



**FACULTAD
DE INGENIERIA**

Universidad de Buenos Aires

75.99 - Trabajo Profesional
Febrero 2019

Two-Face Inpainting mediante GANs

Alumno: Gaston Snaider

Padrón: 96415

E-Mail: gnsnaider@gmail.com

Tutor: Lic. Luis Argerich

1 Introducción

En este proyecto se desarrolló y entrenó un modelo de reconstrucción de imágenes faciales con una región faltante. El objetivo del modelo es poder regenerar dicha región de forma realista, un problema conocido como image inpainting [1, 2]. Por otra parte, en este trabajo se introdujo la idea de que el modelo reciba también una segunda imagen de la misma persona, de forma tal de tener una imagen referencia al momento de completar la región faltante.

La implementación consiste de un modelo basado en Generative Adversarial Networks (GANs) [3], el cual utiliza redes neuronales convolucionales [4] para el procesamiento de las imágenes. Se implementó y entrenó un modelo generador (generator) y dos modelos discriminativos (discriminators), uno local y otro global [2]. Para el análisis de la imagen de referencia, se utilizó el modelo pre-entrenado FaceNet [5], el cual puede detectar la similitud entre imágenes faciales.

A su vez, en este proyecto se desarrolló una aplicación web que permite interactuar con el modelo. La aplicación consiste de un servidor frontend a través del cual los usuarios pueden cargar imágenes y utilizarlas para probar el modelo, y un servidor backend que contiene al modelo entrenado. El servidor backend provee una API REST mediante la cual el servidor frontend puede invocar al modelo con las imágenes provistas por los usuarios.

Modelo de inpainting: <https://github.com/gsnaider/two-face-inpainting>

Aplicación Web: <https://github.com/gsnaider/two-face-web-client>

2 Introducción teórica

Las Generative Adversarial Networks (GANs) fueron introducidas por Ian Goodfellow y un grupo de investigadores de la Universidad de Montreal en el 2014 [3] como un modelo generativo de datos. Para poder comprender el funcionamiento de las GANs, es conveniente comenzar con una distinción entre algoritmos discriminativos y algoritmos generativos [10].

2.1 Algoritmos discriminativos

Los algoritmos discriminativos son algoritmos de aprendizaje supervisado, cuyo principal objetivo es la clasificación de datos. Es decir, en base a los features de un dato, se busca predecir a qué categoría pertenece dicho dato. Por ejemplo, en base a todos los píxeles de una imagen, se puede querer predecir qué objeto hay en la imagen (asumiendo que la imagen contiene un solo objeto). O en base al texto de un email, se podría predecir si el email es spam o no.

En términos más generales, en base a los features X de un dato, se busca determinar cuál es la categoría Y a la cual pertenece el dato. Es decir, se busca encontrar la probabilidad $P(Y|X)$, o lo que es lo mismo, la probabilidad de la categoría Y dados los features X.

Algunos ejemplos de estos algoritmos son Logistic Regression, Support Vector Machines, y Redes Neuronales, entre otros.

2.2 Algoritmos generativos

A diferencia de los algoritmos discriminativos, los algoritmos generativos tienen como objetivo determinar la distribución conjunta $P(X, Y)$. Usando las reglas de Bayes, es posible calcular $P(Y|X)$, para usar al modelo como discriminador. Sin embargo, también es posible utilizar esta distribución para generar ejemplos de los features X que pertenezcan a la distribución del set de datos de entrenamiento. Este es uno de los principales usos de los algoritmos generativos. Por ejemplo, a partir de un set de imágenes de dígitos manuscritos como MNIST [7], podría utilizarse un algoritmo generativo para generar nuevas imágenes similares a las del set de datos, es decir, imágenes con una alta probabilidad de pertenecer a dicho set de datos.

Algunos ejemplos de algoritmos generativos son Pixel RNN/CNN, Variational Autoencoders (VAEs), y Generative Adversarial Networks (GANs) [11]. En este trabajo nos enfocaremos en este último algoritmo.

2.3 Generative Adversarial Networks (GANs)

Como se mencionó en la sección anterior, las GANs son un ejemplo de un modelo generativo. Es decir, pueden ser utilizadas para generar datos con una alta probabilidad de pertenecer a un set de datos particular. Este modelo se compone de dos redes neuronales, una red generativa, y una red discriminativa. La red generativa recibe como entrada un conjunto de datos aleatorios, y su objetivo es generar un dato “realista” a partir de dicha entrada. Por otro lado, la red discriminativa puede recibir como entrada tanto datos reales como datos generados por la red generativa, y su objetivo es poder determinar si un dato es real o generado por la otra red. Para entrenar el modelo, se deben entrenar a las dos redes en simultáneo en base a sus respectivos objetivos. Una vez entrenado el modelo, se tendrá una red generativa capaz de generar datos similares a los datos reales a partir de una entrada aleatoria. En la Figura 1 se ilustra este proceso.

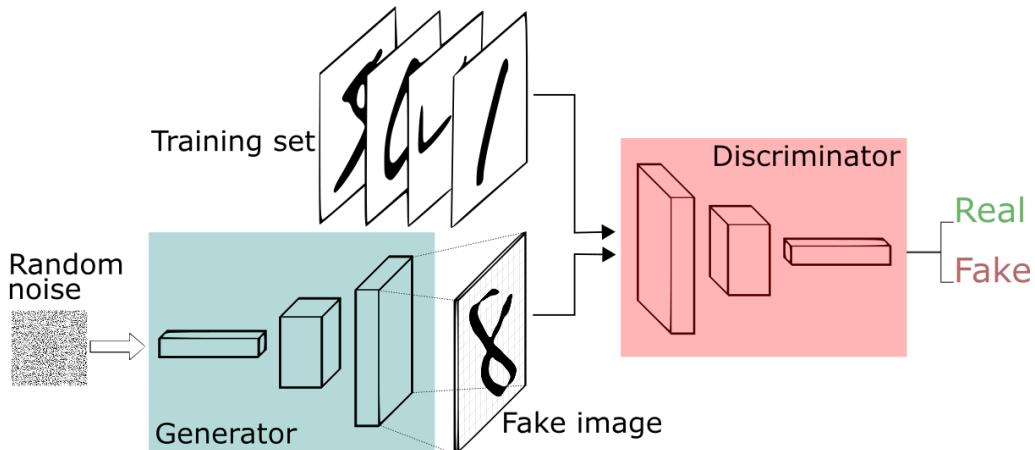


Figura 1: Arquitectura de una GAN. [16]

Hasta ahora solo se mencionó a las GANs como modelos capaces de generar datos similares a los de un set de datos a partir de una entrada aleatoria. Sin embargo, las GANs permiten ir un paso más allá. Podemos no solo generar datos a partir de una entrada aleatoria, sino también condicionar a las GANs a generar datos pertenecientes a una determinada clase del set de datos. Basta con incluir esta clase en los parámetros de entrada del generador y del discriminador. De esta forma el generador aprenderá a producir un dato correspondiente a la clase que reciba en la entrada. Esto abre aún más posibilidades para las GANs. Volviendo al ejemplo de dígitos manuscritos mencionado anteriormente, podría condicionarse a la GAN a generar imágenes de determinado dígito, pasando como entrada la clase (el dígito) que se desea generar.

Si bien existen otros modelos generativos, como Variational Autoencoders (VAEs), las imágenes resultantes en general son menos nítidas que las obtenidas mediante GANs [12, 13]. Esto puede observarse claramente en la Figura 2. Es por esto

que para este proyecto utilizaremos modelos de GANs para la aplicación de inpainting sobre imágenes.



Figura 2: Imágenes generadas por un modelo VAE (primer fila) y por un modelo GAN (segunda fila). [13]

2.4 Image inpainting mediante GANs

Image inpainting (o regeneración de imágenes), consiste en completar partes faltantes dentro de una imagen. Uno de los grandes desafíos de image inpainting es poder regenerar las partes faltantes de forma realista, y que mantengan correlación con el resto de la imagen. Gracias al reciente avance de las redes neuronales convolucionales (CNN) para el procesamiento de imágenes y a las GANs [4], han surgido varios trabajos que aplican dichos modelos para image inpainting [1, 2, 13, 14, 15]. Un ejemplo de esto puede verse en la Figura 3.



Figura 3: Ejemplo de image inpainting mediante GANs [13]. En la primer columna se tiene una imagen real, en la segunda columna se tiene la misma imagen con una región faltante, y en la última columna se muestra a la imagen con dicha región regenerada mediante GANs.

3 Modelo de inpainting

3.2 Arquitectura

3.2.1 Modelo completo

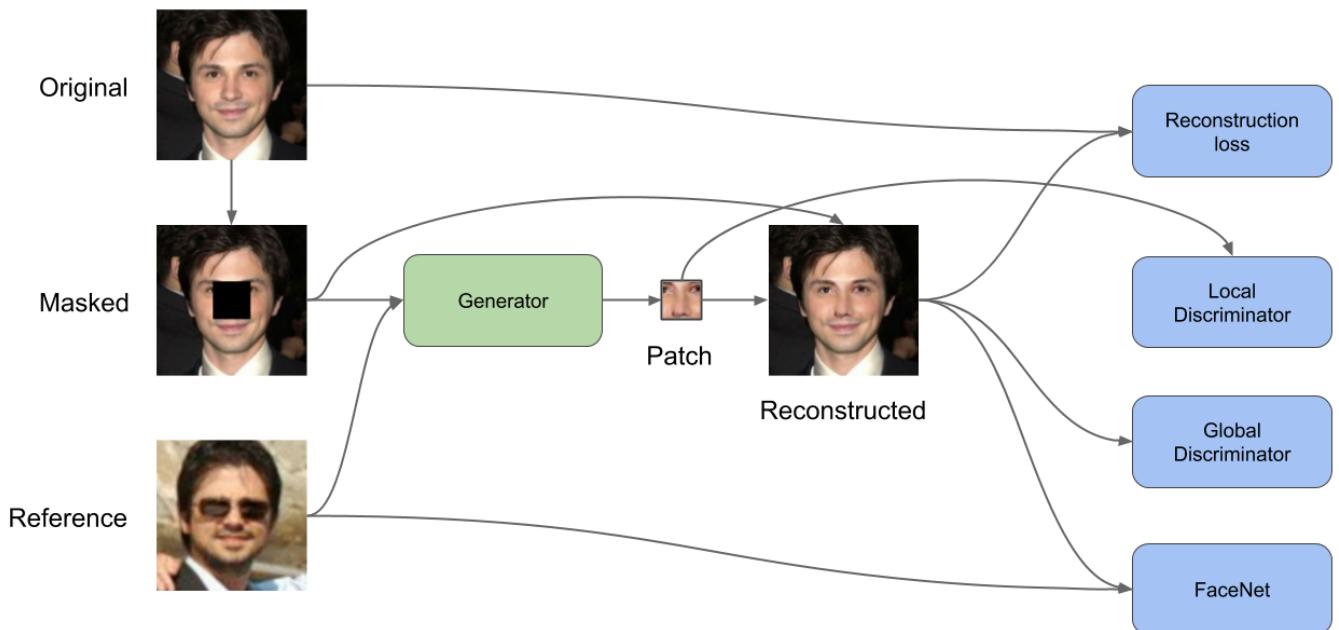


Figura 4: Arquitectura completa del modelo de inpainting

La arquitectura general del modelo implementado en el presente trabajo consiste de un Generator, dos Discriminators, y el modelo pre-entrenado FaceNet, como se ilustra en la Figura 4.

Para entrenar el modelo, se parte de la imagen de un individuo, y se le aplica una máscara a ser rellenada por el Generator. Además, se debe tener una imagen de referencia del mismo individuo. El Generator recibe tanto la imagen enmascarada como la de referencia, y genera la región faltante en la primera imagen. Con esta región generada, se reconstruye la primera imagen.

Luego entran en juego los distintas funciones de costo (losses) que se aplican sobre el Generator. El primero es un Reconstruction loss, que mide la distancia entre la imagen reconstruida y la original.

El segundo loss corresponde al Local Discriminator [2]. Este modelo determina si las regiones generadas por el Generator son reales o falsas, por lo que el Generator debe aprender a generar regiones cada vez más reales.

El tercer loss está dado por el Global Discriminator [2]. Este modelo es similar al Local Discriminator, pero en vez de analizar sólo la región generada, el Global Discriminator analiza toda la imagen reconstruida para ver si es real o falsa.

Por último, se tiene un loss correspondiente al modelo de FaceNet. Este es un modelo pre-entrenado, que a partir de la imagen de una persona, el modelo genera un vector representando a dicha persona. Dos imágenes de la misma persona deberían dar vectores más cercanos, mientras que las imágenes de personas distintas dan vectores más alejados entre sí. Por lo tanto, el último loss corresponde a la distancia entre los vectores de FaceNet de la imagen reconstruida y la de referencia. De esta forma, se busca que el Generator complete las imágenes teniendo en cuenta la similitud con la imagen de referencia.

Entre el Generator, ambos Discriminators, y FaceNet, el modelo completo tiene aproximadamente 50 millones de parámetros, de los cuales 20 millones son entrenables. Esto da una idea de la complejidad que hay detrás de esta arquitectura, y el desafío que implica llevar a cabo el entrenamiento de este modelo.

A continuación se describen en detalle las arquitecturas internas tanto del Generator como de ambos Discriminatos.

3.2.2 Generator

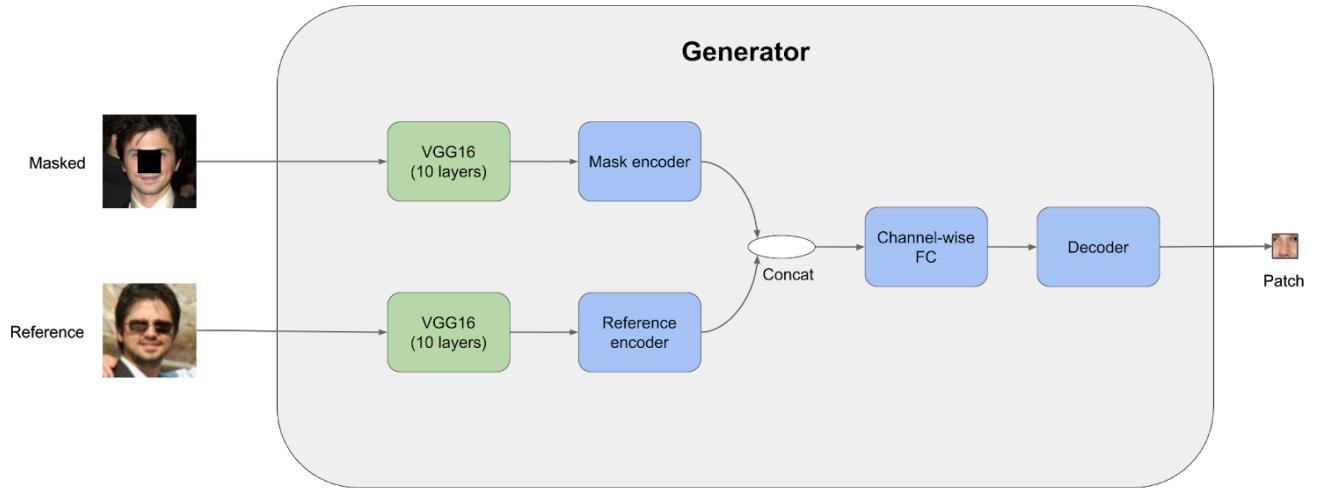


Figura 5: Arquitectura del Generator

La arquitectura del Generator puede verse en la Figura 5. Como se mencionó, el Generator recibe tanto la imagen enmascarada como la de referencia, y genera la región faltante en la primer imagen. Inicialmente, cada imagen es procesada por las 10 primeras capas del modelo pre-entrenado VGG16. Esto permite analizar los rasgos más concretos de las imágenes, como bordes, texturas, polígonos, etc. Luego de esto, cada imagen pasa por un encoder, los cuales consisten de dos capas convolucionales con batch normalization, y una capa final de max pooling. De esta forma, se obtiene un vector representativo de cada imagen. Estos dos vectores luego se concatenan, y se procesan a través de una channel-wise fully-connected layer [1]. Esta capa es básicamente una fully-connected layer, pero solo conecta cada activation map, lo cual evita una explosión de parámetros. Finalmente, se utiliza un decoder para reconstruir la región faltante en la primer imagen. El decoder consiste de 5 capas convolucionales transpuestas, 2 capas de batch normalization, y 2 capas de upsampling.

Tanto los encoders como el decoder y la channel-wise fully-connected layer fueron entrenados de cero (representados en azul en el gráfico), mientras que el modelo VGG16 fue utilizado con sus parámetros pre-entrenados (representado en verde en el gráfico).

3.2.3 Discriminators

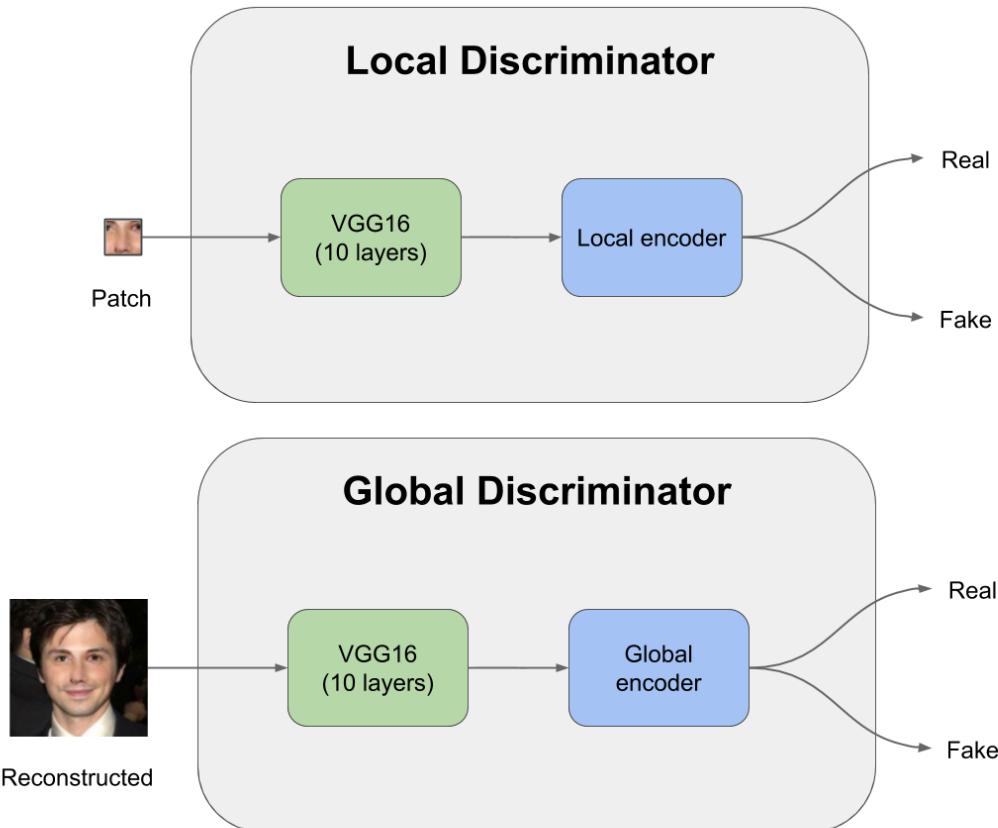


Figura 6: Arquitectura del Local Discriminator y Global Discriminator

En la Figura 6 se puede apreciar la arquitectura del Local Discriminator y Global Discriminator. Ambos Discriminators tienen una arquitectura similar, ya que su objetivo en esencia es el mismo: dada una imagen, determinar si esta es real o falsa (es decir, determinar si fue generada por el Generator o no). La única diferencia es que el Local Discriminator debe analizar las regiones que son generadas por el Generator, mientras que el Global Discriminator analiza la imagen completa.

Al igual que en el Generator, ambos modelos comienzan con las primeras 10 capas del modelo pre-entrenado VGG16, lo cual permite realizar un procesamiento inicial de la imagen. Luego, en cada modelo sigue un Encoder que continúa analizando el resultado de VGG16. En el Local Discriminator, el encoder se compone de 2 capas convolucionales con batch normalization, una capa de max pooling, y 2 capas fully-connected. El encoder del Global Discriminator es similar, pero tiene una capa convolucional y una capa de max pooling extras, ya que se requiere analizar una imagen de mayor tamaño. Finalmente, en ambos encoders la última capa es una fully-connected de un solo output, el cual representa si la imagen es real o falsa.

Así como en el Generator, ambos encoders fueron entrenados de cero, mientras que el modelo de VGG16 fue utilizado con sus parámetros pre-entrenados.

3.3 Proceso de entrenamiento

Para el proceso de entrenamiento, al igual que en [2], se utilizó la metodología de Curriculum Learning [6]. Esto significa que las distintas funciones de loss fueron activándose gradualmente a lo largo del proceso de entrenamiento. Inicialmente, se comenzó entrenando al modelo solo teniendo en cuenta el Reconstruction Loss. De esta forma, se buscó que el Generator comience aprendiendo la estructura general de la región a regenerar. Los resultados obtenidos con este loss son un tanto borrosos, ya que el Generator busca minimizar el Reconstruction Loss lo más posible sin comprometerse a generar una imagen específica. Una vez entrenado el modelo con el Reconstruction Loss, se encendió el loss correspondiente al Local Discriminator. Esto obligó al Generator a comenzar a generar regiones más definidas y realistas, aunque estas aún no necesariamente se corresponden al 100% con la imagen completa. Luego del entrenamiento con el Local Discriminator, se activó el loss del Global Discriminator. De esta forma, el Generator comenzó a generar regiones que se combinan mejor con la imagen completa. Un detalle en los pasos de activación de ambos Discriminators es que antes de activar el Adversarial Loss, primero se activó por un lapso breve de tiempo cada Discriminator, sin afectar aún al Generator. La idea de esto es que si los Discriminators comienzan a afectar al Generator sin haber sido entrenados previamente, estos podrían arruinar lo ya aprendido por el Generator. Por último, se activó el loss correspondiente a FaceNet, de forma tal de incluir en el entrenamiento las imágenes de referencia.

Todo el proceso de entrenamiento del modelo final tardó 26 horas utilizando una placa gráfica NVIDIA Tesla P100 en Google Cloud Platform.

3.4 Tecnologías

Para la implementación del modelo y el pipeline de entrenamiento se utilizó Python 3.5 junto con TensorFlow 1.12. Se utilizó además la API de Keras provista por TensorFlow para la definición de los modelos. Otras de las librerías utilizadas para el desarrollo de experimentos y utilitarios fueron Jupyter Notebook, NumPy, Matplotlib, entre otras.

Para el entrenamiento del modelo, se utilizó el servicio de ML Engine dentro de Google Cloud Platform, el cual permite ejecutar procesos de entrenamiento en la nube utilizando GPUs. En este trabajo se utilizó principalmente una placa gráfica NVIDIA Tesla P100 para llevar a cabo el entrenamiento, aunque algunas pruebas también fueron realizadas con una NVIDIA Tesla K80.

4 Aplicación web

4.1 Arquitectura

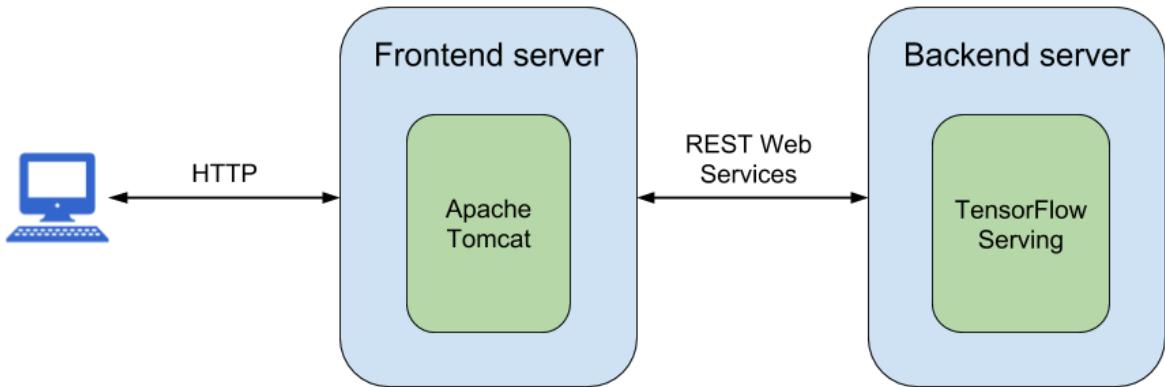


Figura 7: Arquitectura general de la Aplicación Web.

La aplicación web está compuesta de dos servidores: un servidor frontend y un servidor backend. Esto se ilustra en la Figura 7. El servidor frontend contiene la aplicación web a través de la cual el usuario puede interactuar con el modelo. El usuario puede conectarse con el servidor frontend mediante su navegador a través del protocolo HTTP. El servidor backend contiene al modelo entrenado, y provee una API REST para poder invocar al modelo de forma remota desde el servidor frontend.

4.2 Frontend Server

El servidor frontend consiste de una aplicación web Java que se ejecuta sobre un servidor Apache Tomcat. A través de esta aplicación, el usuario puede cargar las imágenes a usar en el modelo, seleccionar las regiones donde se encuentran las caras, invocar al modelo usando dichas imágenes, y obtener los resultados.

La implementación del servidor frontend está hecha en Java EE 1.8, y la aplicación se ejecuta sobre un servidor Apache Tomcat. Tomcat nos permite por un lado deployar de forma sencilla la aplicación, y además nos provee varias funcionalidades como manejo de requests y de sesiones, multithreading, etc. Se utilizaron a su vez varias librerías de Java para las distintas funcionalidades requeridas por la aplicación: Java Server Faces y PrimeFaces para los componentes visuales, Jersey para el llamado a la API REST del servidor backend, Jackson para la conversión a formato JSON de las

imágenes, y Maven para el manejo de dependencias y ciclo de deployment de la aplicación.

La aplicación se implementó siguiendo el patrón de diseño MVC (Model View Controller). La arquitectura interna de la misma puede verse en la Figura 8.

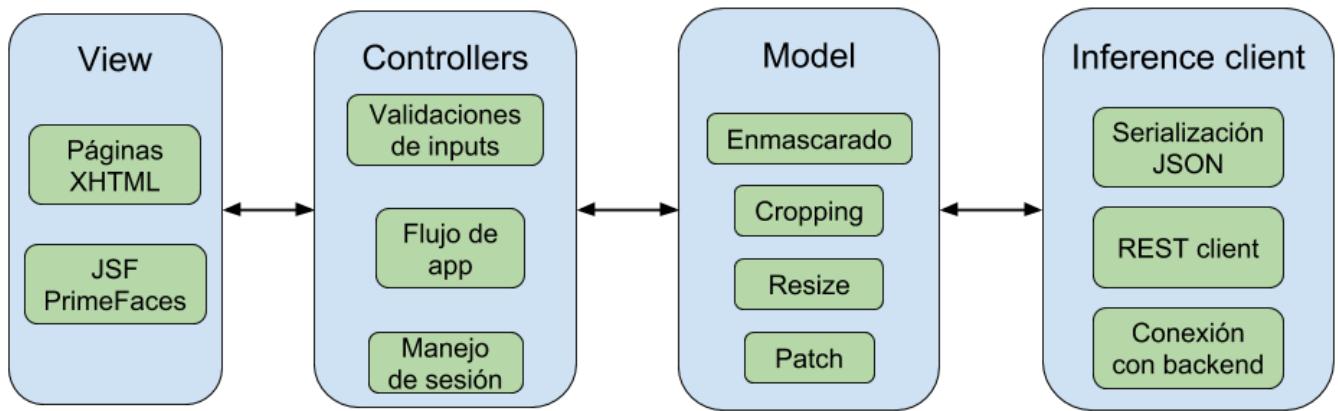


Figura 8: Arquitectura interna de la aplicación web.

El componente de la vista (View) consiste de las páginas XHTML a través de las cuales el usuario puede interactuar con la aplicación. En ellas se utilizan componentes web de PrimeFaces que permiten, entre otras cosas, cargar imágenes, visualizarlas, seleccionar regiones en las mismas, etc.

Por cada vista (es decir, por cada página XHTML) se tiene un Controller que administra dicha vista. Los controllers se encargan de recibir los inputs de los usuarios, aplicar validaciones sobre los mismos, administrar el flujo de la aplicación (es decir, determinar cuál es la siguiente pantalla en cada paso), y manejar la sesión del usuario. El hecho de administrar una sesión permite mantener las imágenes que carga cada usuario durante los distintos pasos de la aplicación.

El modelo de la aplicación contiene toda la lógica para el procesamiento de las imágenes. A través del modelo se puede enmascarar una imagen (es decir, remover una sección para luego ser rellenada), recortar las imágenes, cambiar su tamaño, y llenarlas con las regiones generadas por el modelo.

Por último, se tiene un cliente de inferencia (Inference Cliente), cuya principal función es la de comunicarse con el servidor de backend. En este componente, se serializan las imágenes a formato JSON, y se utiliza un cliente de servicios REST para invocar al modelo en el servidor de backend. A su vez, en este componente se maneja la respuesta del backend, se realiza un control de errores, se deserializa la respuesta, y se retorna la imagen de la región regenerada al modelo.

A continuación se describe el funcionamiento de la aplicación desde el punto de vista del usuario, con capturas de pantalla ilustrando las distintas funcionalidades de la aplicación.

Two-Face Inpainting

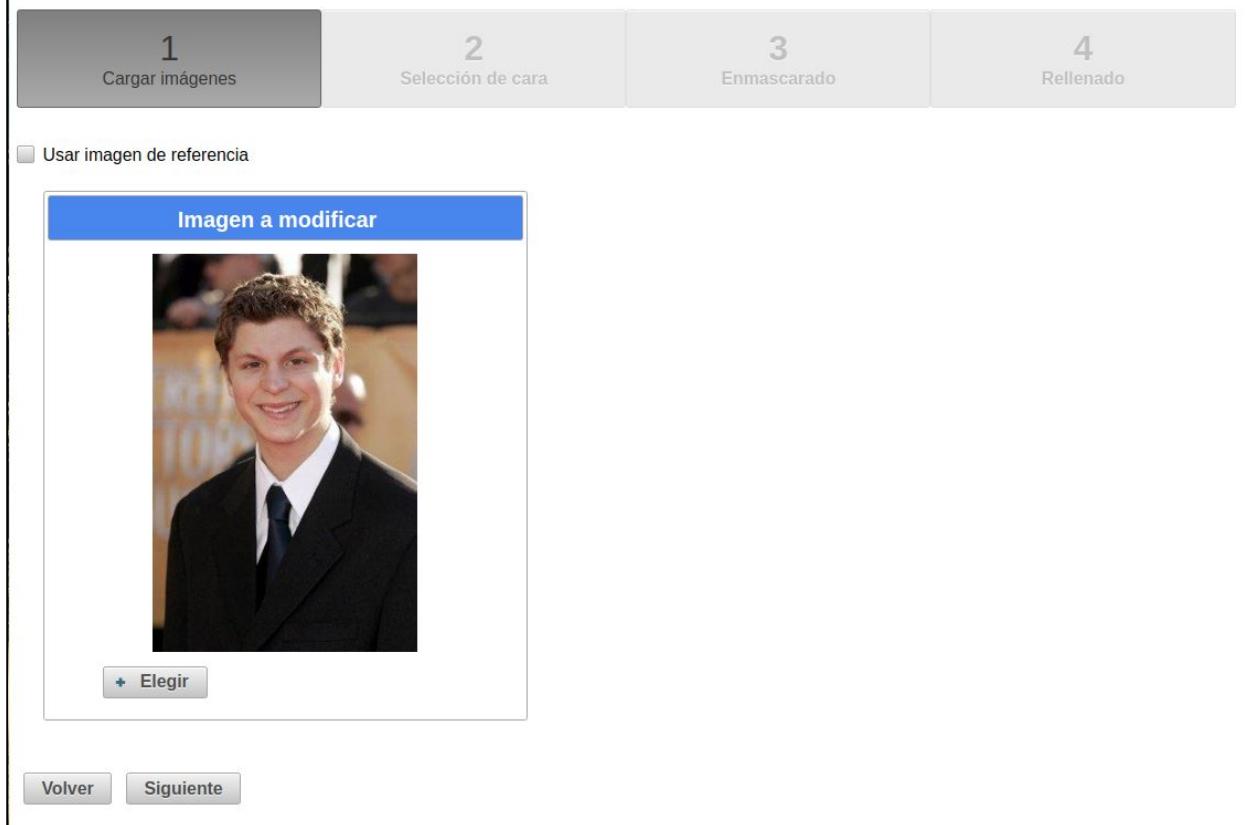


Figura 9: Primer paso de la aplicación: "Carga de imágenes". Sin imagen de referencia.

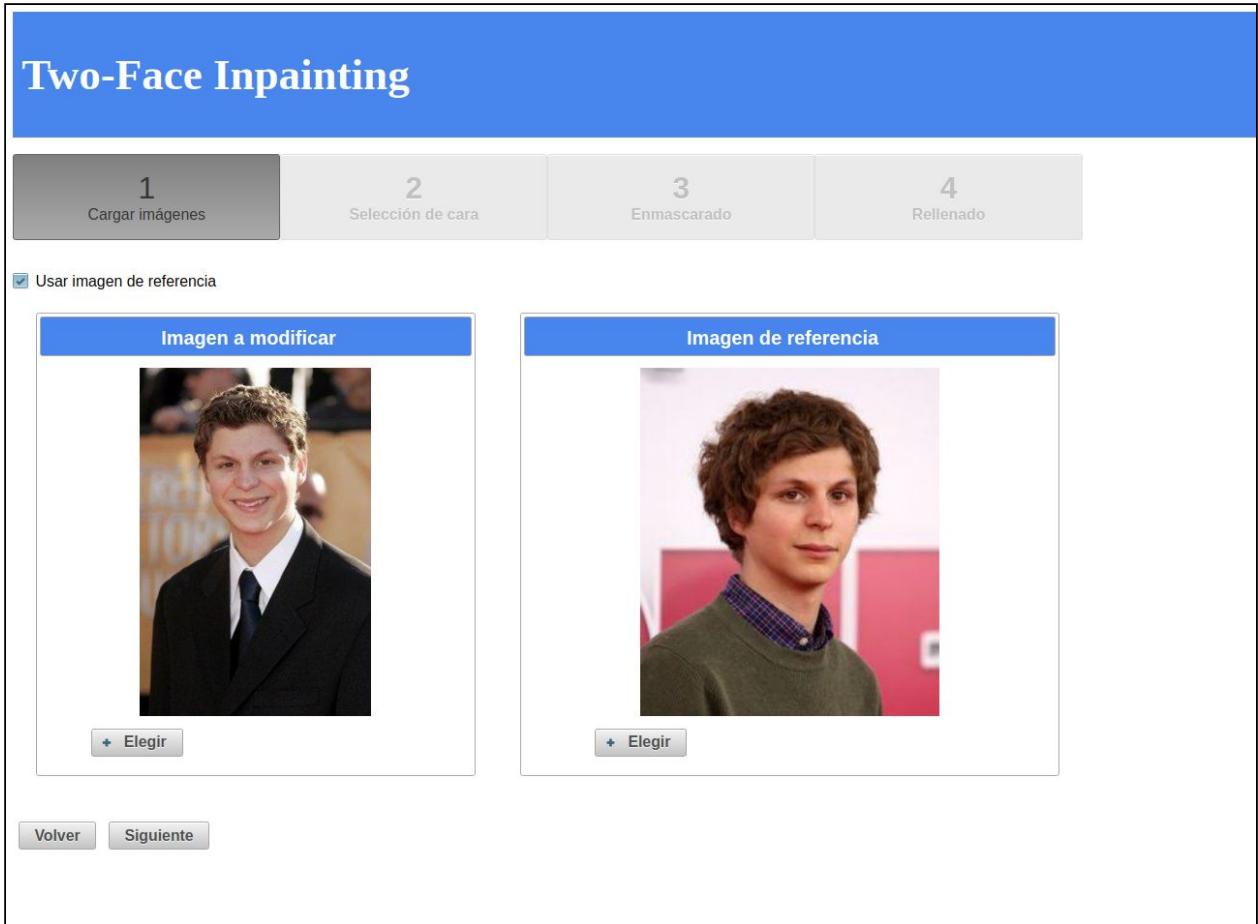


Figura 10: Primer paso de la aplicación: "Carga de imágenes". Con imagen de referencia.

En la primer pantalla de la aplicación, el usuario puede elegir las imágenes desde su computadora para probar el modelo. Esta pantalla puede verse en las Figuras 9 y 10. Se tienen dos opciones para cargar las imágenes: con o sin imagen de referencia. Esto permite al usuario usar el modelo con dos imágenes de la misma persona, o en el otro caso ver cómo el modelo aún sin imagen de referencia es capaz de reconstruir la región faltante. En el caso de usar una imagen de referencia, el usuario debe cargar dos imágenes (la que se regenerará por el modelo, y la referencia). Caso contrario, solo debe cargar la imagen que será regenerada. En las siguientes pantallas mostraremos el flujo utilizando la imagen de referencia. El flujo sin la imagen de referencia es análogo.

Two-Face Inpainting

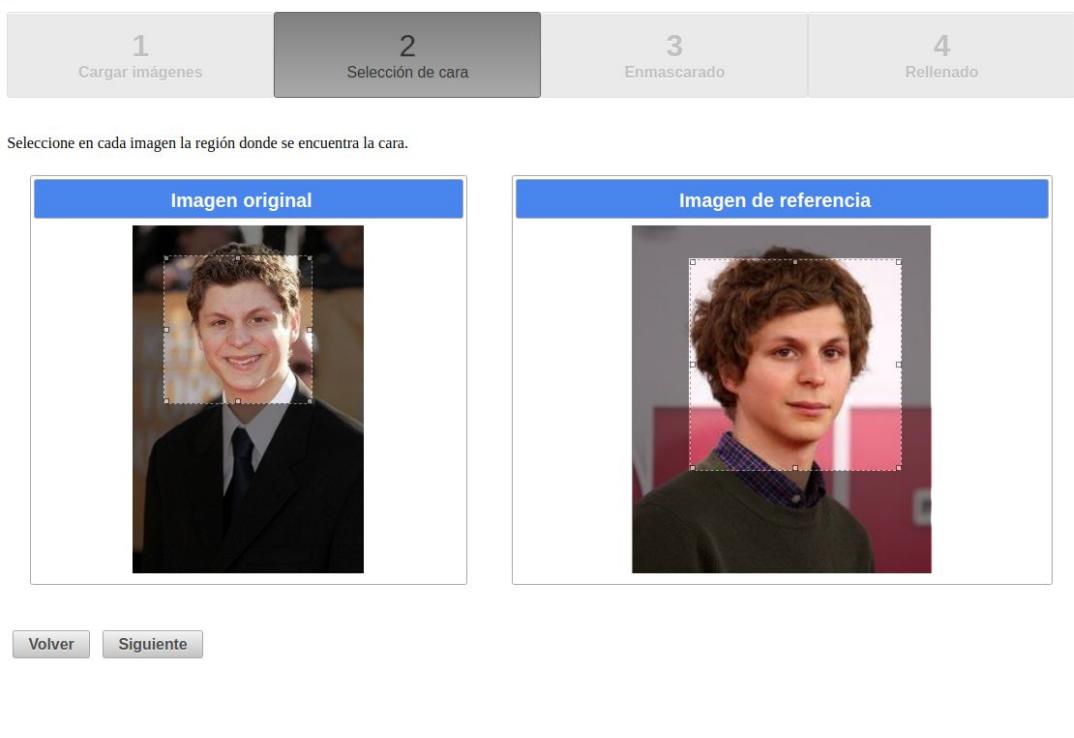


Figura 11: Segundo paso de la aplicación: "Selección de cara"

En el segundo paso de la aplicación (Figura 11), el usuario puede seleccionar la región donde se encuentra la cara en cada imagen. Esto es importante, ya que el modelo requiere como entrada imágenes de caras. De esta forma, si el usuario quiere probar el modelo con una imagen de cuerpo completo como se ve en la Figura 11, se pueden recortar las caras de las imágenes. A su vez, la aplicación obliga a que la región elegida sea cuadrada, ya que también eso es un requisito para la entrada del modelo. Si esto no se hiciera, habría que alterar la proporción de las imágenes, resultando en imágenes estiradas vertical u horizontalmente.

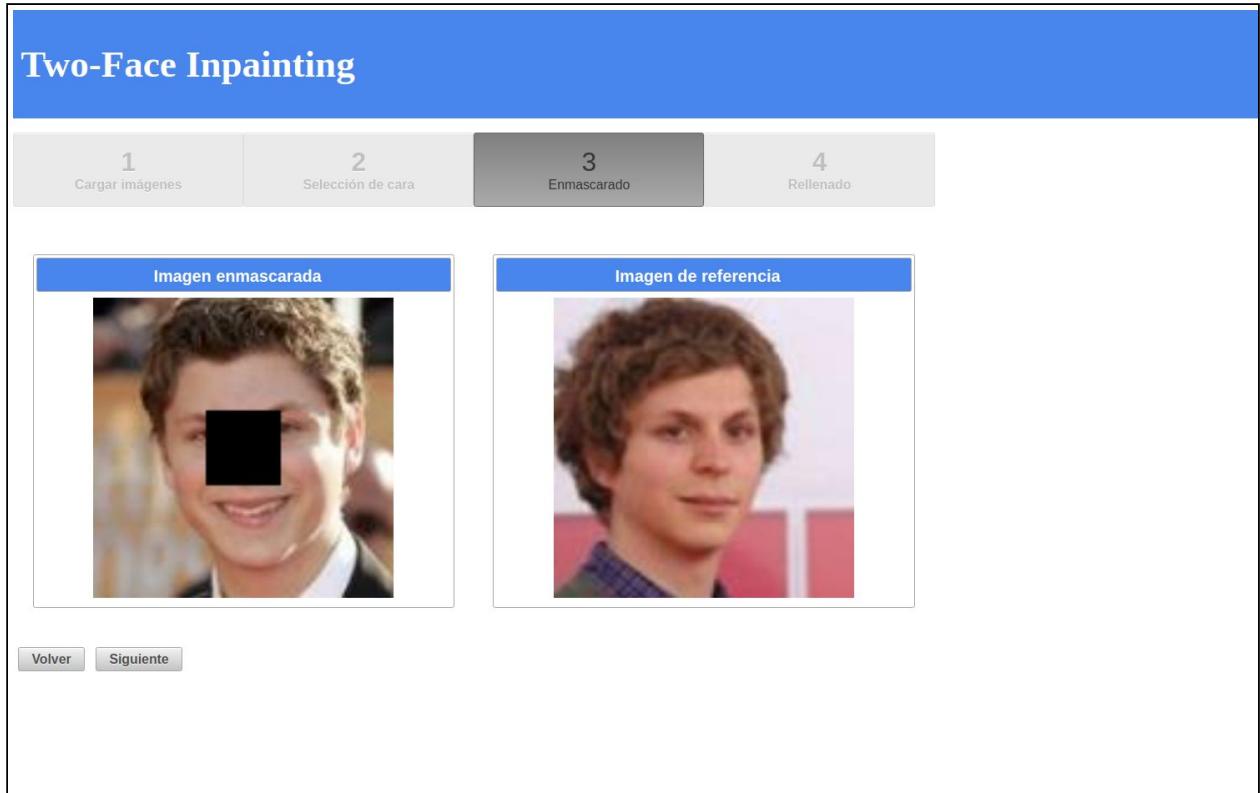


Figura 12: Tercer paso de la aplicación: "Enmascarado"

En el tercer paso de la aplicación (Figura 12), luego de la selección de caras, la aplicación realiza varios procesamientos internos sobre las imágenes. Por un lado, se recortan las regiones de las caras elegidas por el usuario en el paso anterior. Luego, se reduce el tamaño de ambas imágenes a 128x128 píxeles, ya que este es el tamaño requerido como entrada para el modelo. Por último, se aplica una máscara sobre la primer imagen. Los píxeles dentro de la máscara son eliminados, y luego el modelo deberá completar dicha región. En la pantalla de la Figura 12, el usuario puede visualizar el resultado de estos procesamientos antes de enviar las imágenes al modelo. En caso de que no se esté conforme con las regiones elegidas, el usuario puede volver atrás para rehacer la selección.

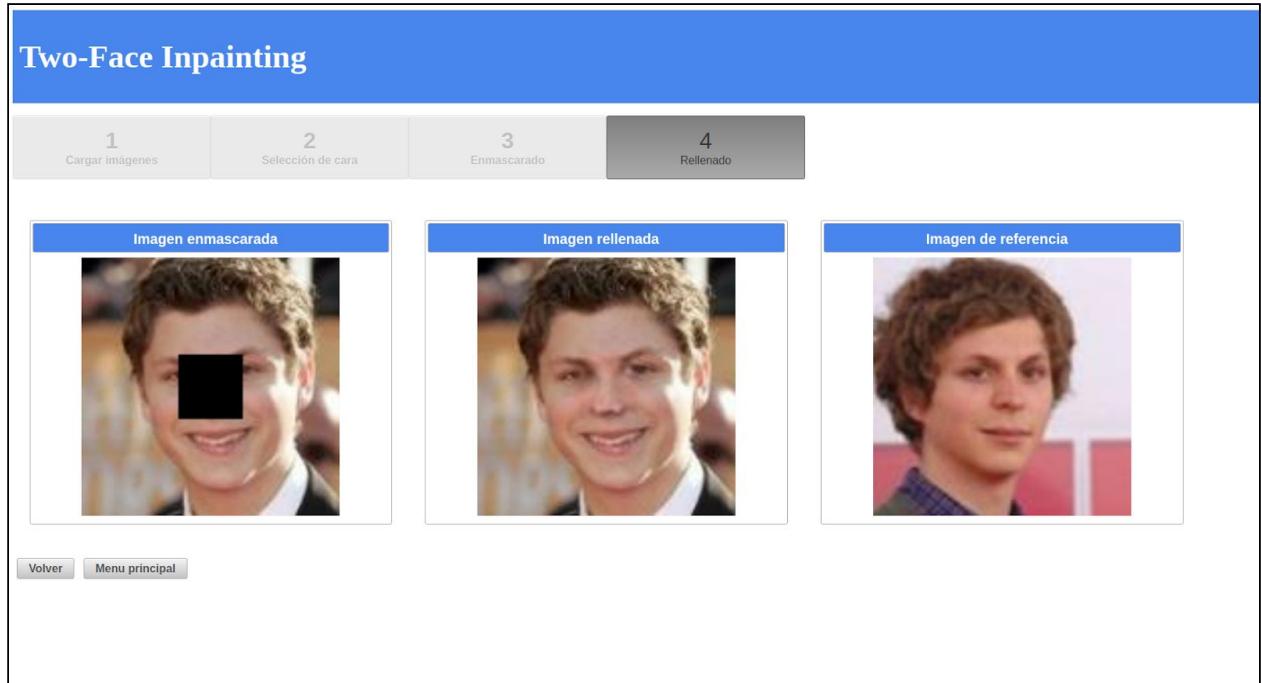


Figura 13: Cuarto paso de la aplicación: "Rellenado"

En el último paso, la aplicación toma las dos imágenes mostradas en el paso anterior, estas son serializadas a formato JSON, y enviadas mediante un servicio REST al backend donde se encuentra alojado el modelo. Una vez que el modelo genera la región faltante, esta es enviada como respuesta del servicio REST (también en formato JSON). Luego, la aplicación web se encarga de deserializar la respuesta, e insertar la región generada en la imagen enmascarada, obteniendo así la imagen rellena o regenerada. Una vez hecho este procesamiento, la imagen enmascarada, la de referencia, y la regenerada son mostradas en la pantalla de la Figura 13.

4.3 Backend Server

El servidor backend es donde se encuentra alojado el modelo entrenado. La función de este servidor es exponer la funcionalidad del modelo para que este pueda ser invocado de forma remota. En nuestra arquitectura, el modelo solo será invocado desde el servidor frontend, pero eso no quita que pueda invocarse desde otros servidores o dispositivos.

Para lograr esto, se utilizó TensorFlow Serving como herramienta para produccionizar el modelo. TensorFlow Serving utiliza Docker para poder deployar de forma sencilla un modelo implementado en TensorFlow, y además provee una API REST a través de la cual se puede invocar dicho modelo. TensorFlow Serving también provee

una API de gRPC para invocar al modelo. En nuestro caso solo utilizamos la API REST, pero podría también invocarse al modelo a través de gRPC.

TensorFlow Serving facilita enormemente la puesta en producción del modelo para inferencia, ya que con un simple comando se puede ejecutar el contenedor Docker de TensorFlow Serving, y este ya queda listo para recibir requests tanto de REST como de gRPC. Esto resultó sumamente útil para realizar pruebas sobre el modelo, como también para llevar a cabo la integración con el servidor frontend.

5 Experimentos y resultados

Durante todo el proceso de implementación y entrenamiento del modelo de Image Inpainting se ejecutaron más de 60 experimentos. En cada uno de ellos se fueron probando distintas arquitecturas, parámetros, funciones de costo, optimizaciones, etc. A continuación se muestran algunos de los experimentos más relevantes.

5.1 MNIST

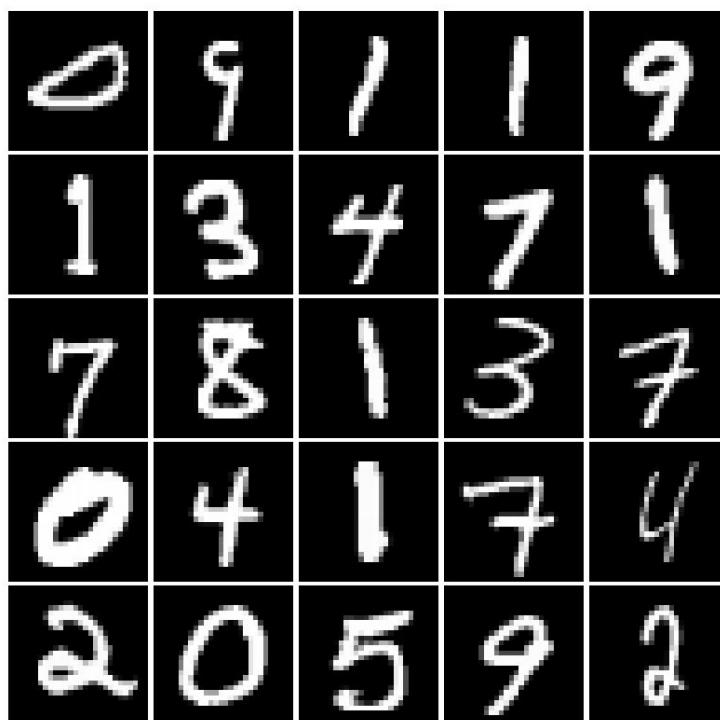


Figura 14: Ejemplos de imágenes del dataset MNIST.

Los primeros experimentos del proyecto fueron realizados utilizando el dataset MNIST [7]. Este dataset consiste de 70.000 imágenes de 28x28 píxeles en blanco y negro de dígitos manuscritos (Figura 14). El dataset de MNIST tiene la ventaja de ser un dataset bastante sencillo, ya que las imágenes son relativamente pequeñas y simples. Esto hace que MNIST sea un dataset ideal para realizar experimentos iniciales. A continuación se describen los experimentos realizados con dicho dataset.

5.1.1 MNIST Inpainting

El primer experimento realizado consistió en implementar y entrenar un modelo que pudiese regenerar regiones faltantes en imágenes de MNIST. En este experimento

no se utilizó una imagen de referencia. El objetivo de este experimento fue desarrollar un modelo inicial que pudiese resolver el problema de image inpainting sobre las imágenes de MNIST.

La arquitectura de este modelo consiste de un Generator y un único Discriminator. El Generator tiene por objetivo generar la región faltante en las imágenes, mientras que el Discriminator se encarga de distinguir entre imágenes reales y falsas (completas). La arquitectura del Generator consiste de un modelo Encoder-Decoder. El Encoder recibe la imagen con la región faltante, y aplica 3 capas convolucionales . El resultado del Encoder luego se pasa al Decoder. El Decoder aplica 3 capas convolucionales transpuestas hasta generar una imagen del tamaño de la región faltante. El Discriminator recibe como entrada una imagen de 28x28. Luego se aplican dos capas convolucionales sobre la imagen, y finalmente se aplica una capa fully-connected de una neurona para determinar si la imagen es real o falsa. Para este modelo se utilizó solamente el loss correspondiente a la arquitectura GAN.

Los resultados obtenidos con este modelo pueden verse en la Figura 15. Se ve que con este modelo inicial se obtuvieron buenos resultados. Salvo por algunos detalles, las imágenes regeneradas se ven bastante realistas.

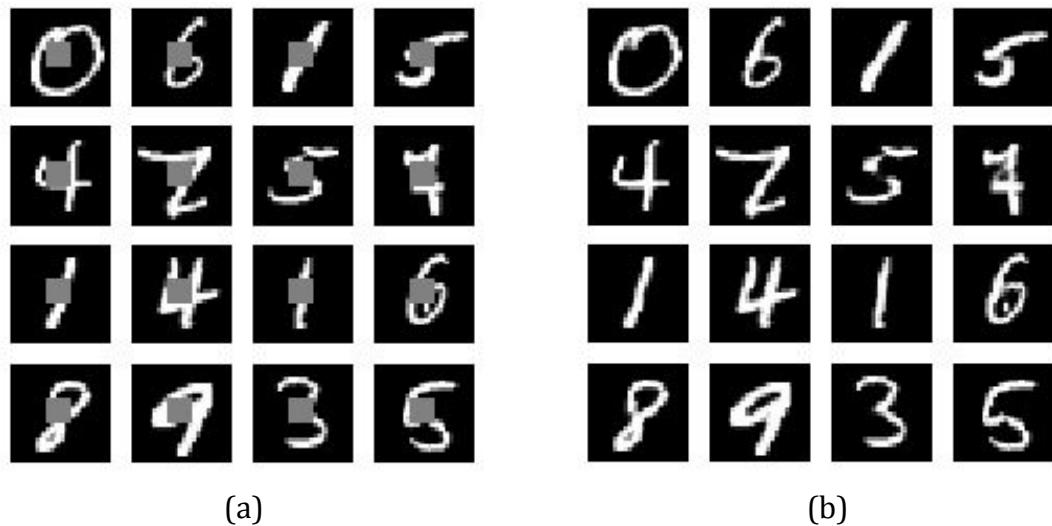


Figura 15: Resultados del experimento MNIST Inpainting. (a): Imágenes enmascaradas. (b): Imágenes regeneradas.

5.1.2 MNIST Two-Image Inpainting

En este segundo experimento se continuó usando el dataset de MNIST, pero esta vez se agregó una segunda imagen a la entrada del Generator. Esta segunda imagen toma el papel de imagen de referencia. Es decir, dada una imagen a regenerar de un determinado dígito (por ejemplo "7"), la imagen de referencia es otra imagen del set de datos del mismo dígito (siguiendo con el ejemplo, también sería una imagen de un "7").

De esta forma, se busca que el Generator tenga en cuenta ambas imágenes a la hora de reconstruir la primer imagen.

Al igual que en el primer experimento, la arquitectura del modelo consiste de un Generator y un Discriminator. El Generator consiste de dos Encoders y un Decoder. Un Encoder recibe la imagen enmascarada, y el otro Encoder recibe la imagen de referencia. Cada Encoder consiste de 3 capas convolucionales, con Batch Normalization y Leaky ReLU. Luego, los resultados de ambos encoders son concatenados y enviados al Decoder. El Decoder aplica 3 capas convolucionales transpuestas hasta generar una imagen del tamaño de la región faltante. El Discriminator recibe como entrada dos imágenes: una imagen que puede ser real o tener una sección regenerada por el Generator, y otra imagen de referencia de ese mismo dígito. Su objetivo es distinguir entre imágenes reales e imágenes regeneradas por el Generator, basándose tanto en la primer imagen como en la imagen de referencia. La arquitectura del Discriminator consiste de dos Encoders y un clasificador. Uno de los Encoders recibe la primer imagen (la cual puede ser real o generada), y el otro Encoder recibe la imagen de referencia. Cada Encoder consiste de 2 capas convolucionales, con función de activación Leaky ReLU y Dropout. Luego, los resultados de ambos Encoders son concatenados y enviados al clasificador. El clasificador aplica 2 capas fully-connected para determinar si la primer imagen es real o falsa.

Los resultados de este experimento se muestran en la Figura 16. Se puede observar que al igual que en el primer experimento, los resultados obtenidos son prometedores, ya que las imágenes generadas siguen siendo bastante realistas.

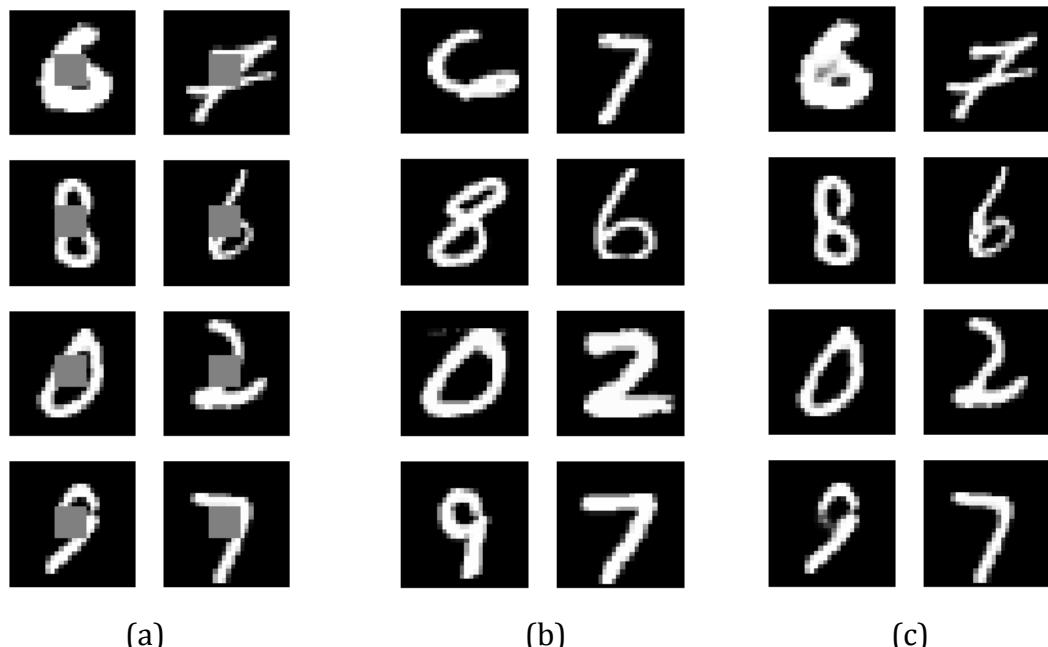


Figura 16: Resultados del experimento MNIST Two-Image Inpainting. (a): Imágenes enmascaradas. (b): Imágenes de referencia. (c): Imágenes regeneradas

5.2 CASIA WebFace

En la segunda serie de experimentos, se pasó a utilizar un dataset de imágenes de rostros. Para este proyecto se utilizó el dataset CASIA WebFace [8], un dataset de 494.414 imágenes de 10.757 identidades (personas). Cada imagen es de 250x250 píxeles en formato RGB, aunque para los experimentos el tamaño de cada imagen fue reducido a 128x128 píxeles para reducir la complejidad del modelo. Un muestreo de imágenes de este dataset puede verse en la Figura 17.

A lo largo del proyecto, se realizaron múltiples experimentos con el dataset de CASIA WebFace hasta obtener el modelo final. En total se hicieron más de 60 pruebas con distintas combinaciones de parámetros y arquitecturas hasta obtener el modelo final. A continuación se mencionan algunos de los experimentos principales.



Figura 17: Ejemplo de imágenes del dataset CASIA WebFace.

5.2.1 Modelo inicial

En este experimento se buscó implementar y entrenar un modelo similar al que fue utilizado para el experimento de MNIST Two-Image Inpainting. Solo que esta vez se utilizó el dataset de CASIA WebFace. El objetivo principal fue poder adaptar el modelo utilizado en dicho experimento para funcionar con imágenes de rostros en vez de

dígitos. En este experimento, el Generator ahora recibe dos imágenes de rostros: una imagen con una región faltante a ser regenerada, y otra imagen de referencia de la misma persona. Tanto la arquitectura general del modelo como la del Generator y del Discriminator son idénticas a las utilizadas en el experimento de MNIST Two-Image Inpainting (con los tamaños de entrada ajustados al nuevo tamaño de las imágenes del dataset CASIA WebFace).

En la Figura 18 se pueden ver algunos de los resultados obtenidos con este experimento. Claramente se puede observar que al tratarse con un dataset mucho más complejo que MNIST, el modelo es bastante limitado como para poder generar buenos resultados. Sin embargo, en este experimento se cumplió el objetivo de tener un modelo inicial funcionando sobre imágenes de rostros.

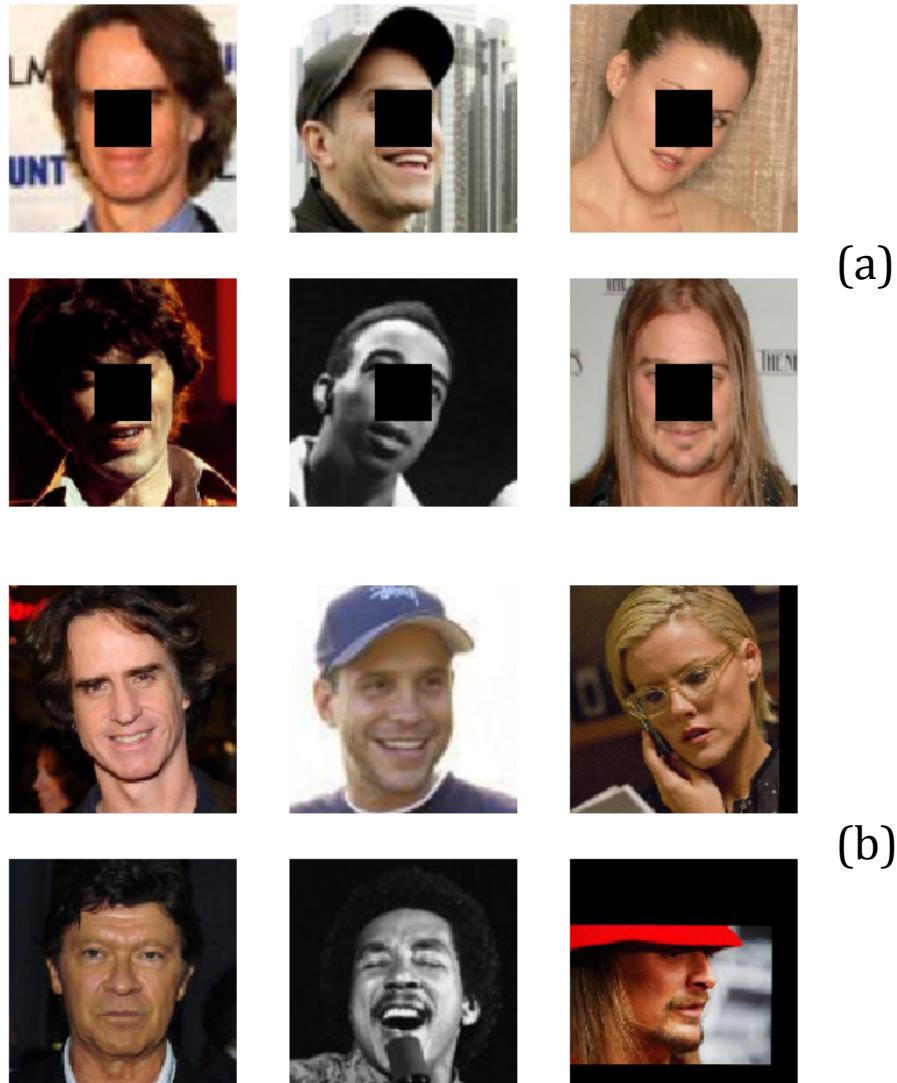




Figura 18: Resultados del experimento inicial con CASIA WebFace. (a): Imágenes enmascaradas. (b): Imágenes de referencia. (c): Imágenes regeneradas

5.2.2 Modelo con VGG

Dados los resultados del modelo inicial utilizado sobre CASIA WebFace, luego de varias pruebas se decidió incrementar el tamaño y capacidad de aprendizaje del modelo. Para eso, se decidió incorporar el modelo pre-entrenado VGG16 [9] al modelo de Two-Face Inpainting. Se agregó el modelo de VGG16 antes de cada encoder del Generator, y también se incorporó este modelo en la entrada del Discriminator. Por un lado, el modelo de VGG16 le da más capacidad de procesamiento a nuestro modelo, ya que incorpora más capas convolucionales que las que se tenían hasta entonces. Por otro lado, al usar el modelo pre-entrenado de VGG16, no es necesario que nuestro modelo aprenda de cero los features básicos de las imágenes (como líneas, bordes, texturas, etc.).

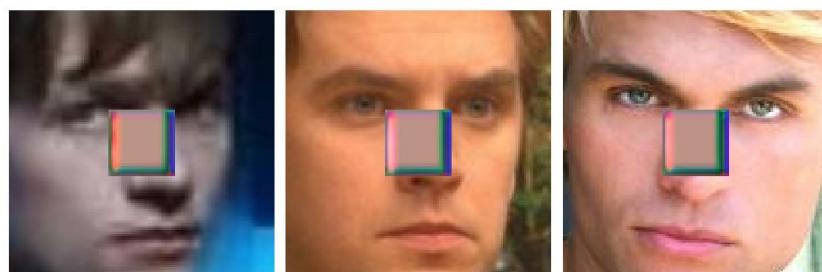
A pesar de haber incorporado al modelo de VGG16 en la arquitectura, se puede ver que aún no se lograron resultados satisfactorios (Figura 19). Viendo que incluso con VGG16 el modelo no logra buenos resultados, se concluye que el incremento en la capacidad del modelo no es suficiente para resolver el problema de inpainting, y es necesario refinar el proceso de entrenamiento.



(a)



(b)



(c)

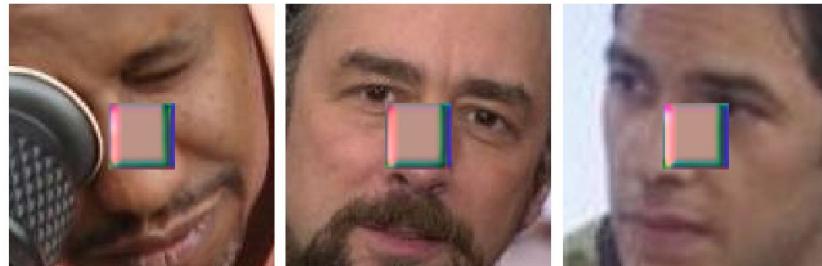


Figura 19: Resultados del experimento con CASIA WebFace y VGG. (a): Imágenes enmascaradas. (b): Imágenes de referencia. (c): Imágenes regeneradas

5.2.3 Modelo final

En base a los resultados de los experimentos anteriores, se decide cambiar el proceso de entrenamiento, que hasta ahora venía siendo un entrenamiento en paralelo del Generator y un único Discriminator. A lo largo de varios experimentos subsecuentes se fueron incorporando los distintos losses descritos en la sección "4.2.1 Arquitectura del modelo". Para recordar, estos son: Reconstruction Loss, Local Discriminator, Global Discriminator, y FaceNet. A su vez, se emplea la técnica de Curriculum Learning descrita en la sección "4.3 Proceso de entrenamiento". La combinación de las 4 funciones de losses y la utilización de Curriculum Learning realmente ayudó al entrenamiento del modelo. Este proceso se fue refinando a lo largo de varios experimentos, hasta obtener la versión del modelo final. Los resultados de dicho modelo pueden apreciarse en la Figura 20.



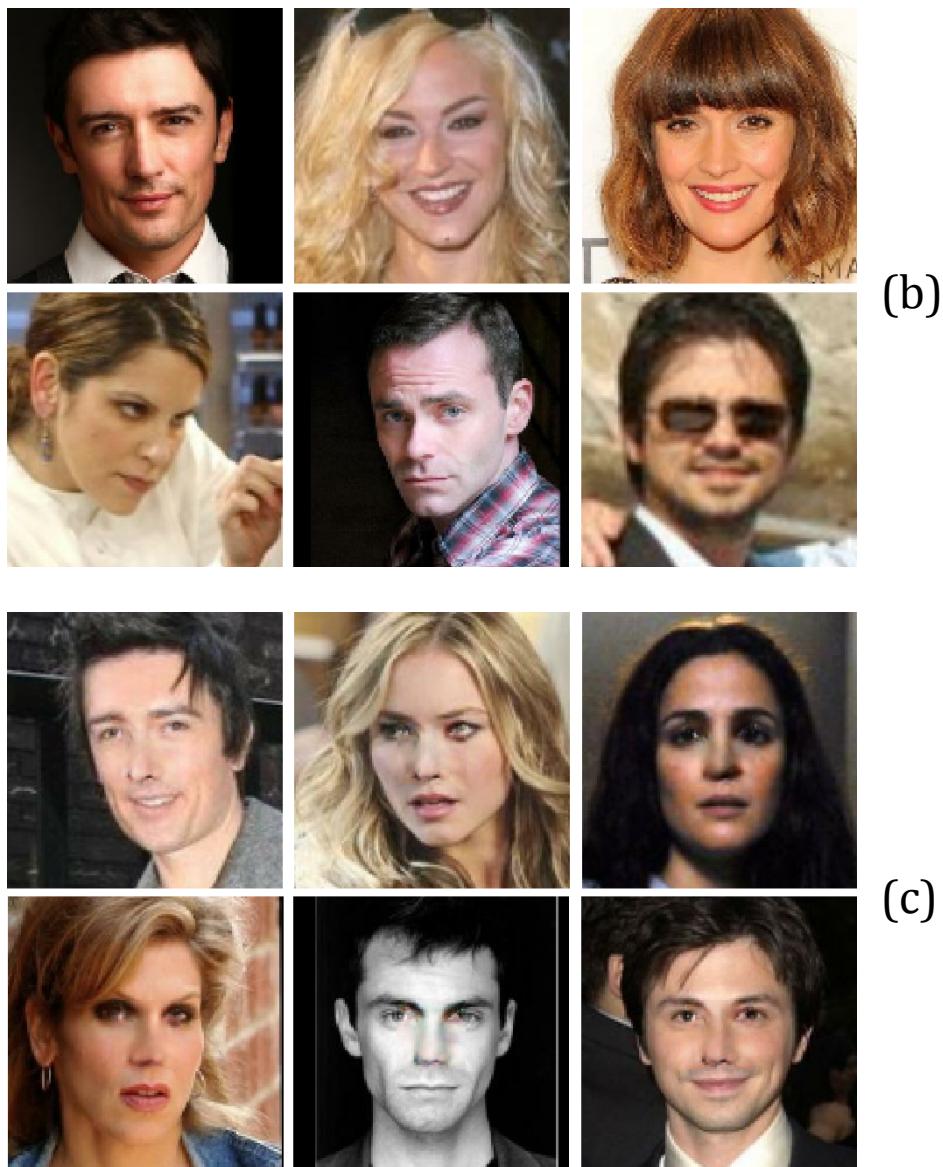


Figura 20: Resultados del modelo final. (a): Imágenes enmascaradas. (b): Imágenes de referencia. (c): Imágenes regeneradas

6 Trabajo a futuro

Dado el tiempo y los recursos predestinados a este trabajo, hubo ideas que quedaron fuera del alcance del proyecto. A continuación se detallan algunas de ellas, de forma tal que puedan ser abordadas en un proyecto posterior.

Una de estas ideas es poder hacer una búsqueda automática de la imagen de referencia. Esto es, dada una imagen a ser regenerada, se querría poder buscar en un dataset de imágenes una imagen apropiada para usar de referencia. Para esto, se podría usar FaceNet para calcular los embeddings de todas las imágenes en el dataset, almacenarlos con las referencias a las respectivas imágenes, e implementar el pipeline para hacer la búsqueda del embedding más cercano al de la imagen que sube el usuario. Notar que en principio no habría garantía de que la imagen que se encuentre sea efectivamente de la misma persona (podría ser otra persona muy parecida, o puede ser que la persona de la primer imagen no tenga otras imágenes en el dataset). Este nuevo modelo debería ser deployado en otro servidor de backend, y habría que actualizar la lógica del frontend para llamarlo y obtener la segunda imagen. Otra cuestión a tener en cuenta sobre esta idea es que hacer la búsqueda sobre todo el dataset cada vez que se llama al modelo puede ser bastante costoso, por lo que habría que investigar posibles optimizaciones.

Por otro lado, debido al costo monetario que tiene ejecutar cada experimento en la nube, y dada la cantidad de experimentos ya realizados durante este trabajo (más de 60), no se siguió realizando una búsqueda exhaustiva de hyper-parámetros una vez obtenido el modelo final con resultados satisfactorios. Un posible trabajo a futuro podría involucrar seguir realizando experimentos sobre el modelo actual, ajustando hyper-parámetros o probando nuevas arquitecturas.

7 Conclusiones

En este trabajo se pudo tomar un problema novedoso y desafiante en el área de Machine Learning como es Image Inpainting, y resolverlo satisfactoriamente con muy buenos resultados. Además, se introdujo un nuevo modelo capaz de tomar una segunda imagen de referencia para completar la región faltante en la primer imagen. Para llevar a cabo este proyecto se aplicaron diversas técnicas y herramientas de última generación en el área de Machine Learning, como Generative Adversarial Networks (GANs), Convolutional Neural Networks, Curriculum Learning, los modelos de FaceNet y VGG16, Google Cloud Platform, TensorFlow, entre otras.

Por otro lado, se implementó una aplicación web a través de la cual es posible realizar pruebas con el modelo entrenado, usando imágenes provistas por el usuario, y con la posibilidad de usar una o dos imágenes de entrada para el modelo. Esto también requirió produccionizar el modelo y deployarlo en un servidor de backend capaz de recibir requests desde la aplicación web. Esta parte del proyecto implicó abordar temas como desarrollo web, infraestructura, servicios REST, y utilizar tecnologías acordes a la tarea como Java EE, Apache Tomcat, JSF, PrimeFaces, TensorFlow Serving, entre otras.

Como se puede ver, en este trabajo se abordó una gran cantidad de temas, algunos vistos a lo largo de la carrera y otros que debieron ser investigados como parte de este proyecto. Visto esto, se puede decir que este trabajo permitió poner en práctica muchos de los conocimientos adquiridos durante la carrera y otros adquiridos durante este proyecto, para finalmente concretar tanto un trabajo de Machine Learning con su serie de experimentos, y una aplicación web que permite mostrar los resultados finales de dicho trabajo.

8 Referencias

[1] Pathak, Deepak, et al. "Context Encoders: Feature Learning by Inpainting". 2016

<https://arxiv.org/pdf/1604.07379.pdf>

[2] Li, Yijun, et al. "Generative face completion." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

<https://arxiv.org/pdf/1704.05838.pdf>

[3] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.

<https://papers.nips.cc/paper/5423-generative-adversarial-nets>

[4] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015). <https://arxiv.org/abs/1511.06434>

[5] Schroff, Florian, Dmitry Kalenichenko, and James Philbin. "Facenet: A unified embedding for face recognition and clustering." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Schroff_FaceNet_A_Unified_2015_CVPR_paper.pdf

[6] Bengio, Yoshua, et al. "Curriculum learning." Proceedings of the 26th annual international conference on machine learning. ACM, 2009.

[7] MNIST database. <http://yann.lecun.com/exdb/mnist/>

[8] Yi, Dong, et al. "Learning face representation from scratch." arXiv preprint arXiv:1411.7923 (2014). <https://arxiv.org/abs/1411.7923>

[9] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

<https://arxiv.org/pdf/1409.1556.pdf%20http://arxiv.org/abs/1409.1556.pdf>

[10] Ng, Andrew Y. and Jordan, Michael I. "On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes." NIPS, 2001.

<https://papers.nips.cc/paper/2020-on-discriminative-vs-generative-classifiers-a-comparison-of-logistic-regression-and-naive-bayes>

[11] Karpathy, Andrej, et al. "Generative Models." OpenAI, 2016

<https://blog.openai.com/generative-models/>

[12] Goodfellow, Ian. "NIPS 2016 tutorial: Generative adversarial networks." arXiv

preprint arXiv:1701.00160 (2016). <https://arxiv.org/abs/1701.00160>

[13] Yeh, Raymond A., et al. "Semantic Image Inpainting with Deep Generative Models."

CVPR. Vol. 2. No. 3. 2017.

http://openaccess.thecvf.com/content_cvpr_2017/papers/Yeh_Semantic_Image_Inpainting_CVPR_2017_paper.pdf

[14] Yu, Jiahui, et al. "Generative image inpainting with contextual attention." arXiv

preprint (2018).

http://openaccess.thecvf.com/content_cvpr_2018/papers/Yu_Generative_Image_Inpainting_CVPR_2018_paper.pdf

[15] Demir, Ugur, and Gozde Unal. "Patch-Based Image Inpainting with Generative

Adversarial Networks." arXiv preprint arXiv:1803.07422 (2018).

<https://arxiv.org/abs/1803.07422>

[16] GAN architecutre image from "A Beginner's Guide to Generative Adversarial

Networks (GANs)." <https://skymind.ai/wiki/generative-adversarial-network-gan>