Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

# Trabajo de Fin de Grado

Grado en Ingeniería Informática

---

# Reconstrucción 3D de la mano mediante imágenes RGB

*3D hand reconstruction from RGB images*

Christian Jesús Pérez Hernández

---

La Laguna, *18* de *mayo* de 2022

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43826207Y profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Jose Demetrio Piñeiro Vera**, con N.I.F. 43774048B profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

**C E R T I F I C A N**

Que la presente memoria titulada:

*“Reconstrucción 3D de la mano mediante imágenes RGB”*

ha sido realizada bajo su dirección por D. **Christian Jesús Pérez Hernández**,

con N.I.F. 51153935A.

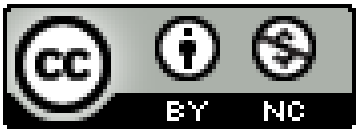
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 21 de noviembre de 2018

# Agradecimientos

En primer lugar me gustaría agradecer tanto al tutor D. Jesús Miguel Torres Jorge, así como al cotutor D. Jose Demetrio Piñeiro Vera por toda la ayuda prestada y por guiarme en la realización de este trabajo de fin de grado.

Así mismo quisiera agradecer a Dña. Carmen Elvira Ramos por la gestión y organización de la asignatura Trabajo de Fin de Grado.

# Licencia



**Resumen**

*Poder conocer en tiempo real la posición y forma de la mano facilitaría una gran cantidad de aplicaciones: realidad virtual, realidad aumentada o la interacción persona-computador, entre otras. El presente trabajo versa sobre la reconstrucción de la pose 3D de la mano a partir de imágenes RGB, donde se ha tratado de explorar un procedimiento mediante el cual estimar los ángulos de rotación de las articulaciones de la mano, utilizando un autoencoder del cual extraemos el codificador y un perceptrón multicapa que prediga las rotaciones a partir de las imágenes codificadas. Además, y de manera paralela, se ha creado un método de generación sintética de datos (imágenes de un modelo de mano genérico) para dicha problemática, se averiguó si todos los vectores codificados al decodificarse, generaban una mano y se exploró una alternativa al autoencoder utilizando unas redes adversarias generativas (GAN), para encontrar una representación codificada del espacio latente de las configuraciones de la mano.*

**Palabras clave:** Estimación de la pose de la mano, generación de imágenes de manos RGB, GAN, autoencoder, representación codificada del espacio latente.

**Abstract**

*Being able to regress the pose and shape of the hand would enhance many applications e.g., virtual reality, augmented reality, human-computer interactions, etc. In this final degree project, we explore a solution to estimate the hand joints rotation angles. The proposed pipeline first encodes the input image using an autoencoder and then feeds it to a multilayer perceptron which estimates the rotation angles. Furthermore, we present a method for synthetic data creation (hand images from a generic hand model), also we used linear interpolation to figure out if all codified vectors represent a valid hand image and we explore an alternative to convolutional autoencoder through a Generative Adversarial Network (GAN) to find a latent space's codified representation of hand configurations.*

**Keywords:** Hand pose estimation, synthetic data generation, GAN, autoencoder, latent space's codified representation.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1 Introducción

Ante el reciente interés y progreso en la realidad virtual y la realidad aumentada, se le ha dado especial importancia a la reconstrucción de la pose 3D de la mano, ya que representa una forma directa y natural de interactuar con el computador, además de otras múltiples aplicaciones como el reconocimiento de actividad, lenguaje de signos o robótica.

Para la realidad virtual o para la realidad aumentada, se debe tener en consideración distintas características que resultan interesantes. En primer lugar, se deberá minimizar al máximo posible el error de la reconstrucción de la postura de la mano. Además, es deseable que esta sea en tiempo real, pues el usuario tiene que recibir una respuesta instantánea de acorde a sus movimientos, con el objetivo de que la experiencia sea más inmersiva. Resulta conveniente también utilizar cámaras RGB, pues aunque existen numerosos estudios que utilizan mapas de profundidad [1, 3, 4], ya que permiten obtener una mayor información de la escena, ofreciendo resultados en general más precisos. No obstante, estas cámaras son considerablemente más caras y complejas que las cámaras RGB y son confiables solo en espacios interiores. Aunque también, se pueden obtener mapas de profundidad utilizando cámaras convencionales, mediante una cámara estereoscópica.

Estimar la posición de la mano en 3D a partir de una única imagen RGB, es un proceso complicado debido a las múltiples ambigüedades que surgen, como la auto oclusión o la ausencia de información de la profundidad. Es por ello, que se utilizan materiales que permitan obtener una información más completa mediante marcadores o guantes, restringiendo la aplicación de estas tecnologías a entornos específicos. Los estudios más recientes optan por utilizar mayormente imágenes de profundidad [1, 2, 3, 4, 5] o imágenes RGB [7, 8, 9, 10] como entrada.

En este trabajo, presentamos una aproximación para estimar las rotaciones de las articulaciones y generar la pose de la mano a partir de estas, a través de una única cámara RGB y sin ser necesario equipamiento especial. Para el entrenamiento de nuestra red, generamos un conjunto de datos sintético con anotaciones de las rotaciones para cada mano generada. Nuestra aproximación consiste en codificar la imagen de entrada utilizando el codificador que extraemos de un autoencoder, previamente entrenado, para que después, una segunda red realice la estimación de los valores de las rotaciones para cada articulación. Además, exploramos un método para averiguar si todos los vectores codificados por el autoencoder corresponden a una mano válida, usando una interpolación lineal entre las imágenes codificadas y otra alternativa con la que encontrar una representación codificada del espacio latente de las configuraciones de la mano, mediante una red generativa antagónica (RGA), también conocida como GAN en inglés, término que utilizaremos para referirnos a ella.

Para llevar a cabo este trabajo, inicialmente se establecieron los siguientes objetivos:

- **Análisis del estado del arte**. Para conocer cómo se había llevado a cabo la estimación de la pose de la mano en otras investigaciones, así como para observar las diferentes técnicas que se estaban utilizando para la estimación de la malla 3D, alineación de la mano...
- **Generación del conjunto de datos**. Para posteriormente entrenar los modelos, es necesario contar con un conjunto de datos, por ello se decidió como segundo objetivo crear un programa que genere imágenes de manos de manera sintética, utilizando para ello la mano de un modelo 3D de makehuman [34].
- **Implementar y entrenar diferentes modelos de aprendizaje automático**. Dentro de este objetivo se pretende implementar y entrenar un autoencoder, un perceptrón multicapa y una red GAN.
- **Análisis de los modelos**. Una vez implementados, se realizaría un análisis comparando los resultados de estos después de realizar ajustes a los distintos hiperparámetros, como pueden ser número de filtros, épocas, tamaño del kernel, conjunto de datos…

# Capítulo 2 Antecedentes y Estado del Arte

La reconstrucción de la pose 3D de la mano desde un único punto de vista monocular, es una problemática que lleva muchos años en estudio. En este apartado, revisaremos cuales son las técnicas más utilizadas y cómo han afrontado anteriormente dicho problema.

En un inicio, métodos como [11, 12] ajustaban la malla de una mano genérica a la imagen de profundidad usando una optimización iterativa. Otros, como de la Gorce et al [13] ajustaban el modelo de una mano utilizando características de las imágenes, como los bordes, color de la piel, silueta… Con los problemas que ello conlleva, las sutiles variaciones de color o la oclusión de las manos.

Generalmente, podemos dividir los estudios en función del tipo de entrada que utilizan, aquellos que usan una entrada RGB y aquellos que usan mapas de profundidad. Aunque en nuestro trabajo sólo predecimos las rotaciones de las articulaciones, normalmente utilizan redes neuronales profundas para estimar las coordenadas tanto 2D como 3D de las articulaciones de la mano, a partir de las cuales predecir los vértices de la malla o los parámetros de modelos paramétricos como puede ser MANO [14]. Estas investigaciones flaquean en la alineación de la malla 3D con la mano del usuario, la cual resulta importante si se quiere dar una experiencia de realidad aumentada, por ello trabajos como [15, 16] utilizan renderizado diferenciable para alinear la mano.

Existen además estudios, como Yohn Yang et al [17] que importan la información temporal que aporta el movimiento de las manos a sus redes neuronales, intentando mejorar el rendimiento de estas con secuencias de imágenes en movimiento de la base de datos BigHand2.2M [18].

Entre los estudios que utilizan como entrada imágenes RGB, Shangchen Han et al [19], llevan a cabo un sistema para realizar el seguimiento de las manos utilizando 4 cámaras RGB, a partir de las cuales recuperar los puntos claves que permiten estimar la malla. Otros, utilizan modelos paramétricos para recuperarla, Lixin Yan [20] implementan una red bisecada, de tal manera que pueden predecir los mapas de calor 2D y la silueta de la mano con dos decodificadores, recuperando por último los parámetros de MANO, siguiendo la implementación de Zhou et al [21].

Shile et al [22] y Ge et al [23] con PointNet [24] utilizan otro tipo de dato como entrada, nubes de puntos desordenadas. Javier et al [25] tratan de capturar las manos y el cuerpo en su conjunto, para dotar a personajes virtuales de un comportamiento realista. Para capturar el cuerpo y las manos utilizan un escáner 4D, a la salida de este las manos pueden tener mucho ruido y una baja resolución, incluso pudiendo llegar a desaparecer.

# Capítulo 3 Fases y desarrollo del proyecto

## 3.1 Generación del conjunto de datos sintético

Durante la primera fase, se realizó un programa para crear conjuntos de imágenes de manos sintéticas, así como anotar sus rotaciones. Se optó por utilizar cuaterniones para representarlas, con el objetivo de evitar el bloqueo del cardán que surge al usar ángulos euler. Este conjunto de datos, sería utilizado en un futuro para entrenar los distintos modelos.

Para ello se extrajo un modelo 3D de un cuerpo de Makehuman [26], el cual fue transformado en Blender para obtener únicamente el modelo de la mano. Además, se le añadió un esqueleto con un total de 20 huesos, que permitieran rotar todas las articulaciones de la mano. Como se aprecia en la siguiente figura.

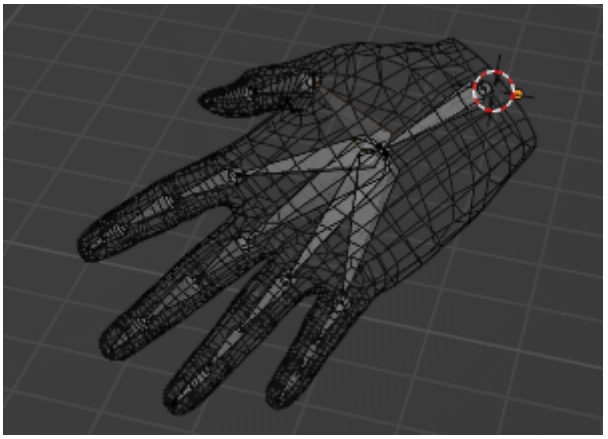


**Figura 1.** Corte de la mano en Blender y esqueleto.

Dicho modelo se importó a Unity, donde se implementaría el programa de la siguiente manera. Se recorren los huesos que conforman la mano asignando a cada uno de estos una nueva rotación y se comprueba si esta produce colisiones con alguna de las articulaciones restantes. En caso de colisión se repite el procedimiento hasta que la rotación nueva sea válida. Realizado este procedimiento con todas las articulaciones, se toma una captura de pantalla y se guardan los valores de las rotaciones de todas las manos en un fichero de texto. Se añadió, adicionalmente, un parámetro mediante el cual poder modificar las dimensiones de la captura de pantalla, de esta manera se pueden crear conjuntos de datos de distintos tamaños.

Para la realización del programa debíamos tener en cuenta ciertas restricciones, las

articulaciones de la mano no pueden rotar libremente 360º, por lo que el primer paso sería establecer de manera manual límites inferiores y superiores para cada articulación, también era necesario que la mano generada fuera una mano válida. Entendemos por válida en este contexto, una mano cuyas articulaciones no producen ninguna colisión entre sí y cuyos ángulos de rotación se encuentran dentro de los límites fijados.

Inicialmente, se intentó realizar la comprobación de colisiones utilizando la función de Unity *Physics.CheckCapsule*, creando una cápsula con centro en cada una de las articulaciones. En caso de que una cápsula de colisión se encuentre dentro o en contacto con la misma, nos devolverá un booleano indicando que se ha producido una colisión. Realizaríamos tantas rotaciones aleatorias como sea necesario, hasta que la articulación estuviera en una posición válida. Debido a un error, en el cual, cuando los dos centros de la cápsula se encuentran muy cerca uno del otro devuelve valores erróneos, fue descartada esta opción.

Finalmente, se optó por utilizar el evento *OnTriggerEnter*, generando una rotación aleatoria para las articulaciones que estén en contacto. De esta manera, detectamos correctamente las colisiones y además solucionamos un inconveniente que tenía la anterior implementación, pues podía ocurrir que dada una articulación, esta no tuviera dentro de su rango de ángulos, una rotación en la cual no colisionase con otra.

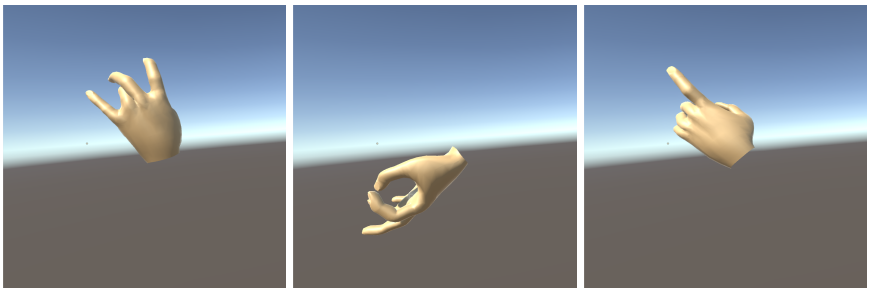


**Figura 2.** Imágenes generadas de manera aleatoria

Como se observa en la figura 2, la mano es rotada aleatoriamente para generar datos desde distintos puntos de vista, permitiendo así tener una mayor variedad de imágenes.

En cuanto a la arquitectura, la mano está formada por 20 *gameObjects*, cada uno contiene un *script, FingerJoint*, el cual se encarga de rotar la articulación a la que está asignado. Otro *script* que actúa de controlador, ejecuta el método *Move* de la clase *Hand,* la cual controla todas articulaciones y llama a su vez al método *Move* de cada *gameObject,* además detecta cuando las articulaciones dejan de estar en colisión unas con otras. En este momento se entiende que la postura de la mano es válida y el *script* controlador llama a la clase dedicada a las capturas de pantalla para guardar la imagen y a la clase *Hand*, para guardar las rotaciones de cada una de las articulaciones.

Como salida, el programa guarda en una carpeta previamente seleccionada por el

usuario, toda la información generada (imágenes y fichero de rotaciones). Las imágenes son guardadas en formato PNG dentro de otra carpeta *Images* para conservar el orden y el fichero en formato TXT se guarda en la carpeta raíz.

## 3.2 Creación del autoencoder

Pudiendo generar un conjunto de datos, el siguiente paso consistió en implementar un autocodificador (autoencoder) [35]. Una red neuronal, usada para aprender a codificar datos a un espacio latente y a decodificar, de tal manera que la salida sea lo más similar posible a la original. De esta manera, podríamos entrenar un codificador sin supervisión, para posteriormente dar como entrada las imágenes codificadas al perceptrón multicapa.

La implementación de un autoencoder depende de dos partes bien diferenciadas, el codificador, que deberá transformar los vectores de $R^n \rightarrow R^p$ y el decodificador que realizará el proceso a la inversa $R^p \rightarrow R^n$.

```
x = layers.Conv2D(16, 6, activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, 6, activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(4, 6, activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(4, 6, activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)
```

**Figura 3.** Implementación del codificador.

Para implementar todos los modelos del proyecto, se utilizó el *framework* de Tensorflow con Keras. El codificador está compuesto por capas convolucionales 2D y capas *max pooling 2D*, utilizando estas últimas para reducir las dimensiones de los vectores. Puesto que utilizamos 4 capas de *maxpooling* con una ventana de (2,2), el vector de entrada se ve reducido en un factor de $2^4$, como se ve en el resumen de la arquitectura de la red en la siguiente figura.

```
Model: "Encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 64, 64, 3)]       0
 conv2d_13 (Conv2D)          (None, 64, 64, 16)        1744
 max_pooling2d_8 (MaxPooling  (None, 32, 32, 16)       0
 2D)
```

```
 conv2d_14 (Conv2D)           (None, 32, 32, 8)         4616
 max_pooling2d_9 (MaxPooling  (None, 16, 16, 8)         0
 2D)
 conv2d_15 (Conv2D)           (None, 16, 16, 4)         1156
 max_pooling2d_10 (MaxPoolin  (None, 8, 8, 4)           0
 g2D)
 conv2d_16 (Conv2D)           (None, 8, 8, 4)           580
 max_pooling2d_11 (MaxPoolin  (None, 4, 4, 4)           0
 g2D)
=================================================================
Total params: 8,096
Trainable params: 8,096
Non-trainable params: 0
_________________________________________________________________
```

**Figura 4.** Resumen de la implementación del codificador, se usó la siguiente secuencia de filtros 16, 8, 4, 4 en las capas convolucionales. Para imágenes RGB de 64x64 píxeles la dimensión de nuestro espacio latente es de (4,4,4).

```
x = layers.Conv2D(4, 6, activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(4, 6, activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(8, 6, activation='relu', padding="same")(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(16, 6, activation='relu', padding="same")(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(3, 6, activation='sigmoid', padding="same")(x)
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

**Figura 5.** Implementación del decodificador y creación del autoencoder

Para implementar el decodificador, debemos diseñar una red que parta de las dimensiones de nuestro espacio latente y que como salida tenga las dimensiones originales de nuestra imagen. Para aumentar las dimensiones de los vectores utilizamos las capas *up sampling*. Como las imágenes de nuestro conjunto de datos son RGB, el número de filtros de la última capa convolucional corresponde al número de canales de la imagen. Tanto para el codificador como para el decodificador se utiliza ReLU como función de activación. Por último compilamos el modelo utilizando un optimizador de estimación del momento adaptativo (Adam) y como función de pérdidas entropía cruzada binaria.

```
Model: "Decoder"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_10 (InputLayer)       [(None, 4, 4, 4)]         0

 conv2d_31 (Conv2D)          (None, 4, 4, 8)           1160

 up_sampling2d_12 (UpSamplin  (None, 8, 8, 8)          0
 g2D)

 conv2d_32 (Conv2D)          (None, 8, 8, 8)           2312

 up_sampling2d_13 (UpSamplin  (None, 16, 16, 8)        0
 g2D)

 conv2d_33 (Conv2D)          (None, 16, 16, 16)        4624

 up_sampling2d_14 (UpSamplin  (None, 32, 32, 16)       0
 g2D)

 conv2d_34 (Conv2D)          (None, 32, 32, 32)        18464

 up_sampling2d_15 (UpSamplin  (None, 64, 64, 32)       0
 g2D)

 conv2d_35 (Conv2D)          (None, 64, 64, 3)         3459

=================================================================
Total params: 30,019
Trainable params: 30,019
Non-trainable params: 0
_________________________________________________________________
```

**Figura 6.** Resumen de la implementación del decodificador, donde se recupera un vector de dimensiones (64,64,3), a partir del espacio latente (4,4,4).

Una decisión importante a tomar a la hora de entrenar el autoencoder, sería decidir el tamaño de las imágenes de entrada, no solo por su impacto computacional, del cual hablaremos más adelante, sino que además la dimensión del espacio latente depende directamente de la imagen inicial. Se realizaron distintos entrenamientos con diferentes dimensiones.

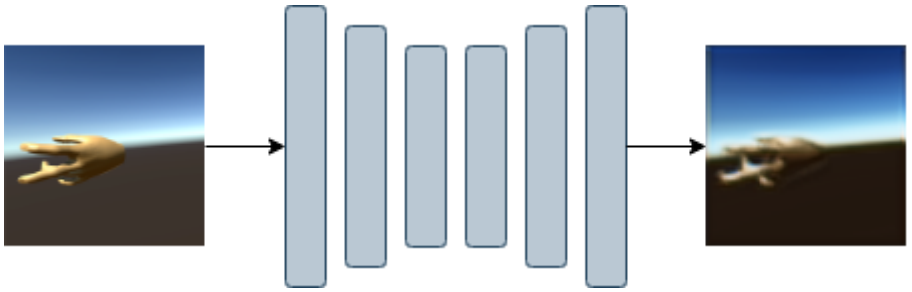


**Figura 7.** A la izquierda imagen de entrada 940x940 píxeles y a la derecha la imagen decodificada.

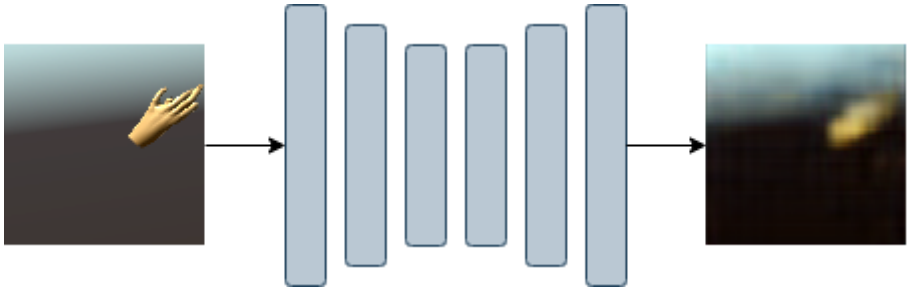


**Figura 8.** A la izquierda imagen de entrada 100x100 píxeles y a la derecha la decodificación.

Entrenar con imágenes de 940x940 píxeles, resultó demasiado costoso computacionalmente, aunque como se observa en la figura 7, la resolución de la imagen decodificada resulta superior a la de la figura 8. Para realizar estudios del número de filtros a elegir o del número de capas a utilizar con el objetivo de minimizar las pérdidas, se generó un nuevo conjunto de datos de 1000 imágenes de 64x64 píxeles. Se probaron dos arquitecturas distintas para la red, utilizando 3 conjuntos de capas convolucionales y capas de *max pooling* para el codificador y 3 conjuntos de capas convolucionales y capas *up sampling* para el decodificador y usando 4 conjuntos. Para el experimento, se utilizaron tamaños de lote de 256 imágenes, 50 épocas y las siguientes secuencias de filtros: para la red de 3 conjuntos, 32, 16, 8, 8, 16, 32 filtros y para la de 4 conjuntos, 32, 16, 8, 4, 4, 8, 16, 32 filtros.

| Arquitectura red | Pérdidas | Pérdidas conjunto de validación |
|---|---|---|
| 3 conjuntos | 0.3900 | 0.3898 |
| 4 conjuntos | 0.3895 | 0.3892 |

**Tabla 1.** Estudio sobre el impacto en las pérdidas del modelo en función de la arquitectura utilizada.

Un segundo estudio realizado, comprueba el efecto del tamaño del lote en las pérdidas de la red, en este caso se utilizó la arquitectura del autoencoder previamente implementado en la figura 3 y 4 y 50 épocas para el entrenamiento.

Aparentemente, cuanto menor sea el tamaño del lote menor serán las pérdidas del modelo, aunque las diferencias entre los valores no son muy acusadas.

| Tamaño del lote | Pérdidas | Pérdidas conjunto de validación |
|---|---|---|
| 256 | 0.3905 | 0.3894 |
| 128 | 0.3904 | 0.3901 |
| 64 | 0.3884 | 0.3885 |
| 32 | 0.3870 | 0.3868 |
| 16 | 0.3865 | 0.3861 |

**Tabla 2.** Efecto sobre las pérdidas del modelo ante los cambios en el tamaño del lote.

Aunque el método de reconstrucción de la imagen del autoencoder siempre tiene errores, se ha tratado a través de estos estudios reducir al máximo posible las pérdidas. En la siguiente figura, se puede observar un ejemplo de imagen generada utilizando el autoencoder anteriormente definido y 16 imágenes por lote.

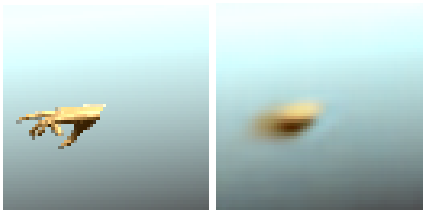
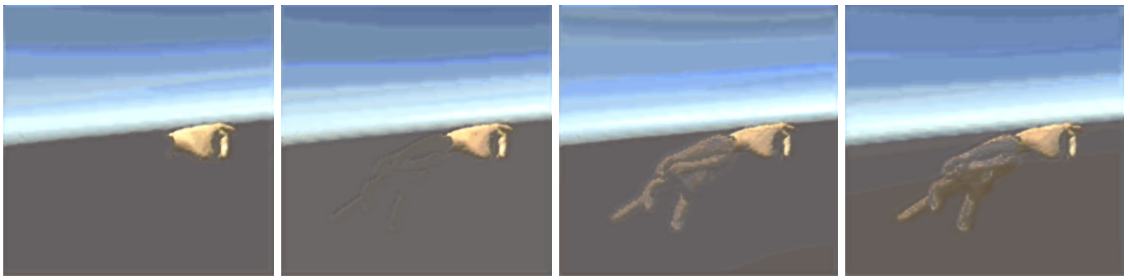


**Figura 9.** Imagen inicial e imagen generada por el autoencoder.

Una vez entrenado el codificador, se aplicó interpolación lineal a imágenes codificadas, para comprobar si todos los vectores codificados corresponden a una mano válida. En ese caso habríamos compactado el espacio de soluciones en uno más pequeño, con vectores de menor tamaño.

Se pretendía generar todas las posturas intermedias a partir de la interpolación lineal de la codificación de dos imágenes, obteniendo otra codificación, tal que al decodificar, el vector resultante sea una mano.

.



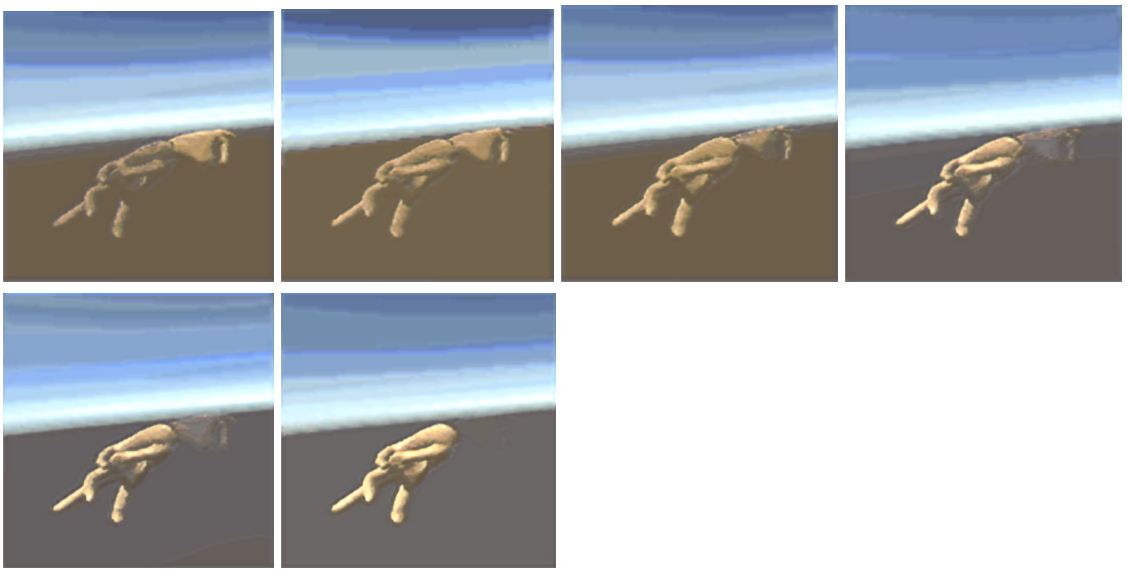


**Figura 10.** Interpolación lineal de dos imágenes codificadas, agregando $0.1$ al valor de $a$ en cada paso hasta llegar a 1.

La interpolación se realizó basándose en la siguiente fórmula, $v = (1 - a) * v1 + a * v2$, donde $v1$ corresponde al vector de la primera imagen codificada y $v2$ al vector de la segunda imagen. Como se puede observar, cuando la $a$ toma el valor de 0 el resultado que obtenemos es la primera imagen, mientras que cuando la $a$ vale 1 el resultado corresponde a la segunda imagen, siendo los resultados intermedios una combinación de estas. Las imágenes intermedias no son imágenes de manos válidas, pues la interpolación de los vectores corresponde a una interpolación de sus píxeles. Posiblemente el problema pueda estar en el diseño del autoencoder, pues un autoencoder convencional, al contrario que uno variacional (VAE), tiene un espacio latente con menor significado.

## 3.3 Perceptrón multicapa

En la siguiente etapa, se creó un perceptrón para llevar a cabo la estimación de las rotaciones de las articulaciones. Anteriormente, ya habíamos mencionado que el modelo de la mano contaba con 20 articulaciones. Al estar representando estas mediante cuaterniones (x,y,z,w), tendríamos que estimar las 80 coordenadas que conforman la pose de la mano.

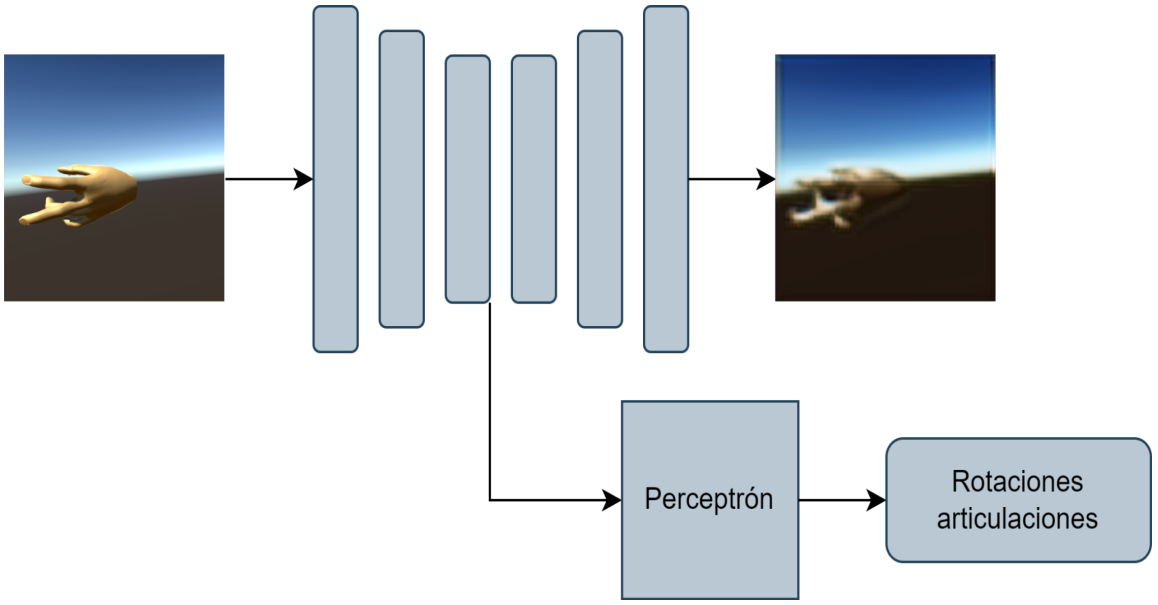



**Figura 11.** Diagrama de la estructura de la red, codificamos la imagen de entrada y estimamos las rotaciones de las articulaciones mediante el perceptrón.

```
x = layers.Flatten()(self.inputShape)
x = layers.Dense(128, activation="tanh")(x)
x = layers.LeakyReLU(alpha=0.2)(x)
x = layers.Dense(128, activation="tanh")(x)
x = layers.LeakyReLU(alpha=0.2)(x)
x = layers.Dense(128, activation="tanh")(x)
x = layers.LeakyReLU(alpha=0.2)(x)
x = layers.Dropout(0.4)(x)
x = layers.Dense(80, activation="tanh")(x)
self.mlpLayers = layers.LeakyReLU(alpha=0.2)(x)
```

**Figura 12.** Una de las implementaciones del perceptrón multicapa.

Como entrada, el perceptrón recibe la imagen codificada, por lo que primero tendríamos que convertir el vector en 2 dimensiones a un tensor de una única dimensión mediante la capa *flatten* para poder utilizar capas densas. Posteriormente, combinamos capas densas con la función de activación tangente hiperbólica con capas *leaky relu.* Por último, una capa densa con el número de neuronas igual al número de coordenadas a predecir, seguido de una función de activación *leaky relu* estiman las coordenadas de las articulaciones de la mano.

Con motivo de probar la precisión del modelo, se ha creado un programa que realiza inferencias sobre el mismo, obteniendo el porcentaje de acierto del perceptrón. Para ello recorremos las coordenadas predichas y las comparamos con las coordenadas reales, si el valor absoluto de la diferencia es mayor a un límite, tomamos dicha predicción como errónea.

Si nos fijamos en el rango de los valores de los cuaterniones, ver figura 13. Podemos ver que se encuentran entre $10^{-3}$ y $10^{-1}$, por ello como límites hemos probado $0.1,\ 0.01\ y\ 0.001$.

0,207592 -0,02851993 0,06865209 -0,9753867
-0,06077047 0,08917465 -0,04336016 -0,9932144
0,04715686 0,01933461 -0,0180279 0,9985377

**Figura 13.** Fichero quaternions.txt, donde se guardan las rotaciones de las articulaciones.

Para el entrenamiento de la red se han realizado múltiples pruebas siempre intentando mejorar la precisión del modelo. Se ha probado a utilizar como función de activación la función ReLU en lugar de la tangente hiperbólica, se ha probado a modificar el número de neuronas de cada capa, se ha aumentado el número de capas de perceptrón, se ha introducido una capa de descarte para evitar el sobreajuste… Como función de pérdidas estamos utilizando el error cuadrático medio y el optimizador adam, al igual que en los otros modelos. A continuación presentamos una tabla con las pérdidas y la precisión del modelo.

| Arquitectura de la red | Pérdidas del conjunto de entrenamiento | Pérdidas del conjunto de validación | Precisión del modelo |
|---|---|---|---|
| 128,lr,128,lr,256,lr, 80,lr tanh | 0.0867 | 0.1005 | 0.116 |
| 128,lr,128,lr,128,lr, 80,lr tanh | 0.0868 | 0.1000 | 0.128 |
| 128,lr,128,lr,128,lr, 80,lr relu | 0.1335 | 0.1445 | 0.0 |
| 64,lr,64,lr,128,lr,80, lr tanh | 0.0908 | 0.1011 | 0.13 |
| 64,lr,64,lr,128,lr,80, lr tanh sin dropout | 0.0869 | 0.1020 | 0.147 |

**Tabla 3.** Como resulta complicado describir la arquitectura de una red, se establecerán las siguientes convenciones, el número de capas densas vendrá dado por el número de neuronas utilizadas, por ejemplo: un perceptrón de 3 capas densas con 64 neuronas en todas sus capas, se representaría como 64,64,64. Se especificará a continuación la función de activación que se ha usado en todas sus capas densas y se utilizará la abreviatura *lr* para declarar que se ha empleado una capa *leaky relu.* Siendo el formato resultante 64,lr,64,lr,64,lr, lo que se traduce en 3 capas densas de 64 neuronas intercaladas por capas *leaky relu.*

En todos los casos no se supera el 15% de precisión, lo que demuestra que nuestra arquitectura planteada no es capaz de dar solución al problema. Para todos los entrenamientos, se ha usado un conjunto de datos de imágenes de 64x64 píxeles de 1000 elementos. Se ha usado un tamaño de lote de 256, aunque se probaron distintos tamaños, la precisión solo se veía alterada por un factor de $10^{-3}$. También

se probó a entrenar durante distintas épocas 30, 50 y 100, pero al igual que con el tamaño del lote, la precisión apenas se ve afectada.

## 3.4 Red GAN

La última fase de implementación, fue realizar una red GAN como alternativa al autoencoder para intentar encontrar la representación codificada del espacio latente.

Para implementar una red GAN, debemos definir las dos redes neuronales que la conforman: el discriminador que dada una imagen de entrada devuelve si esta es falsa o verdadera y el generador que a partir de un vector de valores aleatorios genera una imagen. La manera de entrenar la red consiste en enfrentar estas, de ahí su nombre, de tal forma que el discriminador aprenda a detectar las imágenes falsas y el generador aprenda a crear imágenes lo más realistas posibles.

```
def model(self):
  model = tf.keras.Sequential()
   model.add(layers.Conv2D(64,  4,  strides=(2,2),  padding='same',
   input_shape=[self.width, self.height, self.dimensions]))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(128, 4, strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(128, 4, strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Flatten())
    model.add(layers.Dropout(0.6))
    model.add(layers.Dense(1, activation="sigmoid"))
    return model
```

**Figura 14.** Implementación del discriminador.

Veamos nuestra implementación del discriminador, utilizamos una combinación de capas convolucionales 2D y capas *leaky relu* como funciones de activación, al final aplanamos el vector usando la capa *flatten* y por último, mediante una capa densa definimos la salida, un valor entre 0 y 1 que se traduce como la probabilidad de que la imagen introducida sea verdadera.

Si observamos la definición de keras para la capa convolucional, *tf.keras.layer.Conv2D(filters, kernel, strides=(1,1)...* Podemos ver que estamos definiendo una zancada (*strides)* de (2, 2), para reducir la resolución de los mapas de características, tal y como indica F. Chollet [28], intentado así evitar los gradientes

escasos que pueden ser inducidos por la función de activación *relu* y por la capa *max pooling*. Además, para que en la imagen generada no se observen patrones semejantes al tablero de ajedrez (ver figura 7), utilizamos un tamaño de núcleo (*kernel size)* divisible por el tamaño de la zancada, tanto en las capas convolucionales 2D, como en las capas convolucionales 2D transpuestas.

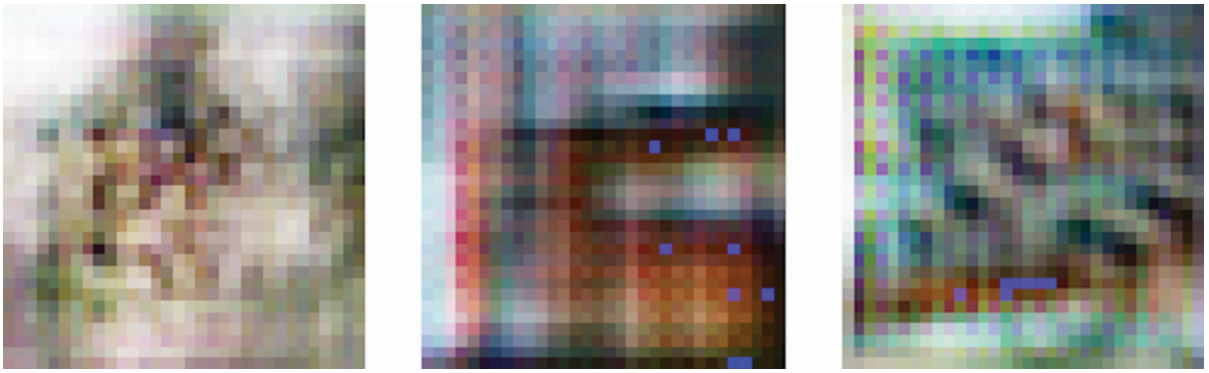


Figura 15. Patrones de tablero de ajedrez, causados por tener un tamaño del núcleo indivisible por el tamaño de la zancada.

También destacar, que utilizamos una capa *dropout*, con el objetivo de que el discriminador no acabe dominando al generador durante el entrenamiento y en caso de que ocurriera, podemos incrementar el ratio de descarte para evitarlo.

La elección de los filtros se realizó siguiendo la misma secuencia propuesta en [28], aunque no con todas las dimensiones de imagen. Se probaron múltiples secuencias empezando por 2 filtros hasta 64, siendo estas siempre múltiplos de 2.

```
Model: "discriminator"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)            (None, 32, 32, 64)        3136
 leaky_re_lu_9 (LeakyReLU)    (None, 32, 32, 64)        0
 conv2d_6 (Conv2D)            (None, 16, 16, 128)       131200
 leaky_re_lu_10 (LeakyReLU)   (None, 16, 16, 128)       0
 conv2d_7 (Conv2D)            (None, 8, 8, 128)         262272
 leaky_re_lu_11 (LeakyReLU)   (None, 8, 8, 128)         0
 flatten_1 (Flatten)          (None, 8192)              0
 dropout_1 (Dropout)          (None, 8192)              0
 dense_3 (Dense)              (None, 1)                 8193
=================================================================
Total params: 404,801
Trainable params: 404,801
Non-trainable params: 0
_________________________________________________________________
```

**Figura 16.** Resumen de capas del discriminador y formatos de salida de cada capa.

```
def model(self, latent_dimension, size):
  model = tf.keras.Sequential()
   model.add(layers.Dense(int(size / 8 * size / 8 * latent_dimension),
   input_shape=(latent_dimension,)))
   model.add(layers.Reshape((int(size / 8), int(size / 8),
   latent_dimension)))
   model.add(layers.Conv2DTranspose(latent_dimension, 4, strides=2,
   padding='same'))
   model.add(layers.LeakyReLU(alpha=0.2))
   model.add(layers.Conv2DTranspose(latent_dimension * 2, 4,
   strides=2, padding='same'))
   model.add(layers.LeakyReLU(alpha=0.2))
   model.add(layers.Conv2DTranspose(latent_dimension * 4, 4,
   strides=2, padding='same'))
   model.add(layers.LeakyReLU(alpha=0.2))
   model.add(layers.Conv2D(3, 5, activation='sigmoid',
   padding='same'))
   return model
```

**Figura 17.** Método que implementa el generador de la red GAN.

Para implementar el generador seguimos un proceso similar a la del discriminador, el generador deberá convertir un vector del tamaño del espacio latente en una imagen. Comenzamos con una capa densa la cual tiene $(s / 2^{c})^{2} * l$ neuronas, siendo $s$ el tamaño de la imagen, $c$ el número de capas convolucionales utilizadas en el discriminador y $l$ la dimensión latente, de esta manera producimos el mismo número de coeficientes que teníamos en la capa *flatten* del discriminador. Posteriormente, cambiamos la forma del vector de salida de la capa densa a una con dos dimensiones, para poder utilizar capas convolucionales 2D transpuestas. Seguimos utilizando las zancadas con el mismo valor que en el discriminador (2,2), en este caso, al estar utilizando capas convolucionales transpuestas aumentamos la resolución de los mapas de características, recuperando en la última capa las dimensiones de la imagen.

```
Model: "generator"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 8192)              1056768
 reshape (Reshape)            (None, 8, 8, 128)         0
 conv2d_transpose (Conv2DTra  (None, 16, 16, 128)       262272
 nspose)
 leaky_re_lu (LeakyReLU)      (None, 16, 16, 128)       0
 conv2d_transpose_1 (Conv2DT  (None, 32, 32, 256)       524544
 ranspose)
```

```
 leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 256)       0
 conv2d_transpose_2 (Conv2DT  (None, 64, 64, 512)       2097664
 ranspose)
 leaky_re_lu_2 (LeakyReLU)    (None, 64, 64, 512)       0
 conv2d (Conv2D)              (None, 64, 64, 3)         38403
=================================================================
Total params: 3,979,651
Trainable params: 3,979,651
Non-trainable params: 0
_________________________________________________________________
```

**Figura 18.** Resumen de capas del generador y los formatos de las salidas de cada capa.

Por último, definimos una clase GAN, que se encargará de conectar el generador y el discriminador, para cada una de las épocas realizaremos los siguientes pasos:

1. Generaremos puntos aleatorios en el espacio latente.
2. Crearemos las imágenes utilizando el generador y el ruido generado como entrada.
3. Combinaremos las imágenes generadas con las reales.
4. Entrenaremos el discriminador utilizando dicha combinación, etiquetando “falsa” para las imágenes generadas y “verdadera” para las reales.
5. Generaremos nuevos puntos aleatorios en el espacio latente.
6. Entrenaremos el generador con los vectores generados y esta vez etiquetaremos como “verdadera” las imágenes creadas. Actualizando los pesos del generador, intentando que el discriminador prediga como verdaderas las imágenes generadas.

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
     random_latent_vectors = tf.random.normal(shape=(batch_size,
     self.latent_dim)) # generamos los vectores del espacio latente
    generated_images = self.generator(random_latent_vectors)
     combined_images = tf.concat([generated_images, real_images],
     axis=0) # concatenamos las imágenes generadas y las reales
     labels = tf.concat([tf.ones((batch_size, 1)), tf.zeros((batch_size,
     1))],axis=0) # las etiquetamos correctamente
    labels += 0.05 * tf.random.uniform(tf.shape(labels))
    with tf.GradientTape() as tape: # entrenamos el discriminador
      predictions = self.discriminator(combined_images)
      d_loss = self.loss_fn(labels, predictions)
    grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
     self.d_optimizer.apply_gradients(zip(grads,
```

```
        self.discriminator.trainable_weights))
        random_latent_vectors = tf.random.normal(shape=(batch_size,
        self.latent_dim))
        # las imágenes generadas esta vez se etiquetan como verdaderas
    misleading_labels = tf.zeros((batch_size, 1))

    with tf.GradientTape() as tape: # entrenamos el generador
        predictions =
        self.discriminator(self.generator(random_latent_vectors))
      g_loss = self.loss_fn(misleading_labels, predictions)
    grads = tape.gradient(g_loss, self.generator.trainable_weights)
    self.g_optimizer.apply_gradients(zip(grads,
    self.generator.trainable_weights))

    self.d_loss_metric.update_state(d_loss)
    self.g_loss_metric.update_state(g_loss)
    return {"d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result()}
```

**Figura 19.** Implementación del código a ejecutar por cada época.

Para entrenar el modelo, definiremos además una función callback, que guarde al final de cada época imágenes creadas por el generador, permitiendo monitorear la generación de la red. Utilizamos entropía cruzada como función de pérdidas y dos optimizadores de estimación del momento adaptativo (Adam), uno para cada modelo. El criterio de parada se estableció de forma manual, en función de las características del entrenamiento y a través de experimentación.

Entrenar una red GAN es un proceso complicado, en el que no se persigue alcanzar un mínimo de optimización, como normalmente ocurre, sino que se trata de encontrar un equilibrio entre las dos redes. Es por ello que requieren de realizar cuidadosos ajustes a la arquitectura y a los parámetros de entrenamiento.

Durante este trabajo se han llevado a cabo múltiples entrenamientos, intentando generar una imagen lo más realista posible, como se puede ver en las figuras 12 y 8, este objetivo no ha sido cumplido con éxito pues las imágenes generadas están aún lejos de reproducir las imágenes originales.

Se comenzó realizando entrenamientos con imágenes de 940x940 píxeles, el problema de este conjunto de datos, es que las dimensiones resultaban muy grandes para aplicar un número considerable de filtros. Tanto en Google Colab con aproximadamente 12GB de RAM disponibles, como en el ordenador en el que se probó con 16GB, no pudo utilizarse más de 16 filtros, un conjunto de datos de 200 imágenes y un tamaño de lote de 1. Por lo que se tomó la decisión de probar con imágenes en escala de grises, para poder tener mayor libertad con la que probar modificaciones de los distintos hiperparámetros de la red.

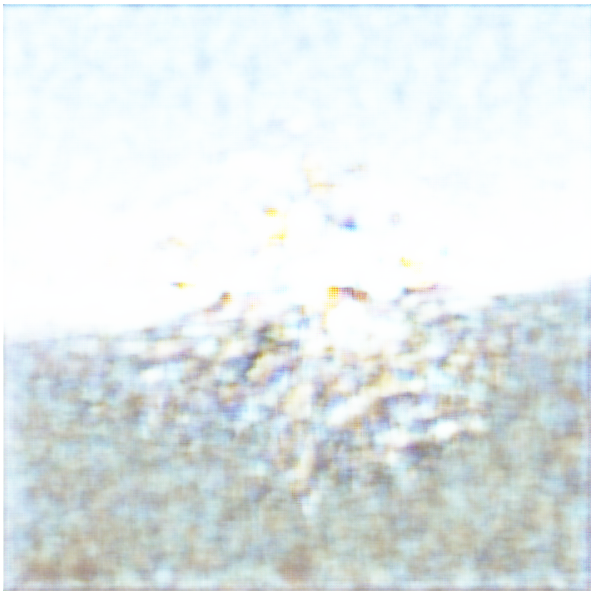


**Figura 20.** Primer entrenamiento con imágenes 940x940 de la red GAN, imagen generada tras 230 épocas

Para generar imágenes en escala de grises, se utilizó la librería Pillow. A través del método *convert*, se puede especificar el modo al que queremos convertir la imagen, “L” para escala de grises, “RGB” para convertir a imágenes RGB… Para poder entrenar, debemos además hacer un pequeño ajuste a la arquitectura de la red. Pues en este caso, en el generador los filtros en su última capa deben coincidir con los canales que tendrá la imágen, por tanto, debemos de modificar a 1 los filtros de dicha capa, pues las imágenes en escala de grises solo tienen un canal. Aún así fue insuficiente para poder modificar los parámetros de entrenamiento, pues seguía superando los 16GB de memoria RAM. En la figura 8, se puede ver el resultado tras 200 épocas de un entrenamiento en escala de grises, utilizando como filtros: 4 en la primera capa convolucional, 8 en la segunda y 8 en la tercera para el discriminador y 8, 16 y 16 filtros para las capas del generador. Se usó el mismo conjunto de datos y tamaño del lote que en el anterior entrenamiento.

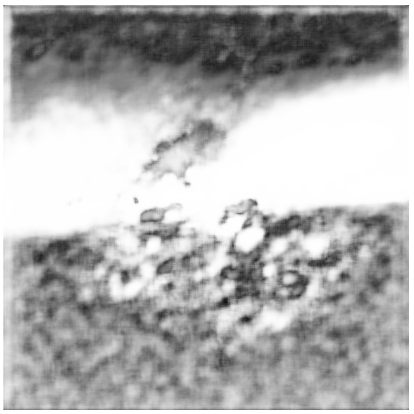


**Figura 21.** Imagen en escala de grises generada tras 200 épocas.

Finalmente, se optó por reducir la dimensionalidad a 100x100 píxeles y a 64x64, después. El problema que surge con tamaños que no resultan ser múltiplos de dos, como es el caso de las imágenes de 100 píxeles, es que se debe ajustar el relleno

(padding) de las capas convolucionales 2D transpuestas del generador, eliminando el relleno de algunas capas, como se puede ver en la figura 14, para que la imagen que crea tenga las mismas dimensiones que las del conjunto de datos. Razón por la cual se decidió un conjunto de datos con dimensiones que fueran múltiplos de dos.

```
model.add(layers.Conv2DTranspose(latent_dimension * 2, 4, strides=2))
 model.add(layers.LeakyReLU(alpha=0.2))
model.add(layers.Conv2DTranspose(latent_dimension * 4, 4, strides=2,
padding='same'))
 model.add(layers.LeakyReLU(alpha=0.2))
 model.add(layers.Conv2D(3, 5, activation='sigmoid'))
 return model
```

Figura 22. No aplicamos relleno ni en la primera, ni última capa convolucional para ajustar las dimensiones.

Con las imágenes de 100 píxeles, se aumentó el tamaño del lote a 128 y el conjunto de imágenes de entrenamiento a 2000, además, se pudo ajustar los filtros de las capas para realizar diversas pruebas.

**Figura 23.** Generación de imagen 100x100 tras 500 épocas, utilizando 64 filtros en su primera capa y siguiendo la secuencia de selección de filtros mencionada más adelante.

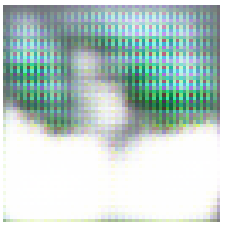


**Figura 24.** Generación de imagen 64x64 tras 500 épocas.

Como es apreciable en la figura 15 y en la 16, la generación de imágenes para dichos tamaños es considerablemente peor a las anteriores mostradas, el discriminador en estos casos estaba dominando al generador. Mientras que las pérdidas del generador aumentaban rápidamente, las del discriminador se reducían, arrojando imágenes poco precisas. Mediante ajustes en la capa *dropout* y en el ratio de aprendizaje del discriminador, se logró equilibrar las pérdidas de las dos redes, como se puede apreciar en la tabla 4.

| Ratio de descarte | Ratio de aprendizaje del discriminador | Pérdidas del discriminador tras 50 epoch | Pérdidas del generador tras 50 epoch |
|---|---|---|---|
| 0.4 | 0.0001 | -1.856 | 50.8238 |
| 0.4 | 0.00001 | 0.7114 | 0.7248 |
| 0.6 | 0.0001 | 0.2858 | 2.3335 |
| 0.6 | 0.00001 | 0.7074 | 0.7234 |

**Tabla 4.** Comparativa de las pérdidas de los modelos al manipular el ratio de aprendizaje del discriminador y el ratio de descarte de la capa dropout, se utilizaron imágenes de 64 píxeles, un tamaño del lote de 32, pues daba mejores resultados que otros tamaños y un conjunto de filtros tal que las capas del discriminador estaban formadas por 64 filtros en la primera capa, 128 y 128 en las sucesivas y el generador usaba 128, 256 y 512.

Como se puede observar los valores de 0.6 para el ratio de descarte y 0.00001 para el ratio de aprendizaje resultan en las menores pérdidas. Con dichos valores fijados, se procedió a estudiar qué combinación de filtros daba mejores resultados, siguiendo la secuencia $f, f * 2, f * 2, f * 2, f * 4, f * 8$, siendo $f$ el número de filtros iniciales. Se probaron como valores para $f$ 16, 32, 64 y 128, como se ilustra en la tabla 5.

| Filtros | Pérdidas del discriminador | Pérdidas del generador |
|---|---|---|
| 16 | 0.6823 | 0.7744 |
| 32 | 0.6929 | 0.7402 |
| 64 | 0.6873 | 0.7451 |
| 128 | 0.6954 | 0.7324 |

**Tabla 5.** Comparativa del impacto del número de filtros sobre las pérdidas de los modelos, utilizando los parámetros mencionados en la tabla 4.

No se observan grandes variaciones en cuanto a las pérdidas, aunque el tiempo de cada época si se ve enormemente afectado cuantos más filtros tenga la red. Tras los dos estudios, las imágenes generadas se pueden ver en la figura 17.

**Figura 25.** Imágenes correspondientes a la época 500, realizadas utilizando 16, 32 y 128 filtros respectivamente.

# Capítulo 4 Conclusiones y líneas futuras

Este trabajo de fin de grado se ha centrado en las siguientes problemáticas, por un lado predecir la rotación de las articulaciones de la mano a partir de una imagen RGB, explorar el espacio latente del autoencoder mediante interpolaciones lineales y una alternativa a este para encontrar la representación codificada del espacio latente de las configuraciones de la mano.

En cuanto a la red GAN, no ha sido capaz de generar imágenes de manos similares a las originales, el entrenamiento de una red GAN es un proceso complicado que necesita de un ajuste cuidadoso para que las dos redes puedan llegar a aprender. La interpolación lineal entre las imágenes codificadas no produjo resultados satisfactorios, ya que la interpolación arroja imágenes que corresponden a la interpolación de los píxeles. En un futuro podría realizarse una implementación de un autoencoder variacional, para comprobar si su espacio latente de soluciones tiene un mayor significado que el del autoencoder convencional utilizado.

Para la estimación de las rotaciones y al igual que con la red GAN, poder contar con un mayor *dataset* podría mejorar la precisión del perceptrón de un 15% a un valor superior y más aceptable.

Aunque la estimación de la pose resulta crucial en el proceso de recuperar la malla 3D de la mano, también resulta conveniente calcular la forma de esta y alinearla con la mano del usuario en el espacio de la imagen. Es por ello, que como líneas futuras se podrían explorar técnicas para estimar la silueta de la mano y métodos para alinear la mano recuperada con la del usuario utilizando por ejemplo representación diferenciable [15, 16].

La implementación de las redes neuronales para este trabajo, así como el programa de generación sintética puede ser accedido a través del siguiente enlace: https://github.com/alu0101030531/TFG

# Capítulo 5 Summary and Conclusions

In this final degree project we have focused on two different problems, predict the hand joints rotation from a RGB image and augment the hand images dataset.

The GAN network wasn't able to generate decent hand images, in order to get better results we have to try training with a larger dataset, one the size of the state of art. This could not be extended due to physics limitations. The linear interpolation couldn't produce satisfying results, the resulting images correspond to the linear interpolation of their pixels. We could implement a variational autoencoder to test if its latent space is more significant than the conventional autoencoder.

Equivalents happen with the hand joints regression, we could try to face the ambiguity inherent in the RGB hand images increasing the dataset and it would significantly improve the accuracy of the model higher than 15%.

Although hand pose estimation is crucial in the process of 3D hand mesh reconstruction, also it's required to calculate the hand silhouette and the reconstructed hand should align with the user hand in the image space. Therefore, as future researches, we should explore different techniques for hand silhouette estimation and also methods for hand alignment using for example differentiable rendering [15, 16].

The neural networks implementation, as well as the synthetic data generation program can be accessed through the following link: https://github.com/alu0101030531/TFG

# Capítulo 6 Presupuesto

El objeto de este presupuesto constituye las implementaciones y entrenamientos de los distintos modelos, además de los distintos programas y métodos para la generación de conjuntos de datos. En él sólamente se hará constar la partida de mano de obra, ya que no se ha utilizado elementos materiales, no obstante se añade una partida que implica el desgaste natural del uso del equipo así como los gastos derivados de su utilización.

| Tipos | Descripción | Importe |
|---|---|---|
| Mano de obra | 250 horas | 2750€ |
| Utilización de equipos | 325 horas | 650€ |
| | Total presupuesto: | 3400€ |

**Tabla 1.** Resumen de tipos

# Bibliografía

[1] Jameel Malik, Ibrahim Abdelaziz, Ahmed Elhayek, Soshi Shimada, Sk Aziz Ali, Vladislav Golyanik, Christian Theobalt y Didier Stricker. HandVoxNet: Deep voxelbased network for 3D hand shape and pose estimation from a single depth map. En el acto de la conferencia IEEE en Visión por computador y Reconocimiento de Patrones (CVPR), páginas 7113–7122, 2020.

[2] Sameh Khamis, Jonathan Taylor, Jamie Shotton, Cem Keskin, Shahram Izadi y Andrew Fitzgibbon. Learning an efficient model of hand shape variation from depth images. En el acto de la conferencia IEEE en Visión por computador y Reconocimiento de Patrones (CVPR), páginas 2540–2548, 2015.

[3] Linpu Fang, Xingyan Liu, Li Liu, Hang Xu y Wenxiong Kang. JGR-P2O: Joint graph reasoning based pixel-to-offset prediction network for 3D hand pose estimation from a single depth image. En ECCV, páginas 120–137, 2020.

[4] Gyeongsik Moon, Ju Yong Chang y Kyoung Mu Lee. V2V-PoseNet: Voxel-to-voxel prediction network for accurate 3D hand and human pose estimation from a single depth map. In CVPR, páginas 5079–5088, 2018.

[5] Franziska Mueller, Micah Davis, Florian Bernard, Oleksandr Sotnychenko, Mickeal Verschoor, Miguel A Otaduy, Dan Casas y Christian Theobalt. Real-time pose and shape reconstruction of two interacting hands with a single depth camera. ACM Transactions on Graphics (SIGGRAPH), 38(4):49:1–49:13, 2019.

[6] Chengde Wan, Thomas Probst, Luc Van Gool, and Angela Yao. Dual grid net: Hand mesh vertex regression from single depth maps. En el acto de la Conferencia Europea sobre Visión por Computador (ECCV), páginas 442–459, 2020.

[7] Jiayi Wang, Franziska Mueller, Florian Bernard, Suzanne Sorli, Oleksandr Sotnychenko, Neng Qian, Miguel A. Otaduy, Dan Casas y Christian Theobalt. RGB2Hands: real-time tracking of 3D hand interactions from monocular RGB video. ACM Transactions on Graphics (SIGGRAPH Asia), 39(6):218:1–218:16, 2020.

[8] Xingyu Chen, Yufeng Liu, Chongyang Ma, Jianlong Chang, Huayan Wang, Tian Chen, Xiaoyan Guo, Pengfei Wan y Wen Zheng. Camera-space hand mesh recovery via semantic aggregation and adaptive 2D-1D registration. En el acto de la conferencia IEEE en Visión por computador y Reconocimiento de Patrones (CVPR), páginas 13274–13283, 2021.

[9] Zhipeng Fan, Jun Liu y Yao Wang. Adaptive computationally efficient network for monocular 3D hand pose estimation. En el acto de la Conferencia Europea sobre Visión por Computador (ECCV), páginas 127–144, 2020.

[10] Neng Qian, Jiayi Wang, Franziska Mueller, Florian Bernard, Vladislav Golyanik y Christian Theobalt. HTML: A parametric hand texture model for 3D hand reconstruction and personalization. En el acto de la Conferencia Europea sobre Visión por Computador (ECCV), páginas 54–71, 2020.

[11] David Joseph Tan, Thomas Cashman, Jonathan Taylor, Andrew Fitzgibbon, Daniel Tarlow, Sameh Khamis, Shahram Izadi y Jamie Shotton. Fits like a glove: Rapid and reliable hand shape personalization. En CVPR, páginas 5610– 5619, 2016.

[12] Jonathan Taylor, Richard Stebbing, Varun Ramakrishna, Cem Keskin, Jamie Shotton, Shahram Izadi, Aaron Hertzmann y Andrew Fitzgibbon. User-specific hand modeling from monocular depth sequences. En CVPR, páginas 644–651, 2014.

[13] Martin de La Gorce, Nikos Paragios y David J Fleet. Model-based hand tracking with texture, shading and self-occlusions. En CVPR. 2008.

[14] Javier Romero, Dimitrios Tzionas y Michael J. Black. Embodied hands: Modeling and capturing hands and bodies together. ACM Transactions on Graphics (SIGGRAPH Asia), 36(6):245:1–245:17, 2017.

[15] Seungryul Baek, Kwang In Kim y Tae-Kyun Kim. Pushing the envelope for RGB-based dense 3D hand pose estimation via neural rendering. En CVPR, páginas 1067–1076, 2019.

[16] Xiong Zhang, Qiang Li, Hong Mo, Wenbo Zhang y Wen Zheng. End-to-end hand mesh recovery from a monocular RGB image. En ICCV, páginas 2354–2364, 2019.

[17] John Yang, Hyung Jin Chang, Seungeui Lee y Nojun Kwak. SeqHAND: RGB-sequence-based 3D hand pose and shape estimation. En el acto de la Conferencia Europea sobre Visión por Computador (ECCV), páginas 122-139, 2020.

[18] Yuan, S., Ye, Q., Stenger, B., Jain, S., Kim, T.K.: Bighand2. 2m benchmark: Hand pose dataset and state of the art analysis pp. 4866–4874, 2017.

[19] Shangchen Han, Beibei Liu, Randi Cabezas, Christopher D Twigg, Peizhao Zhang, Jeff Petkau, Tsz-Ho Yu, Chun-Jung Tai, Muzaffer Akbay, Zheng Wang, et al. MEgATrack: monochrome egocentric articulated hand-tracking for virtual reality. ACM Transactions on Graphics (SIGGRAPH), 39(4):87:1–87:13, 2020.

[20] Lixin Yang, Jiasen Li, Wenqiang Xu, Yiqun Diao y Cewu Lu. BiHand: Recovering hand mesh with multi-stage bisected hourglass networks. En BMVC, 2020.

[21] Yuxiao Zhou, Marc Habermann, Weipeng Xu, Ikhsanul Habibie, Christian Theobalt y Feng Xu. Monocular real-time hand shape and motion capture using multi-modal data. arXiv preprint arXiv:2003.09572, 2020.

[22] Shile Li y Dongheui Lee. Point-to-pose voting based hand pose estimation using residual permutation equivariant layer. En el acto de la conferencia IEEE en Visión por computador y Reconocimiento de Patrones (CVPR), páginas 11927–11936, 2019.

[23] L. Ge, Y. Cai, J. Weng y J. Yuan. Hand pointnet: 3d hand pose estimation using point sets. En el acto de la conferencia IEEE en Visión por computador y Reconocimiento de Patrones (CVPR), páginas 8417–8426, 2018.

[24] L. Ge, Z. Ren y J. Yuan. Point-to-point regression pointnet for 3d hand pose estimation. En ECCV, Springer, 1, 2018.

[25] Javier Romero, Dimitrios Tzionas y Michael J. Black. Embodied hands: Modeling and capturing hands and bodies together. ACM Transactions on Graphics (SIGGRAPH Asia), 36(6):245:1–245:17, 2017.

[26] Makehuman, "*Makehuman*", makehumancommunity.org, http://www.makehumancommunity.org/ (accedido el 13, junio. 2022)

[27] Dor Bank, Noam Koenigstein y Raja Giryes. Autoencoders. arXiv:2003.05991v2, 2021.

[28] Francois Chollet, "Introduction to generative adversarial networks", en *Deep learning with Python*, F. Chollet, Manning Publication, 2018.