes del VAE se generan de manera totalmente independiente, nos centraremos en bloques cuyas caras sean iguales.

Algunos de los bloques por defecto de Minecraft son los siguientes:

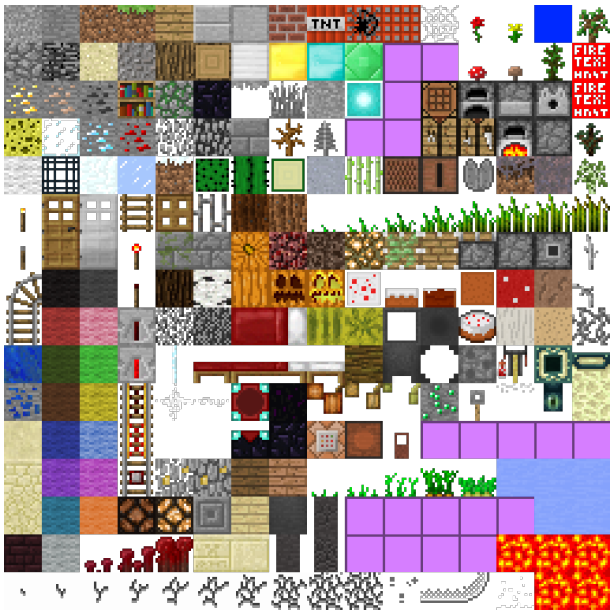


Figura 7.2: Bloques por defecto Minecraft [29]

El objetivo tras el entrenamiento es sustituir parcial o completamente dichas texturas por defecto por otras generadas con el modelo VAE entrenado.

7.2.1. Conjunto de datos

La primera fase del desarrollo consiste en buscar un conjunto de datos con un tamaño lo suficientemente grande para obtener buenos resultados. Los conjunto de datos usados en las pruebas iniciales MNIST y CIFAR10 contienen alrededor de 50.000 imágenes, por tanto, buscamos conseguir uno que tenga como mínimo 10.000 imágenes, todas ellas de 16x16. Tras no conseguir encontrar ningún conjunto de datos que se ajuste a nuestras necesidades, opté por crear uno propio. Para ello, se utilizan dos páginas web [23, 26] de las que se extraen diversos paquetes de texturas confeccionando un conjunto de datos formado por 10.116 imágenes.

7.2.2. Ajustes

Una vez escogido el conjunto de datos, hay que modificar nuestro código usado con CIFAR10 para adaptarlo al mismo. Las imágenes son ahora de 16x16, por lo que debemos ajustar el tamaño de la entrada del modelo, así como la función de error y otros valores. También ha sido necesario actualizar los canales de color, ya que CIFAR10 tiene 3 canales de color RGB, mientras que los bloques de Minecraft tienen 4 canales RGBA, incluyendo el de transparencia de las imágenes.

Además, ha sido necesario incluir código para crear el conjunto de datos de las imágenes locales, puesto que el conjunto de datos ya no se importa como se hacía en MNIST y CIFAR10.

7.2.3. Pruebas

Comenzamos a hacer pruebas con los cambios incluidos y con los parámetros con los que hemos obtenido los mejores resultados con CIFAR10:

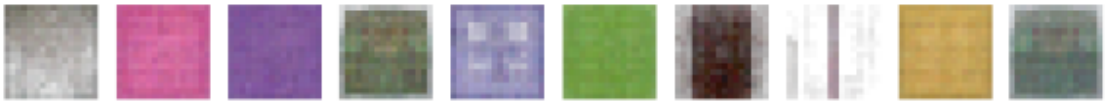


Figura 7.3: Primera prueba Texturas

Observamos que los resultados son bastante buenos. El cambio np.clip(x, 0,1) explicado anteriormente que hacía que los valores menores que 0 sean 0 y los que se pasen de 1 sean 1 ha permitido que estas primeras muestras no contengan contrastes. Debido a esta primera prueba exitosa guardamos el modelo en un archivo .pth para su posterior uso. Procederemos a generar una gran cantidad de muestras para ver si podemos encontrar patrones en las mismas e identificar bloques. Véase la Figura7.4.

Figura 7.4: Patrones bloques

En algunas muestras ya podemos identificar algunas texturas. Podemos ver un raíl, un arbusto o árbol, un bloque de piedra. Sin embargo, es complejo identificar texturas de bloque si no estás familiarizado con el videojuego. Por ello, mostraremos de manera más visual y dentro del juego (in-game) las texturas generadas en el capítulo de resultados.

7.3. Aspectos

Aparte de la modificación de texturas de bloques para dar un toque distinto al juego, los usuarios de Minecraft cambian también el aspecto de su personaje dentro del juego para diferenciarse del resto de usuarios. En la búsqueda de un conjunto de datos de texturas de bloques pude observar que a pesar de no hallar conjuntos de texturas, había

numerosos conjuntos de datos de aspectos de personaje, popularmente conocidos como su nombre en inglés skins.

Los aspectos de Minecraft tienen la siguiente estructura:

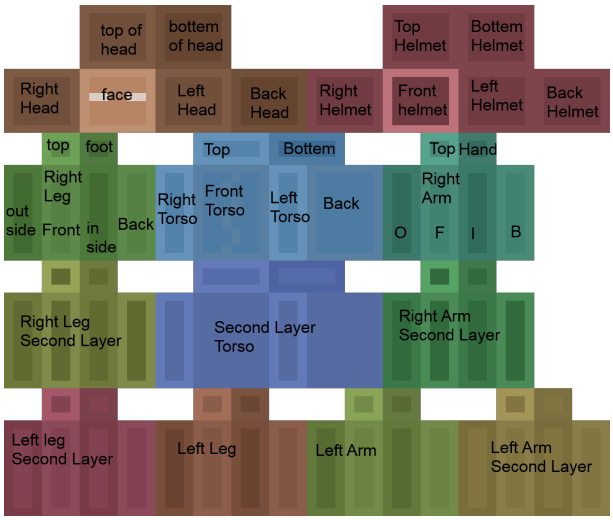


Figura 7.5: Estructura Skins Minecraft [28]

Son imágenes 64x64 del cuerpo del personaje de manera desplegada. No todas las partes definidas en la imagen están incluidas en cada aspecto (second layer, helmet, etc), ya que algunas partes del esquema están incluidas para añadir relieve o accesorios, como un caso, pero son opcionales. Los aspectos se incorporan en Minecraft manualmente o mediante el lanzador del juego. La imagen por defecto es la siguiente:

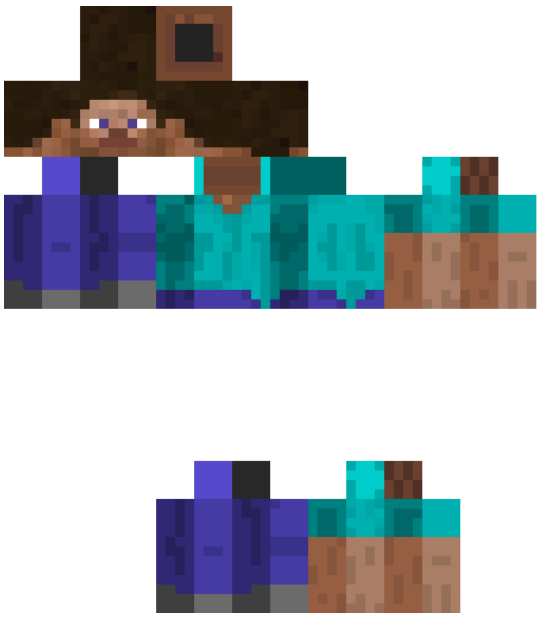


Figura 7.6: Aspecto por defecto en Minecraft [31]

7.3.1. Conjunto de datos

Para conseguir el conjunto de datos he usado una página conocida de Datasets, Kaggle, de la que he extraído un conjunto de imágenes de 20.000 skins [1].

7.3.2. Ajustes

Las modificaciones respecto a los bloques son mínimas, el único ajuste que se realiza es cambiar los tamaños anteriores de 16x16 a 64x64 en todas las imágenes y funciones.

7.3.3. Pruebas

Con los ajustes realizados, entrenamos el modelo, que esta vez requirió mucho más tiempo, debido al aumento de tamaño de texturas, así como un mayor número de muestras del conjunto de datos. El entrenamiento duró alrededor de 13 horas. Obtenemos un archivo .pth que contiene el modelo entrenado al que llamaremos para generar las muestras

Modificamos el código para llamar al modelo en lugar de entrenarlo y generamos imágenes como las mostradas en la Figura7.7

Figura 7.7: Primeras muestras generadas

Observamos que la mayoría de las skins se distinguen correctamente en la primera prueba. Posteriormente, procedemos a generar un gran número de muestras y obtenemos resultados como los mostrados en la Figura 7.8

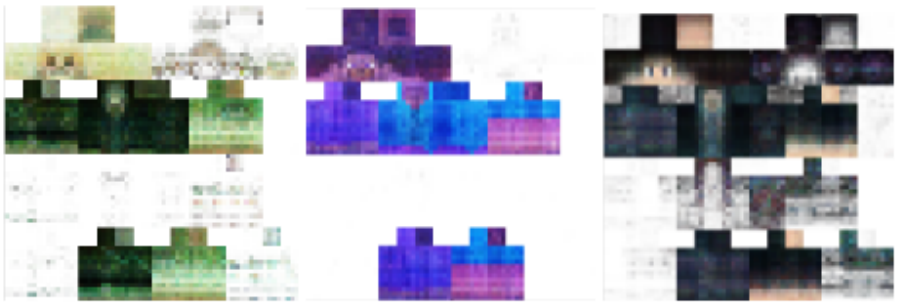


Figura 7.8: Ejemplo de aspectos generados

7.4. Generacion con VAE convolucional

Como hemos comentado en capítulos anteriores, las redes convolucionales tienen gran importancia en el mundo de la visión por computador. Por ello, hemos decidido ampliar nuestro código añadiendo capas convolucionales y observar qué resultados obtenemos con los bloques, así como con las skins.

7.4.1. Bloques

Tomando como punto de partida el código definitivo del VAE de texturas, añadimos 4 capas convolucionales en el encoder y otra 4 en el decoder. Además, reducimos el número de capas fully-connected a 4, siendo 1 para el encoder, 1 para la media, 1 para la varianza y 1 más para el decoder [34]. Durante el entrenamiento, utilizamos un tamaño de filtro (kernel) de 2x2 para atravesar las capas convolucionales y fully-connected. Manteniendo los mismos ajustes realizados en el VAE básico, procedemos a probar el VAE convolucional con las nuevas capas añadidas y obtenemos muestras como las de la Figura

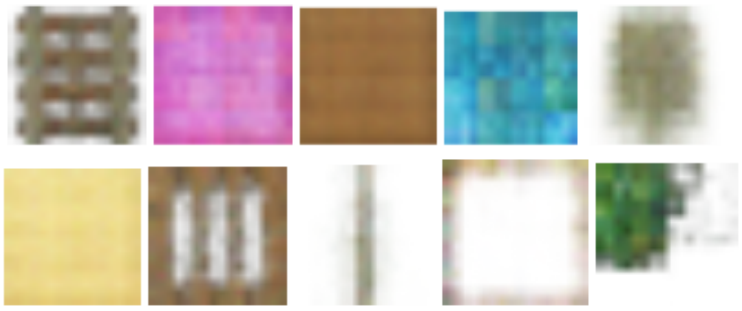


Figura 7.9: Primeras generaciones de bloques con VAE convolucional

Durante las primeras generaciones, notamos que existen bloques similares, pero también aparecen nuevos patrones en aquellos bloques que utilizan mayor cantidad de transparencias. Además, encontramos nuevas formas, como marcos, que pueden resultar

muy útiles en la creación del paquete de texturas.

Como hemos visto, las redes convolucionales se centran en establecer relaciones locales entre píxeles pasando el filtro (kernel) por la imagen. Por otro lado, las redes fully-connected usadas en el VAE básico, establecen relaciones entre píxeles independientemente de su lejanía. Esta diferencia a la hora de establecer relaciones entre los píxeles es lo que provoca que se generan resultados diferentes.

Sin embargo, en nuestro caso, los bloques tienen un tamaño muy pequeño de 16x16. Esto implica que el filtro de las capas convolucionales no tendrá mucho recorrido, lo que provoca que las diferencias entre la fully-connected y la convolucional no sean muy notorias.

7.4.2. Skins

Tomando como punto de partida el código definitivo del VAE de skins, añadimos 4 capas convolucionales en el encoder y 6 en el decoder. Además, reducimos el número de capas fully-connected a 2, siendo 1 para la media y otra para la varianza. Durante el entrenamiento, utilizamos un tamaño de filtro (kernel) de 4x4 para atravesar las capas convolucionales y fully-connected. Manteniendo los mismos ajustes realizados en el VAE básico, procedemos a probar el VAE convolucional con las nuevas capas añadidas y obtenemos muestras como las de la Figura

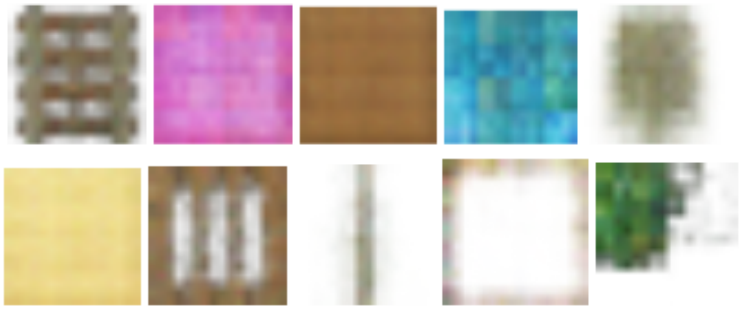


Figura 7.10: Primeras generaciones de bloques con VAE convolucional

Observamos que la generación de skins sigue siendo muy buena. En este caso, las dimensiones de las skins son de 64x64, que sigue siendo un tamaño muy pequeño para observar diferencias notorias.

Capítulo 8
Resultados

El objetivo final de este trabajo es confeccionar un paquete de texturas para poder incorporar al juego, así como probar las diferentes skins en el personaje. Para ello vamos a usar tanto el VAE básico como el convolucional en la generación para elegir los mejores resultados entre ambos tipos. Dicha elección se realizará de manera manual.

8.1. Bloques

Tras el entrenamiento y las pruebas iniciales se procede a la generación de grandes conjuntos de muestras para conformar un paquete de texturas para incorporar al juego y cambiar el aspecto visual del entorno. El aspecto de los bloques por defecto viene dado en la Figura 7.2. Tras la generación de los datos se diferencian 3 tipos de texturas claras: Un bloque completo, vegetación y raíles como podemos ver en la Figura 8.1

Figura 8.1: Tipos de bloque

8.1.1. Incorporación al juego

La mayor parte de las texturas de Minecraft son bloques completos, el objetivo es filtrar ahora aquellos bloques que se reconozcan como diferentes partes del entorno. Debido a mi familiaridad con este videojuego he tenido facilidad a la hora de escoger dichos bloques para poder conformar un paquete de texturas. Con un mapa propio creado previamente

he buscado texturas similares para poder visualizarlas de la mejor manera posible. El mapa con las texturas por defecto se muestra en la Figura 8.2

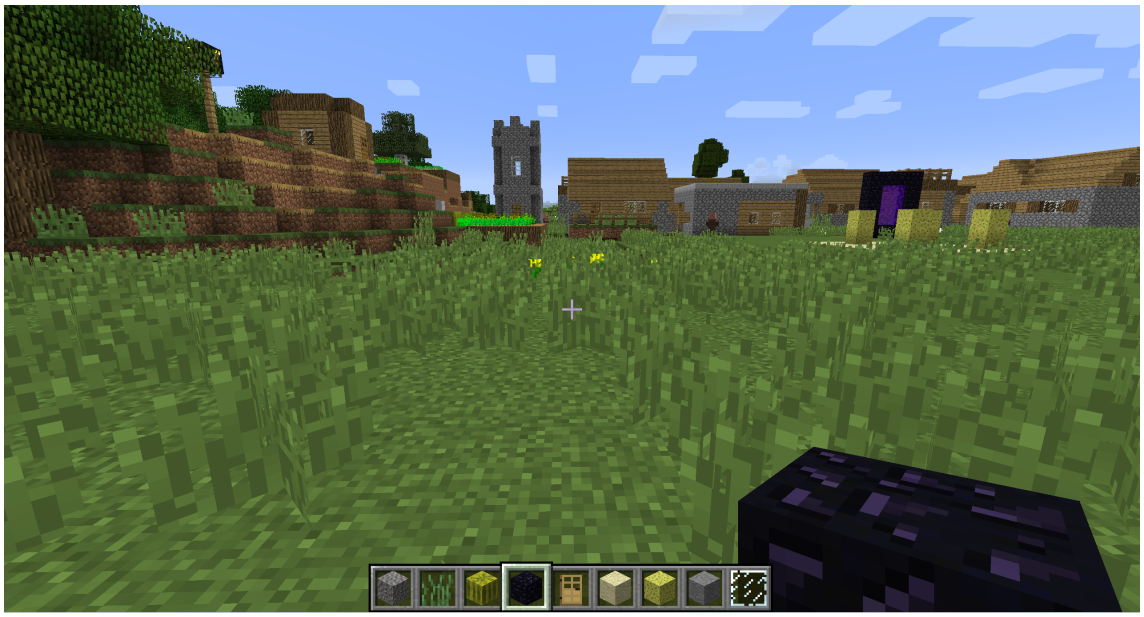


Figura 8.2: Texturas originales incorporadas en el juego

Fijándonos en los bloques usados en el mapa por defecto, he seleccionado texturas generadas creando un paquete de texturas con lo siguiente:

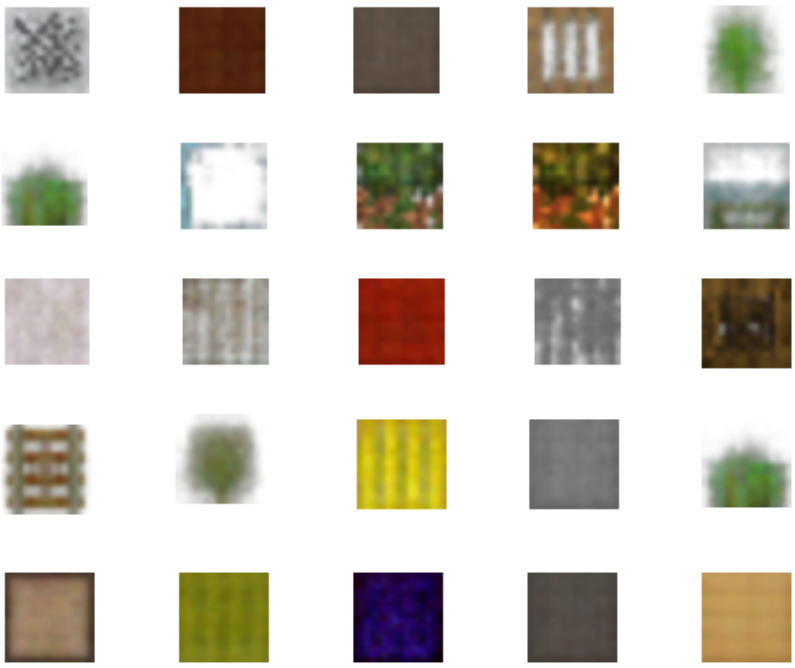


Figura 8.3: Texturas seleccionadas para crear el paquete

Se han cambiado elementos como troncos de madera, piedra refinada, malas hierbas, bloque terreno con césped, etc. Los cambios se observan en la Figura 8.4

Figura 8.4: Texturas generadas incorporadas en el juego

Podemos observar un cambio notorio en el aspecto del entorno del juego. Podemos ver los resultados de más de cerca en la Figura 8.5 en la que mostramos por pares diversos bloques mostrando a la izquierda la textura de bloque por defecto y a la derecha el bloque con la textura generada.

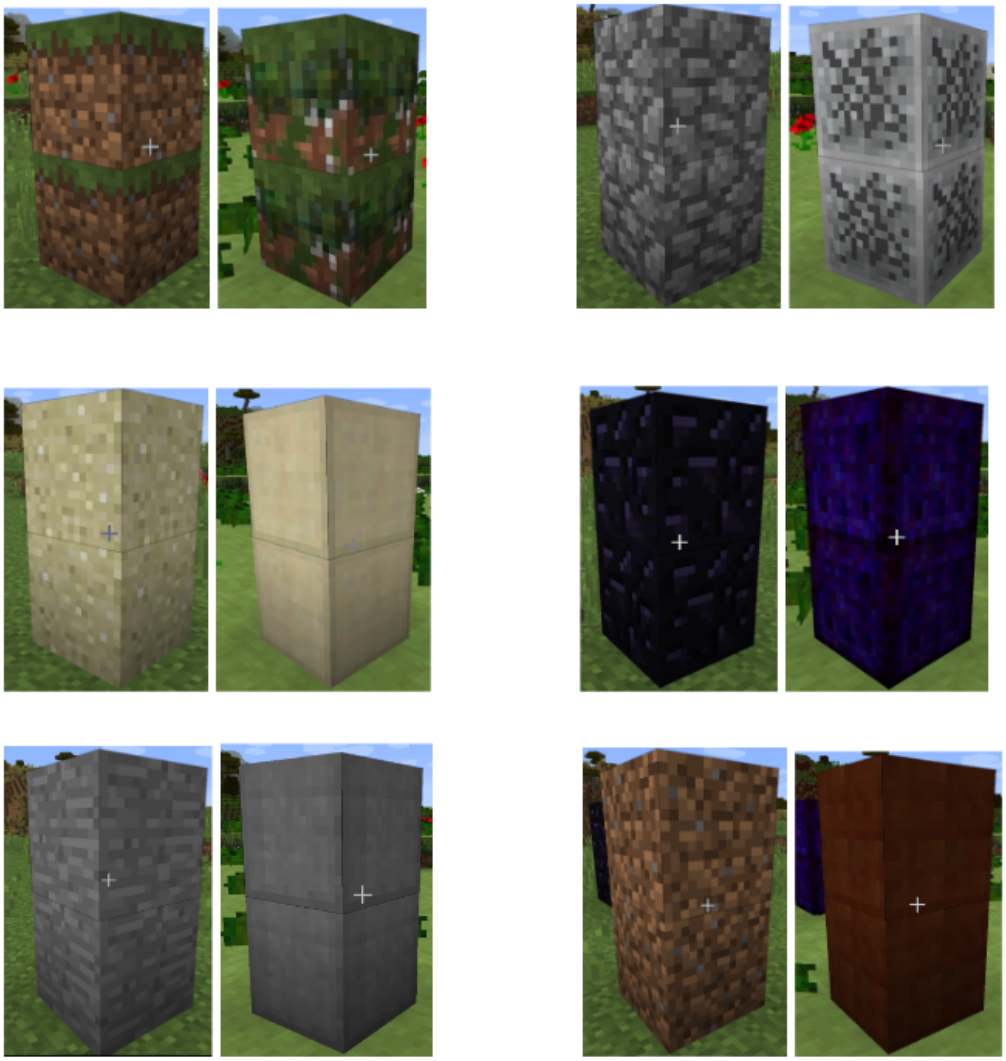


Figura 8.5: Texturas generadas incorporadas en el juego de cerca

Estos cambios nos abren un abanico de posibilidades a la hora de personalizar nuestro entorno. Con el modelo creado, otros usuarios podrían usar las generaciones para crear paquetes de texturas a su antojo.

8.2. Skins

Tras el entrenamiento y las pruebas iniciales se procede a la generación de skins al juego para cambiar el aspecto visual del personaje del jugador. El aspecto del personaje por defecto viene dado en la Figura 7.6. La aplicación de minecraft cuenta con soporte para personalizar el aspecto del jugador, por lo que probar diversos aspectos no ha sido complicado. Algunos de los resultados son los mostrados en la Figura 8.6

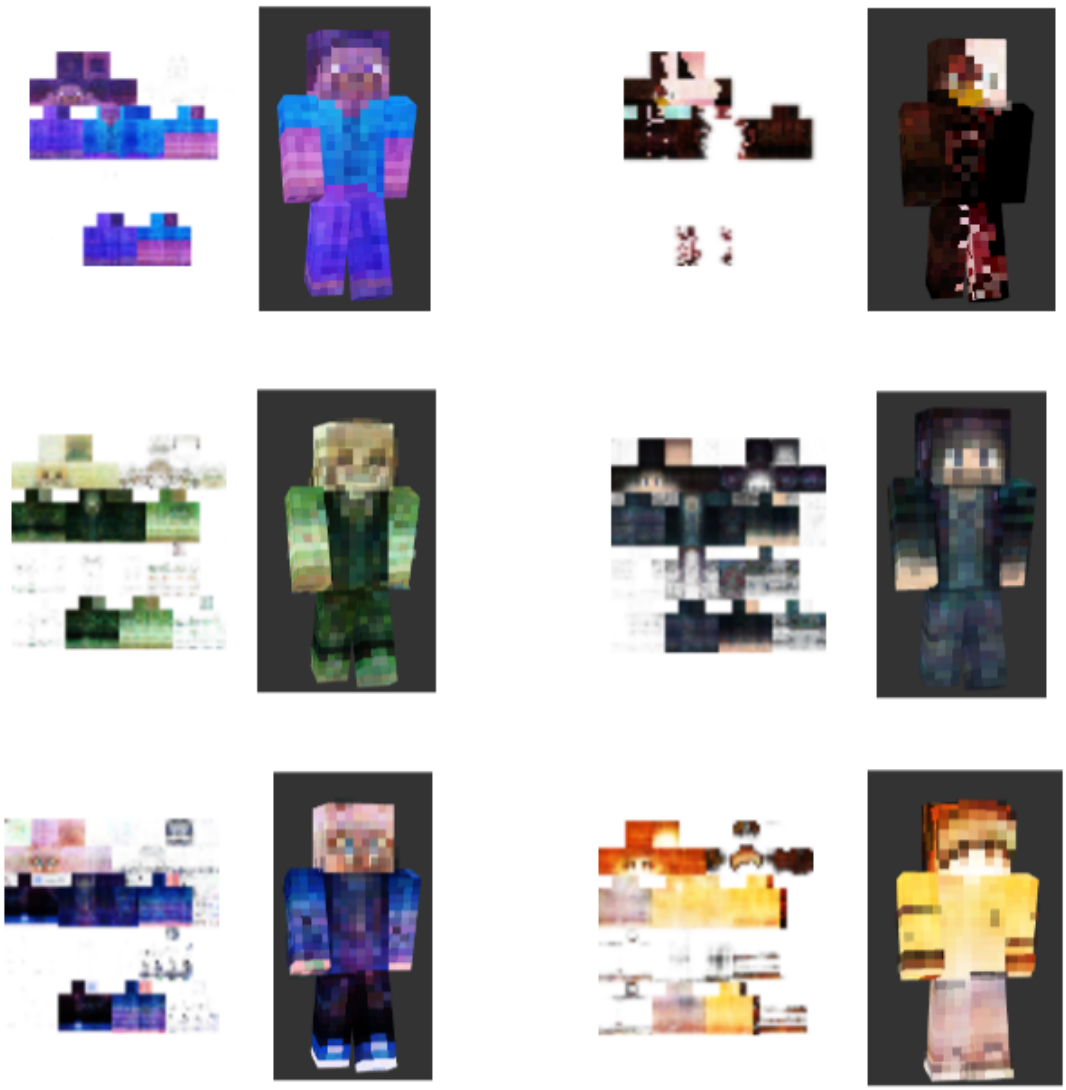


Figura 8.6: vista previa de aspectos generados

Podemos ver que la calidad de los resultados es muy alta. En todos ellos observamos como se ha mantenido la simetría en ambos lados, tanto piernas como brazos. Además, se distingue el rostro en cada uno de ellos.

8.2.1. Incorporación al juego

Como he comentado, la incorporación en el juego es sencilla importando la imagen desde la aplicación de Minecraft. Dentro del juego quedaría como podemos ver en la figura 8.7

Figura 8.7: aspecto generado incorporado en el juego

8.3. Código del proyecto

Todo el código del proyecto con los archivos e imágenes generadas se encuentra en GitHub en este enlace: https://github.com/MaiKKy34/TFG-Miguel-Vaes-Minecraft-textures/tree/main

Capítulo 9
Conclusiones

Durante el desarrollo de mi trabajo de fin de grado, he estudiado el aprendizaje automático y las redes neuronales, adquiriendo conocimientos fundamentales sobre inteligencia artificial. En este proceso, he comprendido en detalle los autoencoders y su variante generativa, los variational autoencoders (VAE).

La implementación del VAE nos ha brindado un mayor entendimiento sobre bibliotecas como PyTorch y torchvision, las cuales nos permiten implementar redes neuronales mediante código. Gracias a este proceso, hemos logrado adentrarnos en el funcionamiento interno de las redes y comprender de manera más profunda cómo se producen cambios en las imágenes a medida que atraviesan la red, observando las transformaciones en los vectores.

En cuanto a los resultados obtenidos, tras realizar pruebas iniciales con los conjuntos de datos MNIST y CIFAR10, he podido apreciar cómo los ajustes de los parámetros influyen en los resultados. Al aplicar el VAE para la generación de texturas de Minecraft, hemos logrado obtener resultados lo suficientemente buenos como para integrarlos directamente en el juego, sin necesidad de ningún tipo de post-procesamiento adicional.

Como líneas de trabajo futuro, sería interesante explorar la aplicabilidad de nuestros modelos en otros videojuegos con texturas más realistas, con el objetivo de comprender cómo se comportaría un VAE en ese contexto. Además, podríamos ver las diferencias de resultados entre un VAE básico y un VAE convolucional de manera más visual al trabajar con texturas de mayor tamaño que las utilizadas en Minecraft.

En nuestro proyecto hemos tenido que crear el paquete de texturas manualmente seleccionando entre las muestras generadas. Una posible solución a esto sería, entrenar el VAE por clases, de tal manera que no hiciera falta analizar las generaciones para identificar

las texturas.

También sería interesante probar otros modelos generativos populares, como los GANs y los diffusion models, en el mismo videojuego, con el fin de observar y comparar las diferencias y peculiaridades de cada modelo. Estas exploraciones nos permitirían ampliar nuestro conocimiento sobre las distintas técnicas generativas y sus aplicaciones específicas en el ámbito de los videojuegos.

Bibliografía

[1] Kaggle.

[2] Abien Fred Agarap. Implementing an autoencoder in pytorch. 2020.

[3] CC Aggrawal. Neural networks and deep learning: A textbook, 2018.

[4] Stability AI. Stable diffusion public release.

[5] Pragati Baheti. Activation functions in neural networks [12 types use cases].

[6] Health big data. Redes neuronales convolucionales.

[7] Vitaly Bushaev. How do we 'train' neural networks ? 2017.

[8] Praveen Kumar Chandaliya and Neeta Nain. Conditional perceptual adversarial variational autoencoder for age progression and regression on child face. In *2019 International Conference on Biometrics (ICB)*, pages 1–8. IEEE, 2019.

[9] CodificandoBits. Introducción a las redes neuronales recurrentes, 2019.

[10] Alexander Van de Kleut. Variational autoencoders (vae) with pytorch. 2021.

[11] Universidad Politécnica de Madrid. Newsletter Trimestral Cátedra iDanae - 1T21 : Algoritmos de machine learning. *Newsletter Trimestral Cátedra iDanae*, April 2021.

[12] Aaron Defazio. Técnicas de optimización ii.

[13] DotCSV. ¿qué es el descenso del gradiente? algoritmo de inteligencia artificial, 2018.

[14] DotCSV. ¿qué es una red neuronal? parte 1: La neurona, 2018.

[15] DotCSV. ¿qué es una red neuronal? parte 3 : Backpropagation, 2018.

[16] DotCSV. ¡redes neuronales convolucionales! ¿cómo funcionan?, 2020.

[17] Jaime Durán Suárez. Redes neuronales convolucionales en r: Reconocimiento de caracteres escritos a mano. 2017.

[18] fotosmagicas.info. Minecraft block structure.

[19] Eman Ijaz. Facial image reconstruction using autoencoders in keras.

[20] KDnuggets. An intuitive explanation of convolutional neural networks.

[21] Dhruv Khattar, Jaipal Singh Goud, Manish Gupta, and Vasudeva Varma. Mvae: Multimodal variational autoencoder for fake news detection. In *The world wide web conference*, pages 2915–2921, 2019.

[22] Romany F Mansour, José Escorcia-Gutierrez, Margarita Gamarra, Deepak Gupta, Oscar Castillo, and Sachin Kumar. Unsupervised deep learning based variational autoencoder model for covid-19 diagnosis and classification. *Pattern Recognition Letters*, 151:267–274, 2021.

[23] Planet Minecraft. Planet minecraft.

[24] Reo Neo. Beginner guide to variational autoencoders (vae) with pytorch lightning. 2021.

[25] OpenAI. Dall-e 2, ai system that can create realistic images and art from a description in natural language.

[26] Minecraft Texture Packs. Minecraft texture packs.

[27] Julio Manuel Vega Perez. Plantilla latex tfg-tfm.

[28] pikpng. Minecraft skins template.

[29] Mark William Reiley. Minecraft textures, hd png.

[30] Janosh Riebesell. Autoencoder.

[31] Mojang studios. Minecraft skin steve.

[32] M. Cristina Victorio. Neurona biológica, 2021.

[33] Wikipedia. Mnist database wikipedia.

[34] Galen Xing. Ccb skills seminar: Building a vae with pytorch!

[35] Roger Yong. Variational autoencoder(vae). 2021.