UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

SÍNTESIS DEL HABLA POR MEDIO DE GENERACIÓN SINTÉTICA DE EMBEDDINGS DE TACOTRON 2 MULTISPEAKER, BASÁNDOSE EN EL GÉNERO DE VOZ

TESIS

QUE PARA OPTAR POR EL GRADO DE:

MAESTRÍA EN INGENIERÍA Y CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

JORGE ACOSTA DOMÍNGUEZ

TUTOR PRINCIPAL

DR. CALEB ANTONIO RASCON ESTEBANÉ

INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y EN SISTEMAS

JURADO ASIGNADO:

Presidente	Dr. Gibrán Fuentes Pineda Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Secretario	Dra. Wendy Elizabeth Aguilar Martínez Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Vocal	Dr. Caleb Antonio Rascon Estebané Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
1er. Suplente	Dra. Jimena Olveres Montiel Facultad de Ingeniería
2o. Suplente	Dr. Iván Vladimir Meza Ruiz Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Ciudad de México, Abril del 2022

Ciudad Universitaria, CD. MX., Abril del 2022

Ésta tesis está dedicada para todos aquellos que me apoyaron, en especial a mis padres, el IIMAS, la UNAM y el CONACYT por haberme ayudado en diversas maneras a poder llegar a éste punto de mi vida académica.

También le dedico ésta tesis a todos aquellos que con sus pequeñas aportaciones a la ciencia y tecnología han podido lograr que lleguemos a éste nivel de avances tecnológicos y que cada vez sea menos complicado el camino para aquellos que van empezando.

Reconocimientos

A mis padres por apoyarme siempre. Sin ustedes no hubiera llegado tan lejos.
Al IIMAS y la UNAM por haberme dado un espacio para desarrollar mi tesis, así como haberme dado una educación de calidad.
Al CONACYT por haberme apoyado económicamente durante mi estancia en la maestría y haberme permitido el poder continuar con mis estudios de ésta manera.
A mis profesores por haber compartido su conocimiento conmigo, además de haberme dado consejos cuando fue necesario.
A mi asesor de tesis Caleb Rascón Estevané, muchas gracias por apoyarme durante todo éste proceso; aprendí mucho de usted.
A mis amigos que siempre me apoyaron y dieron un pequeño empujoncito para seguir adelante a pesar de todo, muchas gracias.
Al usuario Corentin Jemine por haber hecho público su repositorio dónde implementó el modelo T2MS y con ello haberme ahorrado implementar algo fuera del alcance de mi proyecto.

Declaración de autenticidad

Por la presente declaro que, salvo cuando se haga referencia específica al trabajo de otras personas, el contenido de esta tesis es original y no se ha presentado total o parcialmente para su consideración para cualquier otro título o grado en esta o cualquier otra Universidad. Esta tesis es resultado de mi propio trabajo y no incluye nada que sea el resultado de algún trabajo realizado en colaboración, salvo que se indique específicamente en el texto.

Jorge Acosta Domínguez. Ciudad Universitaria, CD. MX., Abril del 2022

Resumen

En éste escrito presentamos varias propuestas para poder generar audios de voces las cuales puedan ser condicionadas por un género seleccionado. Todo esto se realiza utilizando la arquitectura T2MS la cual nos permite sintetizar audio de voces incluso aún no vistas, y tomando ventaja de ello utilizamos el espacio de embeddings que genera su módulo codificador como base para nuestra generación condicionada.

Las propuestas para lograr ésta tarea se basan en técnicas de *machine learning* y de *deep learning*. Para machine learning utilizaremos PCA para reducir la dimensionalidad del espacio de *embeddings* (el original es de 256 dimensiones), posteriormente utilizaremos SVM para poder generar dos subespacios, uno para cada género de voz, y por último utilizaremos dos técnicas distintas para generar los *embeddings*: una parametrizando los subespacios y otra utilizando K-means. Para ambos, el *embedding* se genera al muestrear los subespacios o un punto del vecindario de los *clousters* en K-means, respectivamente, y posteriormente aplicando una PCA inversa para llevarlo de nuevo al espacio de *embeddings*. Para *deep learning* utilizaremos modelos generativos: βVAE y cGAN, ambos con el espacio de *embeddings* original, para generar embeddings.

Se obtuvieron resultados súmamente favorables en las técnicas de *machine learning*, ya que en ellas se generan audios de voces donde el género es el solicitado, y es claramente identificable, además de que las voces pueden variar según los parámetros (no es una única voz). Mientras que para las técnicas de *deep learning* se obtuvieron resultados mixtos, ya que, aún cuando en algunas arquitecturas (βVAE) se pudieron obtener buenas representaciones latentes y con esto audios de voz con el género solicitado y con voz que puede cambiar según los parámetros, no en todos los modelos se logró esto e incluso los audios no pertenecían a lo que se puede considerar una voz humana (casi todos los de cGAN), sino algo más cercano algo proveniente de bestias.

Así pues, la gran mayoría de las propuestas cumplieron con generar voces de humanos según el género pedido, pero además, en los modelos de βVAE se pudieron encontrar modelos donde una única variable latente podía (al ser variado su valor) cambiar el género de la voz, lo cual da pauta a que tomando éste conocimiento como base se pueda aplicar a otras características de la voz además del género de voz.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La comunicación oral es una de las maneras más eficaces que ha tenido el ser humano para transmitir sus conocimientos y expresarse con los demás. A lo largo de la evolución del ser humano como especie, hemos adaptado la forma de comunicación pasando de sonidos puramente guturales a generar un lenguaje articulado avanzado. Asimismo, con los nuevos avances tecnológicos, la necesidad de expresarse ha aumentado drásticamente, y los mensajes de texto no son suficientes en muchas de las situaciones, por lo que opciones multimedia (voz y video) resultan una mejor opción para esto.

Sin embargo, existen ciertas limitaciones derivadas del entorno y circunstancias por las que no es posible utilizar multimedia. Ya sea por falta de ancho de banda necesario, limitaciones técnicas, o bien, discapacidades (como mudez). Por lo tanto, se han desarrollado máquinas y sistemas creados (por ejemplo robots de servicio y chatbots), que originalmente carecen de una manera para poder comunicarse oralmente con los humanos.

A lo largo de los años, se han creado y mejorado sistemas que permiten a éstos el poder comunicarse con los humanos de manera oral. Sin embargo, estos aún se encuentran en fases donde no se puede replicar la voz humana fielmente ya que no se pueden copiar muchos de sus aspectos que la hacen diferenciarse de la voz sintética. Para poder acercarnos a una replicación fiel de la voz es necesario empezar con generación de audios de voces variando aspectos de base de la voz, como el que se muestra éste trabajo: el variar el género de voz.

1.1. Presentación del problema

Al proceso de generar sonidos emulando la voz humana, se le llama síntesis de voz, y es una tarea que se ha estado desarrollando y perfeccionando desde hace siglos (4), sin embargo, sólo en los últimos años es cuando se ha podido dar grandes avances al imitar la voz humana (1). Pero aún estamos muy lejos de poder recrear voces humanas de una manera tan realista que sea imposible diferenciar fácilmente las sintéticas de las

reales. Un paso para lograrlo es poder controlar características de la voz humana que hacen que hoy en día sea difícil emularlas fielmente.

Dado que existen muchas características de la voz, nos centraremos en una que intrínsecamente lleva información de otras (por ello podría ser refinado posteriormente el trabajo para centrarse en cada una de ellas). La característica en cuestión es el género de la voz (aclaración: aquí se llamará género de voz a la clasificación que se puede hacer para clasificar a una voz en género femenino o masculino), la cual nos servirá para poder generar voces que cumplan nuestro criterio de selección a la hora de su síntesis.

Un tipo de síntesis de voz, muy utilizado a la fecha es la síntesis de texto a voz (TTS por sus siglas en inglés: *Text-To-Speech*) a la cual se le alimenta un texto de entrada y se genera un audio donde una voz pronuncia el texto que se le alimento. Es común que también se le alimente una voz de referencia con la cual se genere el audio. Entre los sistemas de síntesis de voz más avanzados hoy en día tenemos a (1) (se le llamará T2MS, *Tacotron 2 Multi-Speaker*, a partir de ahora) el cual permite sintetizar texto tomando como referencia una voz base. Asimismo, cada voz se puede mapear en un espacio de altas dimensiones donde las voces representan puntos diferentes en dicho espacio, y además es posible utilizar voces que el sistema no conoce. Basándonos en esto, en éste trabajo utilizaremos el espacio de *embeddings* de T2MS para poder centrarnos en la característica de voz elegida para poder generar una voz basada en ella: el género de voz.

Si bien existen algunos trabajos donde se tratan algunas características de la voz (5) (6), en éste nos centraremos en el género de voz ya que ha sido poco explorado, igual como el uso de algoritmos generativos para este propósito.

1.2. Hipótesis

El espacio de *embeddings* creado por T2MS permite el condicionamiento del género en un sistema de síntesis de voz.

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar un sistema que permita generar *embeddings* síntéticos basándose en un género de voz elegido por el usuario utilizando como base el espacio de *embeddings* propuesto en T2MS.

1.3.2. Objetivos específicos

- Investigar métodos de generación sintética de datos

- Implementar y/o adecuar métodos de generación sintética de datos que se puedan adecuar a nuestros datos
- Implementar una versión de T2MS para el uso de su espacio de *embeddings* y síntesis de voz
- Generar *embeddings* sintéticos condicionados a un género de voz
- Generar voces basándonos en los *embeddings* sintéticos generados en éste trabajo (para corroborar auditivamente)

1.4. Contribución

La contribución de este trabajo es proporcionar bases e intuiciones de cómo poder generar *embeddings* de voces con una característica en específico. En éste caso la característica en cuestión es el género de voz. Sin embargo, en un futuro podría retomarse lo mostrado en éste trabajo para poder generar *embeddings* de voces dado otras características como, por ejemplo, el acento, la edad, y el timbre. Esto, claro, haciendo su respectivo pre-procesamiento de datos y etiquetado de los mismos.

Capítulo 2

Síntesis de voz

La síntesis de voz es uno de los campos donde la inteligencia artificial ha participado y tomado un papel muy importante al poder generar con ella sintetizadores de voz que pueden acercarse a la clonación de voz (7)(1), lo cual no hubiera sido pensable tiempo atrás y sólo estaría en los sueños de los fanáticos de la misma.

En este capítulo abordaremos lo relacionado a la síntesis de voz, desde el origen de su necesidad en la sección 2.1, pasando por su primeras aproximaciones y evolución en la sección 2.2, hasta llegar a la actualidad con una de las arquitecturas más novedosas y que abre camino a la posibilidad de generar síntesis de voz sin necesidad de hablantes reales de referencia en la sección 2.3.

2.1. Origen

La comunicación oral es una de las maneras más eficaces que ha tenido el ser humano para transmitir sus conocimientos y expresarse con los demás. A lo largo de la evolución del ser humano como especie, hemos adaptado la forma de comunicación pasando de sonidos puramente guturales a generar un lenguaje articulado avanzado. Asimismo, con los nuevos avances tecnológicos, la necesidad de expresarse ha aumentado drásticamente, y los mensajes de texto no son suficientes en la mayoría de situaciones, por lo que opciones multimedia (voz y vídeo) resultan una mejor opción para esto. Pero existen ciertas limitaciones derivadas del entorno y circunstancias por las que no es posible utilizar multimedia. Ya sea por falta de ancho de banda necesario, limitaciones técnicas, o bien discapacidades como mudez. También existen máquinas creadas por humanos, físicas y digitales (robots de servicio y chatbots, respectivamente), que carecen de una manera de poder comunicar oralmente aquello que se quiere transmitir.

Una forma de solucionar esto son los sistemas sintetizadores que tienen como finalidad generar una salida auditiva la cual asemeje una voz humana. A lo largo de la historia, se han intentado generar sintetizadores de voz con base en la tecnología disponible, desde la máquina hablante de Kempelen (8) que era un modelo mecánico que simulaba el sistema de producción del habla humana utilizando tubos y otros mate-

riales para llegar a producir vocales, llegando hasta métodos de síntesis de voz donde se pueden replicar desde vocales hasta palabras complejas, utilizando desde sistemas embebidos hasta servidores.

2.2. Evolución de la síntesis

Como parte específica de estos sintetizadores, existen los sintetizadores texto a voz (*Text-to-Speech*, TTS), los cuales son capaces de generar una salida de audio dada una entrada de texto. La salida emitirá una voz pronunciando lo dicho en la entrada de texto. Para generar esto, han surgido diversas aproximaciones a lo largo de la era moderna, entre las más populares se encuentran la manipulación de señales de audio de voz directamente, predicción linear, modelado físico del tracto vocal articulatorio y síntesis de formante (4). Sin embargo, últimamente con los avances en inteligencia artificial (IA), y mediante el aumento del poder computacional y el acceso a éste, se han utilizado aproximaciones donde la mayor parte del trabajo de síntesis se deja a los sistemas de IA para poder encontrar las representaciones más adecuadas de la voz para con ello poder generar una voz más natural cada vez.

Un sistema TTS, en la actualidad, se basa mayormente en el uso de redes neuronales artificiales (ANN), redes neuronales recurrentes (RNN), redes neuronales profundas (DNN), módulos de atención y autocodificadores variacionales (VAE). Donde uno de los modelos más destacados por sus resultados en éste campo fue desarrollado por Google llamado "Tacotron"(9), el cual apostó por una arquitectura mayormente compuesta por RNN y un módulo que ellos llamaban CGHG (compuesto por bancos convolucionales y GRU bidireccional). Éste modelo apuntaba a generar un TTS donde sólo nos debíamos de preocupar por tener audios con sus correspondientes textos donde se indica qué se dice en dichos audios. Sin embargo, por su parte Baidu presentaba "Deep voice"(10), con una premisa muy similar respecto a no requerir procesamientos previos de los datos, pero con una arquitectura basada únicamente en DNN, donde la representación del audio se encontraba internamente, al igual que con Tacotron. Tiempo después apareció "Tacotron 2" (11) modificando la arquitectura original cambiándola por un modelo de secuencia a secuencia recurrente y se condicionaba la salida a ser un espectograma de MEL que un vocoder utilizaba para generar formas de onda que resultaban en una voz.

Sin embargo, estos modelos sólo podían generar síntesis de una voz en específico, y por ende los audios de entrenamiento tenían la voz de una sola persona. Esto cambió poco después cuando se liberó "Deep Voice 2" (12), el cual es un TTS multi-hablante donde las características de la voz de la persona se pasaban a los módulos de segmentación, duración y frecuencia dentro de la arquitectura, con el fin de que la salida fuera diferente con cada hablante. Incluso propusieron un modelo multi-hablante para Tacotron con el fin de realizar comparaciones más justas. Sin embargo, al tener que pasar los datos de voz a cada módulo, se dificultaba mucho usar otras voces, menos aún no conocidas.

2.3. Tacotron 2 Multi-Speaker (T2MS)

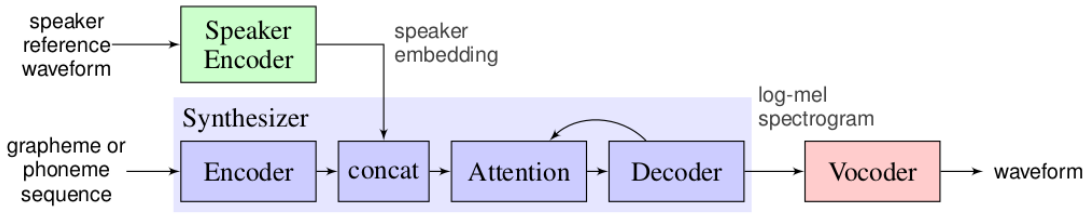

Figura 2.1: Arquitectura completa de T2MS. Tomado de (1).

Con la llegada de "Transfer learning from speaker verification to multispeaker text-to-speech synthesis" (1) se solucionó el problema de poder utilizar otras voces de manera más eficaz ya que las voces de los hablantes se mapean a un espacio de *embeddings* (representaciones vectoriales) de 256 dimensiones en un módulo de codificación del hablante. Los *embeddings* generados aquí son utilizados posteriormente en un módulo llamado "sintetizador" que los utiliza junto con la salida de un codificador de grafemas o fonemas para poder generar un espectograma de MEL a su salida. Por último, se emplea un vocoder para tranformar el espectograma de MEL producido por el sintetizador a una forma de onda de voz. Esta arquitectura se puede ver en la figura 2.1.

Debido a lo bien representados que se encuentran los hablantes dentro del espacio de *embeddings* que genera T2MS es posible utilizar voces de hablantes que no fueron utilizados durante el entrenamiento (ni en validación). Esto se genera manipulando los valores de los *embeddings*, sin embargo no existe parametrización alguna para poder saber cómo será la voz de salida en caso de que se utilice un *embedding* no visto durante entrenamiento, ni utilizando uno generado basándose en una voz ordinaria.

2.3.1. Speaker Encoder: *Embeddings*

Al manejar datos que existen en espacios de alta dimensionalidad, o que sus características no pueden ser separadas de manera eficiente en el espacio en el que habitan, surge la necesidad de poder representar esos datos de una manera que puedan expresarse sus características de una manera que sea más fácilmente identificable, o bien, que puedan separarse sus diferentes características y con ello poder manejarlas más adecuadamente. Ésta es la función principal de los *embeddings*, el poder tener una representación de las características de los datos que puedan ser manejadas más fácilmente, e incluso su visualización no sea un problema cuando se utilizan muy bajas dimensiones.

Podemos considerar a los *embeddings* como una representación única en un espacio vectorial, de un dato de una dimensionalidad diferente a la del *embedding*. Los *embeddings* suelen ser de menor dimensionalidad que el dato original. El proceso para obtener los *embeddings* varia y depende de los datos y el cómo se les tratará, pero básicamente

suelen realizarse operaciones de reducción de dimensionalidad sobre cada dato original para obtenerlos (13). También, más relevante al tema de este trabajo, también se suelen usar las representaciones intermedias de algunos modelos de inteligencia artificial como *embeddings*, justo como pasa en T2MS y presentado en la figura 2.2.

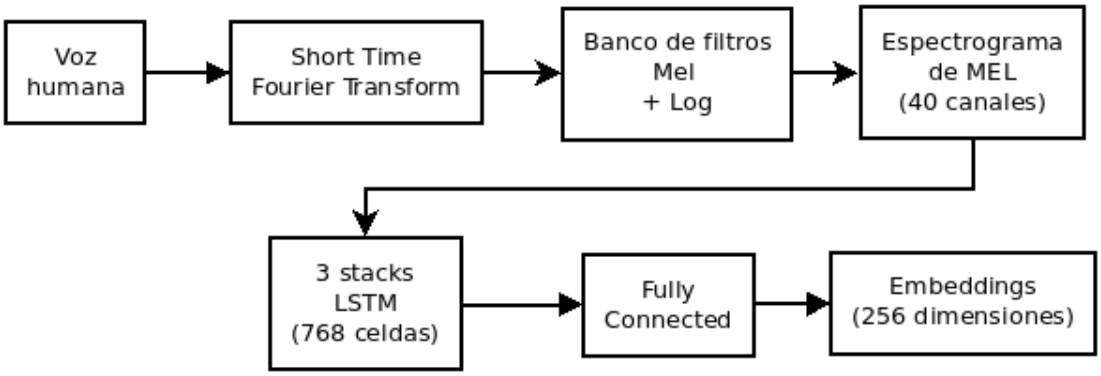

Figura 2.2: Módulo codificador del hablante de T2MS

2.3.2. Sintetizador

El módulo sintetizador en T2MS básicamente se puede describir como un autoencoder (esto se explicará en el siguiente capítulo) con pasos intermedios extra (concatenación y atención). El cuál se entrena con una base de datos de textos que corresponda, en alto nivel (auditivo), a lo que pronuncia el hablante del que se obtiene y pasa el *embedding* de voz. Éste módulo concatena el *embedding* del hablante en la representación intermedia que genera el codificador y genera un vector que se le pasa como entrada a un módulo de atención. Éste módulo de atención alimenta y se reatroalimenta del decodificador, el cual genera como salida un espectograma de MEL. En la figura 2.3 se muestra un diagrama a bloques de éste proceso.

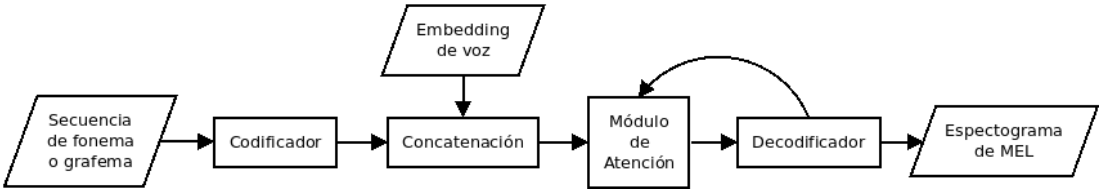

Figura 2.3: Módulo sintetizador de T2MS

2.3.3. Vocoder

La tarea del vocoder es de transformar un espectograma y generar una señal de audio como salida. Existen diversos vocoders que son utilizados para generar las señales

de onda de voz. El más famoso y utilizado hasta hace poco era WaveNet (debido al uso que se le ha dado en Tacotron 2 y T2MS). WaveNet (14) es un modelo generativo recurrente para señales de audio muestreadas a 16 KHz. Genera el audio muestra por muestra basándose en las muestras generadas anteriormente utilizando múltiples capas de redes recurrentes.

Sin embargo, han aparecido otros vocoders que resultan ser mucho más rápidos que WaveNet, así como generar resultados más "naturales", es decir, más cercanos a voces generadas por humanos. Entre estos tenemos a WaveRNN(15), creado por una colaboración entre DeepMind y Google Brain. WaveRNN se diferencia de WaveNet en utilizar una simple capa recurrente así como generar la salida utilizando dos *softmax* para la salida (en lugar de una como WaveNet).

También tenemos a WaveGlow (16) creado por Nvidia, el cual se diferencía de WaveNet al no utilizar redes recurrentes y en su lugar utilizar redes basadas en flujo (que aseguran invertibilidad, lo cual ayuda a calcular mejor similitudes al calcular la pérdida).

Un espectograma de MEL (17) es aquel espectograma (representación en frecuencia de una señal en el tiempo) donde el rango de frecuencias original se transforma en uno que imita el de la percepción humana mediante una escala que toma en cuenta el rango de frecuencias humana. Ésta escala se calcula con la ecuación 2.1

$$Mel(f) = 2595 \cdot log(1 + \frac{f}{700}) \tag{2.1}$$

Capítulo 3

Marco teórico

A lo largo de este capítulo abordaremos técnicas, algoritmos, modelos y arquitecturas de inteligencia artificial que nos ayudarán a entender mejor el presente trabajo. En la sección 3.1 hablaremos sobre PCA y t-SNE, un par de algoritmos que sirven para la reducción de la dimensionalidad y que principalmente suelen utilizarse para visualización de datos de altas dimensionalidades en espacios de más baja dimensionalidad. En la sección 3.2 daremos un abordaje a técnicas de inteligencia artificial que nos sirven para aprender estructuras de los datos que utilicemos. Asimismo veremos arquitecturas de deep learning que se utilizan para la generación/reconstrucción de datos: βVAEs y CGANs.

3.1. Reducción de dimensionalidad

Dentro del ámbito de inteligencia artificial, solemos utilizar bases de datos enormes, tanto en cantidad de datos (o renglones, si lo vemos como una tabla) como en cantidad de características (o columnas). Una forma de realizar análisis de esos datos es el poder visualizar la información de una manera que nos sea más sencilla el poder identificar características como agrupamiento de datos, etc. Sin embargo, ya que para el ser humano el poder visualizar datos más allá de 3D no es factible, es necesario encontrar maneras de transformar nuestros datos de d dimensiones en una dimensionalidad m, donde $d > m$ y $m < 4$, para realizar ésta visualización. Para esto utilizamos algoritmos como PCA y t-SNE que nos ayudan a realizar una reducción de la dimensionalidad que a su vez puede utilizarse para manipular y extraer características de los datos. Sin embargo, al tener menos dimensiones, los datos pierden información por lo que el análisis en una baja dimensionalidad podría no ser útil en algunos casos.

3.1.1. Principal Components Analysis (PCA)

Principal Components Analysis (PCA) es una técnica de análisis de datos originada en 1901 en (18). Su uso principal es de reducción de dimensionalidad, de una dimensión

p a una k donde $k < p$, aunque nos permite obtener características del espacio donde residen los datos. La reducción de la dimensionalidad se realiza al encontrar los componentes principales (de ahí el nombre) que representan al espacio original. Asimismo mantiene la varianza al reducir la dimensionalidad (19). La base de esto es que los datos en p dimensiones pueden tener algunas de ellas correlacionadas entre sí, por lo que es posible llegar a un espacio dimensionalmente más bajo donde se encuentren algunas de sus dimensiones base. Esto se realiza utilizando los bien llamados eigenvectores y eigenvalores, así como una matriz de covarianza.

Partimos de un conjunto de datos X de dimensiones $p \times n$. De este conjunto de entrada obtenemos su matriz de covarianza C usando la ecuación 3.1, después, obtenemos sus eigenvectores e eigenvalores W e λ, los cuales podemos obtener utilzando SVD o SVD truncada (20). Después de utilizar SVD sólo W nos servirá para poder calcular la PCA, ya que para calcularlos utilizamos la ecuación 3.2 donde Y es la matriz resultante donde los datos bajan su dimensionalidad a una k que nosotros hayamos elegido previamente. Aquí Y es de dimensionalidad $k \times n$, W es de $p \times k$ (transpuesto, W', sería $k \times p$) y X es de $p \times n$, siendo k el número de componentes principales con el que queremos trabajar y p las dimensionalidad original. Si bien los eigenvectores que obtenemos al utilizar SVD, o alguna otra técnica, pueden ser más que el número de dimensiones que queremos utilizar, basta con ordernar los eigenvectores respecto a su eigenvalor y utilizar los k que tengan mayor valor.

$$C = \frac{1}{n-1}\left(xx'\right) \tag{3.1}$$

$$Y = W'X \tag{3.2}$$

Singular Value Decomposition (SVD) es una técnica que nos permite encontrar los eigenvalores e eigenvalores de una matriz utilizando operaciones matriciales y determinantes. Teniendo una matriz A, obtenemos una matriz $V = AA^T$, donde V es cuadrada. Considerando ésta matriz V podemos buscar un par de vectores W y λ que corresponden a los eigenvectores e eigenvalores de la matriz y que satisfagan la ecuación 3.3. Dado que la ecuación 3.3 la podemos ver como 3.4, podemos ver que para que la igualdad anterior sea 0 la ecuación 3.5 se debe cumplir para un conjunto único de eigenvalores. Así pues, resolvemos ésta última y obtenemos lo valores de λ, los cuales posteriormente usamos para resolver la ecuación 3.4 y con ello obtenemos W. Cabe señalar que V es de dimensionalidad $p \times p$, W de $p \times k$, y λ de $k \times k$.

$$V\dot{W} = \lambda X \tag{3.3}$$

$$\left(V - \lambda I\right) X = 0 \tag{3.4}$$

$$\left|\left(V - \lambda I\right)\right| = 0 \tag{3.5}$$

3.1.2. t-Distributed Stochastic Neighbor Embedding (t-SNE)

De acuerdo a (21), t-Distributed Stochastic Neighbor Embedding (t-SNE) es una técnica de visualización de datos donde se pasa de datos de una dimensión n a una dimensión m, siendo $m < n$. Es capaz, además, de poder encontrar estructuras globales y de capturar mucha de la estructura local de los datos. Es una variación de SNE (Stochastic Neighbor Embedding) con la diferencia de que la función de pérdida es simétrica, y de que utiliza la distribución student-t, la cual la hace apropiada cuando el número de muestras es pequeño y la desviación estandar es desconocida.

A lo que nos referimos con la función simétrica es al cálculo de la pérdida Kullback-Leibler donde en lugar de minimizar dos probabilidades condicionales, minimizamos una donde son simétricos los valores, siendo $P_{i,j} = P_{j,i}$.

$$P_{j|i} = \frac{exp\left(- \parallel x_i - x_j \parallel^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} exp\left(- \parallel x_i - x_k \parallel^2 / 2\sigma_i^2\right)} \tag{3.6}$$

El proceso normal de optimización de t-SNE es maximizar una función de costo, en éste caso la Kullback-Leibler, donde se calculan probabilidades condicionales de los datos en el eje horizontal así como en el eje vertical, buscando encontrar los vecinos más cercanos en el proceso ya que la probabilidad condicional se ve incrementada cuando los vecinos están más cerca.

El algoritmo simple de t-SNE lo encontramos en el algoritmo 1, donde X es el conjunto de datos de entrada de n elementos, Y siendo la salida y tiene la misma cantidad de elementos, pero tiene una dimensionalidad menor a X. Tenemos T como número de iteraciones, η como tasa de aprendizaje y $\alpha(t)$ como el momentum en el tiempo t.

$$\begin{aligned} Perp(P_i) &= 2^{H(P_i)}, siendo \\ H(P_i) &= -\sum_j p_{j|i} log_2 pj|i \end{aligned} \tag{3.7}$$

$$q_{ij} = \frac{\left(1+ \parallel y_i - y_j \parallel^2\right)^{-1}}{\sum_{k \neq l} \left(1+ \parallel y_k - y_l \parallel^2\right)^{-1}} \tag{3.8}$$

$$\frac{\delta C}{\delta Y} = 4 \sum_j \left(p_{ij} - qij\right)\left(y_i - yj\right)\left(1+ \parallel y_i - y_j \parallel^2\right)^{-1} \tag{3.9}$$

3.2. Inteligencia artificial para reconstrucción

Hoy en día existen ciertos tipos de redes neuronales que son estado del arte para generación/reconstrucción de datos, por lo cual es de suma importancia el explicarlos así, pero más que nada, explicar las bases necesarias para poder entenderlos. En ésta sección abordaremos dichas bases así como detalles relevantes al trabajo aquí presentado.

Algorithm 1 Algoritmo t-SNE original

Computar afinidades $p_{j|i}$ aplicando perplejidad (ecuación 3.7) utilizando la ecuación 3.6

$p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$

Inicializar $Y^{(0)} = y_1, y_2, \ldots, y_n$ de una distribución $N(0, 10^{-4}I)$

for t = 1 to T **do**

Computar afinidades de bajas dimensiones q_{ij} usando la ecuación 3.8

Computar el gradiente $\frac{\delta C}{\delta Y}$ usando la ecuación 3.9

Hacer $Y^{(t)} = Y^{(t-1)} + \eta\frac{\delta C}{\delta Y} + \alpha(t)\left(Y^{(t-1)} - Y^{(t-2)}\right)$

end for

3.2.1. Inteligencia artificial y tipos de aprendizaje

La inteligencia artificial es un campo de las ciencias de la computación que se enfoca en hacer máquinas, y más que nada programas, inteligentes, esto tomando como base el conocimiento y experiencia humanos, pero sin limitarse a la capacidades de éste (22). Para ello, hace uso de otras áreas de la ciencia como las matemáticas, y en específico la estadística una rama de ésta.

La forma en cómo los modelos de inteligencia artificial aprenden tiene que ver, en primera instancia, con el tipo de datos de entrada que tenemos, ya que pueden ser datos etiquetados (explicado más adelante) o no etiquetados, lo que nos llevará a tomar ciertas estrategias en forma de tipos de aprendizaje: superviado y no supervisado. Estos tipos de aprendizaje se analizan en las siguientes subsecciones.

3.2.1.1. Aprendizaje supervisado

Con aprendizaje supervisado, de acuerdo a (23), nos referimos al tipo de aprendizaje donde el ajuste de parámetros se realiza utilizando datos etiquetados. Con etiquetado nos referimos a datos que indiquen la clase/categoría a la que pertenece el dato, o bien, qué resultado genera al procesar esos datos. Ambos se pueden considerar salidas: clasificación y regresión repectivamente.

El saber las etiquetas ayuda a encontrar un modelo que tendrá resultados más cercanos a lo deseado, además evitamos *overfitting* (termino explicado posteriormente) al utilizar grandes conjuntos de datos.

Tenemos 3 fases que se pueden identificar dentro del aprendizaje supervisado:

1. **Entrenamiento**: Nos ayuda a encontrar los parámetros y nos indica una medida cuantitiva del modelo, por ejemplo, MSE, MRSE, etc...
   Al final de ésta fase, después de una cantidad fija de iteraciones, se seleccionan

los mejores parámetros, los cuales se obtienen a través de múltiples iteraciones de entranar el modelo actualizando pesos, todo esto utilizando un conjunto de datos con nombre homólogo a ésta fase (conjunto de entrenamiento).

2. **Prueba**: Nos sirve para confirmar si el modelo fue entrenado con los mejores parámetros al finalizar la fase de entrenamiento, ya que ésta fase (prueba) suele ocurrir justo al terminar la anterior, repitiéndose ambas cuantas veces se especifique. El qué tan bien fue entrenado el modelo se corrobora utilizando un conjunto de datos de entrada diferente al de entrenamiento (se le suele llamar conjunto de prueba). Al introducir el conjunto de entrada obtendremos una sailda que usamos para obtener métricas (dependerá del caso la métrica utilizada) y con ello probar su aptitud.

3. **Validación**: Confirmamos qué tan bueno es el modelo completamente entrenado y viendo cómo reacciona a un conjunto de datos diferente del utilizado (o al menos la combinación usada) en las fases anteriores. Durante éste proceso no se actualiza peso alguno, sólo se evalúa el modelo (mayormente conociendo su exactitud).
   A ésta fase a veces se le puede considerar como "probar el modelo antes de probarlo realmente", ya que podemos saber qué tan bien funcionaría en un entorno real, aún si someterlo a uno.

En la fase de prueba el resultado, a la salida del modelo, no se usa para entrenarlo, sólo para monitorear el progreso y eficacia del entrenamiento.

Los datos para éstas fases se suelen obtener de bases de datos (idealmente grandes), donde se utiliza una división del conjunto de datos total, utilizando un porcentaje alto (60 % o más) para la fase de entrenamiento y el resto para las demás fases, siendo la fase de prueba la que suele usar la mayor parte del resto (10 % en adelante del conjunto total), el resto va para la fase de validación (suele estar cerca del 10 %). No existe una convención para seleccionar el porcentajes del conjunto de datos original para cada fase; depende mucho de cada proyecto.

Tenemos dos modos de entrenamiento de los modelos respecto a cómo entran los datos al modelo para ser entrenados:

- **Por lotes**: En éste modo, el entrenamiento hace uso de conjunto de datos a la vez. Con esto podemos obtener datos estadísticos de los datos (al menos del conjunto en cuestión) en ese instante, lo cual podemos aprovechar en cálculos dentro del modelo.

- **En línea**: En éste modo, el entrenamiento se hace dato por dato. Respecto al modo anterior, perdemos los datos estadísticos obtenidos en conjunto pero cada paso de entrenamiento es más rápido. Éste tipo de entrenamiento se utiliza mayormente cuando no se tienen recursos suficientes para alimentar por lotes al modelo.

3.2.1.2. Aprendizaje no supervisado

En el aprendizaje no supervisado, de acuerdo a (24), no tenemos datos etiquetados, sino simplemente una base de datos, que suele ser grande, con datos crudos. Si bien con el pasar de los años vamos encontrando más y más datos dentro de la internet y otras fuentes, el hecho es que la gran mayoría de ellos llegan a ser no etiquetados, y para poder etiquetarlos es necesario invertir grandes cantidades de tiempo y/o dinero. El objetivo del aprendizaje no supervisado, a falta de etiquetas, se aleja de clasificación y regresión de los datos y se centra más en encontrar patrones en los mismos.

Asimismo, cabe señalar que existe algo llamado "semi-supervisado", el cual combina datos etiquetados y datos no etiquetado, con la particularidad que los datos no etiquetados representan una gran mayoría respecto a los datos etiquetados, esto nuevamente debido a la problemática de etiquetar los datos. Pues bien, en éste tipo de aprendizaje se suelen utilizar los datos no etiquetados para encontrar los patrones de los datos, mientras que los etiquetados se utilizan para darle significado a los patrones encontrados.

3.2.2. Machine Learning

De acuerdo a (24) machine learning "es una rama de la inteligencia artificial que se enfoca en permitir a las máquinas desarrollar su trabajo de forma habilidosa mediante el uso de software inteligente". El software inteligente del que hace referencia está constituido por algoritmos y modelos que se encargan de realizar tareas que un software promedio no podría hacer, (25), y ésta tareas son tareas que son demasiado complejas para programar, o bien, tareas que están más allá de la capacidad del ser humano para ser ejecutada. Dentro de las primeras nos encontramos con el reconocimiento de voces y manipulación de imágenes, mientras que en el segundo nos encontramos con procesos donde se requiere procesar una enorme cantidad de datos para poder extraer del mismo un conocimiento como detección de patrones dentro de bases de datos.

3.2.2.1. Support Vector Machines (SVM)

De acuerdo a (25), las máquinas de soporte vectorial (SVM, de aquí en adelante) son un modelo de aprendizaje supervisado y un algoritmo, como menciona (23), que comúnmente se utilizan para la clasificación de conjuntos de datos (aunque pueden ser utilizadas para regresión como menciona (26), pero no nos centraremos en estos), las cuales pueden ser en dos dimensiones, que es lo más común, pero también pueden aplicarse a altas dimensiones.

Podemos clasificar a los SVM en dos, como lineales y no lineales. Los lineales, representados por la ecuación 3.10, son aquellos cuyo método de clasificación utiliza una función lineal (como una línea o hiperplano) para separar los datos. Los no lineales, representados por la ecuación 3.11 utilizan una función no lineal, como una radial, para

este propósito.

$$f(x) = wx^{'} + \gamma \tag{3.10}$$

$$f(x) = w\phi\left(x^{'}\right) + \gamma \tag{3.11}$$

La forma de entrenarlos como un clasificador, considerando la versión en dos dimensiones, involucra encontrar dos vectores de soporte (de ahí el nombre) que actúan como márgenes antes y después de la línea/hiperplano central. De esta manera, el hiperespacio es separado en dos subdominios D_1 y D_2 (ecuación 3.12), uno por cada clase. Podemos nombrar a los puntos caídos de un lado con la etiqueta -1 y los del otro con la etiqueta 1. Así podemos ver el objetivo de parametrización como lo que tenemos en la ecuación 3.13.

$$\begin{aligned} D_1 = x : wx^{'} + \gamma \leq 0 \\ D_2 = x : wx^{'} + \gamma \leq 0 \end{aligned} \tag{3.12}$$

$$\begin{aligned} x : wx^{'} + \gamma = 1, x \in D_1 \\ x : wx^{'} + \gamma = -1, x \in D_2 \end{aligned} \tag{3.13}$$

donde w es la pendiente de los vectores de soporte, γ es el vector de *bias*, mientras $x^{'}$ es la matriz, transpuesta, de vectores de entrada al modelo.

Los vectores de soporte se calculan maximizando la distancia entre ellos (d) a la vez que se minimiza el error de predicción (23). Esta distancia se rige por la ecuación 3.14.

$$d = \frac{\pm 2}{\parallel w \parallel^2} \tag{3.14}$$

Si a ésta le aplicamos raíz cuadrada y dividimos entre dos ambos lados, obtenemos la ecuación 3.14.

$$\frac{d^2}{2} = \frac{1}{\frac{\|w\|^2}{2}} \tag{3.15}$$

Se puede observar que es más conveniente minimizar $\frac{\|w\|^2}{2}$ que maximizar $\frac{d^2}{2}$. Por ende, en la ecuación 3.16 tenemos el error para dos dimensiones.

$$1 - y\left(wx^{'} + \gamma\right) \tag{3.16}$$

Así pues, la ecuación 3.17 presenta el mecanismo de optimización seguido:

$$\begin{aligned} \underset{w,\gamma}{Minimizar} : \frac{\|w\|^2}{2} \\ sujeto\ a : y\left(wx^{'} + \gamma \geq 1\right) \end{aligned} \tag{3.17}$$

Una vez entrenado el modelo, se puede utilizar para poder realizar una clasificación sobre un conjunto de datos. Cabe mencionar que, como otros modelos, no será perfecta la clasificación y algunos vectores podrían dar una clasificación errónea ya que puede que esos vectores estén del lado erróneo de la línea de clasificación (la que divide el hiperespacio en 2 o más), o están dentro de los márgenes de los vectores de soporte. Esto último es inevitable ya que es muy probable que queden vectores dentro de los márgenes al momento de entrenar.

Un ejemplo de separación del espacio por medio de SVM se muestra en la figura 3.1

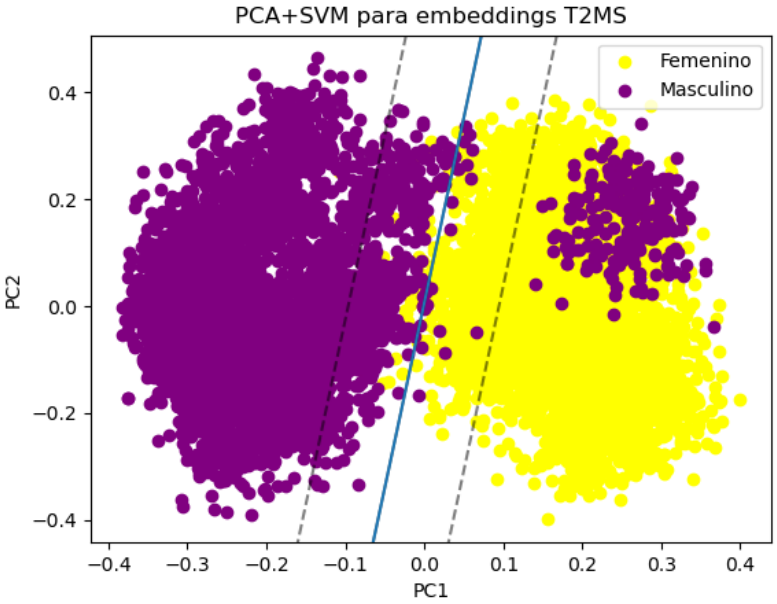

Figura 3.1: Ejemplo de separación del espacio utilizando SVM

3.2.2.2. Clustering

De acuerdo a (27), llamamos *clustering* a la acción de agrupar datos en conjuntos que nos resulten significativos según nuestras necesidades. Éste método se considera un tipo de aprendizaje no supervisado dado que no es necesario tener las etiquetas de los datos ya que sólo se busca agruparlos, sin embargo, se pueden utilizar de forma de aprendizaje semi-supervisado para encontrar primero los agrupamientos de datos y luego darles significado.

Sin embargo, un conjunto de datos puede tener múltiples formas de agrupamiento, por lo que es de suma importancia el tener esto en cuenta para poder elegir un modelo

de *clustering* que satisfaga nuestros requerimientos a la hora de elegirlo. El método para encontrar los *clusters* varía dependiendo del algoritmo de *clustering* utilizando, sin embargo, la gran mayoría de ellos utilizan una función objetivo, la cual puede ser una función de distancia, similaridad, etc.

Como menciona (25), también se puede considerar al *clustering* el acto de poner datos similares juntos mientras que disimilares se encuentren separados. Pero dado que dependiendo del contexto y alcance de éste concepto, podría ser incluso contradictorio y por ello es necesario tener en mente el alcance y contexto de qué es diferente en el conjunto de datos que se agrupará.

3.2.2.3. K-Means

Tal como menciona (25). K-means es un algoritmo de *clustering* de datos, el cual busca generar k *clusters* donde serán asignados (no necesariamente equitativamente) los n vectores del conjunto X de entrada.

Cada *cluster* lo identificamos por el centroide μ_i al que todos sus vectores pertenecientes son más cercanos. El término 'cercano' en éste caso depende del contexto de a qué hagamos referencia como distancia entre 2 puntos. La función/definición de distancia más utilizada es la euclidiana, aunque puede utilizarse una distancia no euclidiana como la utilizada para poder encontrar la distancia entre dos puntos dentro de una esfera, la cual está definida en la ecuación 3.18.

$$d = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.18}$$

El algoritmo con el cual generamos los *clusters* de K-means se encuentra en 2.

Algorithm 2 Algoritmo K-means

Inicializar aleatoriamente centroides $\mu_1, \mu_2, \ldots, \mu_k$

Inicializar en conjutnos vacíos los *clusters* $C_1, C_2, \ldots, C_k$

while No se llegue a convergencia **do**

  for k = 1 to K **do**

    $C_k = \{x \in X : k = argmin_j \parallel x - \mu_k \parallel\}$

  end for

  for k = 1 to K **do**

    $_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$

  end for

end while

Dados k *clusters* a crear, y X siendo el conjunto de datos, el primer bucle *for* asigna los x vectores de X a un *cluster*, mientras que el segundo bucle *for* encuentra el nuevo centroide respecto a la media de los elementos del *cluster*.

No siempre se puede encontrar el óptimo, ni siquiera local, por ello se suele ejecutar varios veces para poder encontrar el mejor dentro de varias corridas, aquí la convergencia dependerá de criterio propio y la definición de la misma bajo esto.

3.2.3. Deep Learning

El aprendizaje profundo (*deep learning*) tiene como base el uso de redes neuronales para la generación de los modelos que se utilizan para encontrar las funciones que generen las salidas deseadas dadas las entradas proporcionadas.

Una red neuronal es una aproximación (no precisa, además de 'primitiva' ya que existen aproximaciones más novedosas) que los humanos hemos creado para poder imitar el comportamiento del cerebro humano y su procesamiento mediante el uso de neuronas artificiales que solemos modelar, en su concepto más básico, como unidades que se conectan a otras neuronas, se transmiten información entre ellas modulando la información que se pasaron, además de utilizar una función de activación que permite a cada neurona el decidir si la información que le llegó a través de las demás neuronas podrá propagarse más allá de esa neurona.

3.2.3.1. Bases y modelos

Redes Feed-forward En el ámbito de redes neuronales el exponente más conocido, y por uno de los más sencillo, son las redes *feed-forward* que se pueden ver, como indica (25), como un grafo acíclico dirigido que podemos descomponer en capas, donde una capa es un conjunto de nodos de una misma altura y que llevan el mismo rol entre ellos. Aquí podemos identificar 3 capas principales: capa de entrada, capas ocultas y capa de salida, y que podemos en la figura 3.2.

- La capa de entrada es una serie de nodos que recibirán la entrada general de la red y pasarán los datos hacia las primeras capas de las capas ocultas.
- Las capas ocultas se encargarán de procesar los datos de entrada y "aprender" de la misma (posteriormente se ahondará en el tema), arrojando la entrada procesada hacia la capa de salida.
- La capa de salida se encarga de generar la salida del sistema y puede ser desde una simple neurona hasta un conjunto de las mismas según sea necesario para el problema a tratar.

Multi-Layer Perceptron (MLP) es un tipo de red *feed-forward* donde las salidas de una neurona entregan información hacia todas las neuronas de la siguiente capa, y por ende cada neurona recibe información de todas las neuronas de la capa anterior, por

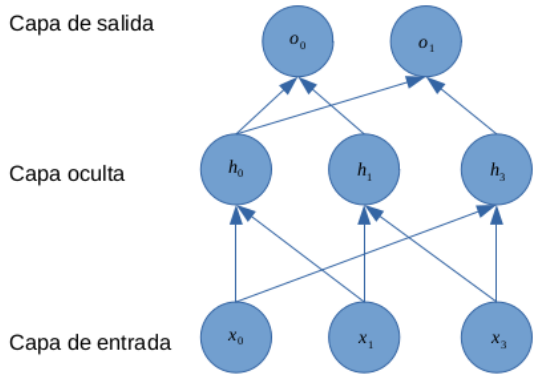

Figura 3.2: Ejemplo de una red $feed - forward$ *simple*.

esto mismo se les puede llamar redes *fully-connected* debido a la propiedad que tienen de conectarse por completo entre ellas, como podemos ver en el ejemplo de la figura 3.3. Aunque cabe considerar que esto no afecta a las capas de entrada y salida dónde sus entradas y salidas (respectivamente) no van hacia otras neuronas, sino hacia el exterior (entrada y salida del modelo).

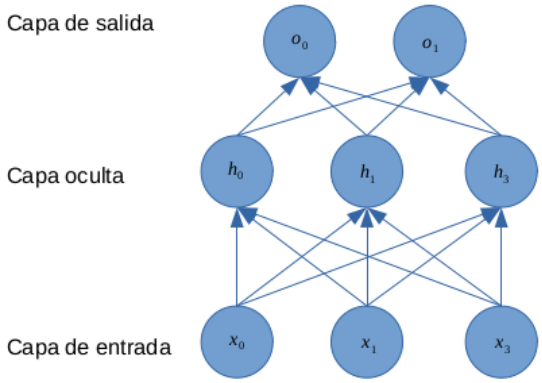

Figura 3.3: Ejemplo de una red *fully-connected* simple.

Podemos ver, de acuerdo a (28), que cada una de éstas capas se puede ver, con notación de álgebra lineal, como una matriz de pesos W y una de sesgos b (*bias* de aquí en adelante). Éstas matrices al tener como entrada una matriz X nos genera una salida O. Normalmente podría verse la salida de una capa como la multiplicación matricial de la entrada por los pesos más el bias ($O = XW + b$, con X siendo la matriz de entrada), pero si aplicamos esto de manera secuencial utilizando la salida de una capa como la salida de la otra, llegamos a lo que se muestra en la ecuación 3.19, lo que termina siendo

otra función lineal.

$$O = (XW^{(1)} + b^{(1)})W^2 + b^{(2)} = XW^{(1)}W^2 + b^{(1)}W^{(2)} + b^{(2)} = XW + b \tag{3.19}$$

Esto ocurre porque la salida de una capa anterior genera una salida lineal y al mantener éste patrón sólo podremos aprender una función lineal en nuestro modelo, lo cual no es deseable. Para evitarlo utilizamos las llamadas **funciones de activación** que transforman la linealidad de entrada (x) en una no-linealidad. Las más famosas son ReLU y sigmoide (se puede ver su comportamiento en las imágenes 3.5 y 3.4), pero se puede utilizar cualquiera que pueda generar una salida, normalmente, en un rango entre -1 y 1 o 0 y 1, y que sea derivable.

En el caso de ReLU (Rectified Linear Unit), la salida tiene un rango de [0, x]. Véase ecuación 3.20

$$\frac{1}{1 + e^{-x}} \tag{3.20}$$

En el caso de sigmoide, la salida tiene un rango de [0, 1]. Véase ecuación 3.21

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.21}$$

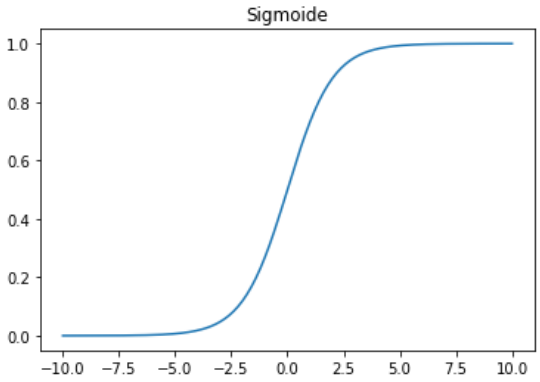

Figura 3.4: Función de activación sigmoide.

Retropropagación: Durante el entrenamiento, se aplica lo que se conoce como propagación hacia adelante (*forward*) que consiste en introducir una entrada a la red neuronal y encontrar su salida de acuerdo a los cálculos de la red neuronal. Ésta salida tendrá un diferencia respecto a la salida esperada dada esa entrada, lo cual genera un error, mismo error que se aprovecha para actualizar los pesos y bias de las red al encontrar la aportación de cada uno de ellos al resultado y actualizándolos, esto se genera utilizando una derivación de los valores y justo es por esto que la función de activación debe de ser derivable.

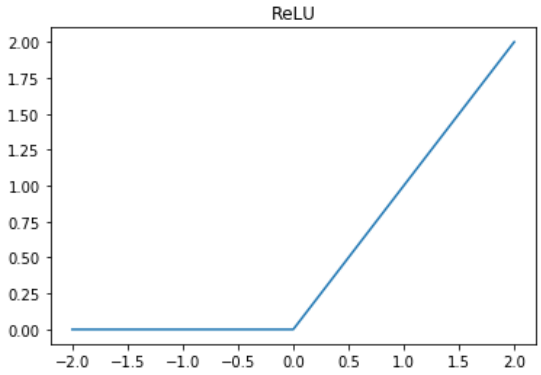

Figura 3.5: Función de activación ReLU.

Regularización: Un algoritmo es estable si un cambio pequeño en su entrada no genera un cambio enorme en la salida. La estabilización del algoritmo ayuda, además, a prevenir el *overfitting*, el cual se puede considerar como una imposibilidad del modelo para poder generalizar, lo cual resulta en que sus salidas estén sesgadas y por ende erróneas. El modelo va aprendiendo con cada vez que aplicamos retropropagación, sin embargo, si la cantidad de datos de entrada no es muy grande o no es demasiado variada, puede ocurrir *overfitting*.

Como se ve en (28), esto se puede aliviar/reducir al utilizar métodos de regularización como “decaimiento de pesosütilizando unas normas de regularización, así como el uso de "Dropouts”. Dentro de las normas de regularización, las más famosas y utilizadas son las normas $\ell 1$ y $\ell 2$ las cuales ayudan durante el entrenamiento a minimizar el error de predicción al agregar penalizaciones a los pesos y/o activaciones y con ello evitar que estos se disparen durante entrenamiento. Dropout se puede ver como activación y desactivación aleatoria de conjuntos de neuronas, durante el entrenamiento, esto ayuda a mejorar el aprendizaje al no tener siempre activas todas las neuronas y hacer más robusto el modelo dando mejor generalización al aprender diferentes patrones.

Redes Neuronales Convolucionales (CNN) Las redes neuronales convolucionales, CNN a partir de ahora, son un tipo de redes neuronales que, como lo menciona (25) utilizan convoluciones en al menos una de sus capas. Para poder entender éstas redes primero es necesario explicar qué es una convolución. Pues bien, una convolución es una operación matemática que puede ser en dominio continuo o discreto y que dadas dos funciones, se obtiene el producto de ellas donde una de ellas está desplazada. Para el caso discreto se utilizan arreglos en lugar de funciones, pero ambos arreglos de entrada deberán tener la misma dimensionalidad.

Para el caso continuo: Ésta operación suele utilizarse mucho en el área de señales, por ejemplo en la transformada de Fourier. La convolución en éste domino termina siendo una integración, sobre uno o más ejes, del producto de una señal por otra donde

la segúnda se desplaza, lo cual se puede ver en la ecuación 3.22.

$$s(t) = f(t * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{3.22}$$

Para el caso discreto: esta operación suele utilizarse mucho en el procesamiento de imágenes al aplicar filtros a las imágenes para poder generar resultados tales como eliminación de ruido, difuminación, etc. La operación es similar a la del caso continuo, sólo que en lugar de ser una integral es una suma de productos como podemos verlo en la ecuación 3.23.

$$s[n] = (f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{3.23}$$

En términos de CNN, f es la matriz de entrada, mientras que a la matriz g se le suele llamar *kernel*, la cual es una matriz con valores que sirven para generar una salida dada la convolución (a ésta salida la llamamos *feature map*) y que en procesamiento de imágenes es el filtro que se mencionó anteriormente. Pues bien, en las CNN los *kernels* son los que suelen ser aprendidos mientras se entrena la red.

Para el caso de convoluciones 2D tenemos la fórmula 3.24, pero ya que la convolución presenta la propiedad de conmutación, se puede ver como la ecuación 3.25, ésta última se puede implementar más fácilmente, como menciona (25) en una biblioteca de machine learning, pero también algunas bibliotecas utilizan una operación similar llamada correlación cruzada la cual es una convolución pero sin invertir el *kernel*, como se ve en la ecuación 3.26.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.24}$$

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3.25}$$

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.26}$$

Redes Neuronales Recurrentes (RNN) Éstas redes son utilizadas para conjuntos de datos donde se puede atribuirle una propiedad de secuencia ("series de entidades donde el orden importa" (29)) y se quiere aprovechar ésta propiedad para encontrar información entre los datos además de sólo información de cada dato de forma aislada. Ejemplo de conjuntos de datos donde exista ésta propiedad son aquellos de señales de voz, vídeos, series de tiempo y lenguaje natural. Ésta última siendo uno de los máximos exponentes que utilizan RNN, como sus más actuales arquitecturas que son estado del arte, como GPT-3 (30).

Las redes neuronales recurrentes (RNN) son un tipo de red neuronal que, a diferencia de las redes vistas anteriormente, aprovechan información obtenida de un *forward* anterior (ya que almacena información de uno o más pasos anteriores, según longitud, en lo

que se le llaman estados ocultos), así como también pueden utilizarse salidas anteriores para realizar sus cálculos.

En éste tipo de redes neuronales se utilizan como unidades básicas de procesamiento unidades recurrentes, las cuales tienen internamente un unidad oculta donde se almacenan los estados ocultos y la cual es recurrente (su salida se utiliza para retroalimentar la entrada). Éstas redes se pueden ver como se muestra en la figura 3.6, lado izquierdo, donde se puede ver cómo existe una retroalimentación con la flecha que va de y llega hacia la unidad oculta pero, podemos además "desdoblar" las redes de forma que se vean más similares a cómo solemos ver las redes neuronales normales como se muestra en la figura 3.6, lado derecho, donde veremos que se puede ver una secuencia de neuronas comunes donde las entradas, salidas y estados ocultos tienen un superíndice ^{t} indicando que cada una de ellas se encuentra haciendo el cálculo en un instante de tiempo diferente.

En la figura 3.6, lado derecho, podemos apreciar mejor los estados ocultos dentro de las unidades recurrentes. Los estados ocultos se van actualizando en cada *forward*, sin embargo, sus pesos se actualizan en entrenamiento y aún cuando la forma desdoblada haga olvidarnos de que son un solo módulo, todos estos estados ocultos pertenecen a una misma unidad por lo que comparten pesos. En el *forward*, el cálculo del estado oculto $h(t)$ se realiza con la ecuación 3.27, donde podemos apreciar el que utiliza un estado oculto anterior.

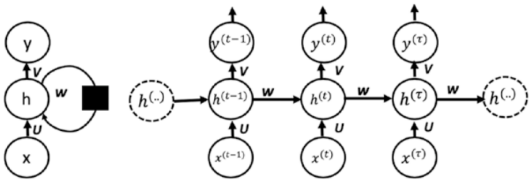

Figura 3.6: Ejemplo de una simple RNN. A la izquierda se muestra en su forma doblada, y a su derecha en su forma desdoblada (a lo largo del tiempo). Imagen tomada de (2)

$$h(t) = \sigma\left(h(t-1), x(t)\right) \tag{3.27}$$

Dentro de las RNN existen dos problemas que hacen difíciles de entrenar a las RNN e incluso inviables para su uso directo, y estos problemas son el desvanecimiento y explosión del gradiente. Estos se dan debido a que al hacer retropropagación, el gradiente se puede hacer o muy grande o muy chico durante la actualización del mismo entre mayor sea la longitud desenrrollada del modelo, como se puede ver en la figura 3.7, haciendo que los gradientes se vuelvan prácticamente 0 o se disparen (desvanecimiento y explosión respectivamente). Para lidiar con éste problema existen artilugios como el *gradient clipping* (29) que lo que hacen es evitar que los gradientes se alejen de cierto umbral donde deberán de estar, sin embargo esto añade otro hiperparámetro más al entrenamiento, pero considerando que ganamos estabilidad al utilizarlo, es un intercambio ventajoso.

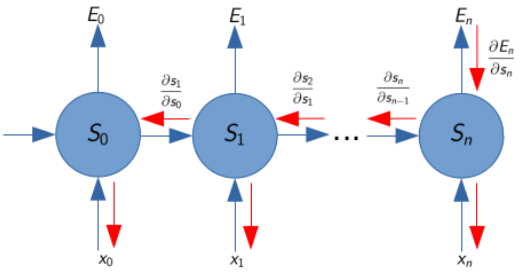

Figura 3.7: Retropropagación de RNN.

A continuación se enlistan un par de RNN que tratan con la explosión y desvanecimiento de gradiente utilizando nuevas características:

- Long ShortTerm Memory (LSTM). De acuerdo a (29), las RNN simples son propensas a ruido y a ser costosas, computacionalmente hablando, para entrenar. Sin embargo, las LSTM son redes que además de evitar estos problemas, capturan mejor las dependencias de término corto y largo, lo que permite un mejor aprendizaje de la secuencia que se está analizando.
  La diferencia en arquitectura que tienen éstas redes con las RNN tradicionales es que, además de utilizar unidades para procesar el estado oculto, se utilizan otras llamadas celdas de estado, las cuales conectan informacion a través de la red, la cual tiene que ver con la entrada y estados anteriores. El estado de la celda se calcula utilizando tres compuertas: entrada, salida y olvido. La primera nos indica qué tanto de la entrada se retendrá en la celda de estado, la segunda determina qué tanto afectarán las celdas de estado a la salida y la última determina qué tanto de la celda anterior se utilizará. Una ventaja que ganamos con ésta configuración es que evitamos la explosión y desvanecimiento del gradiente debido a éstas compuertas que regulan la cantidad de valor que se propagarán.

- Gated Recurrent Units (GRU). De acuerdo a (2), son un tipo de RNN similar a una LSTM que es incluso más rápida de entrenar, además de que trata la explosión y desvanecimiento del gradiente. Igual que la LSTM, la GRU tiene celdas de estado y compuertas, sin embargo, son diferentes tanto en nombre como en acción. Aquí tenemos únicamente a las compuertas de reinicio y actualización. La compuerta de reinicio controla qué partes del estado computan el siguiente estado, introduciendo una linealidad entre estado pasado y futuro. Y la compuerta de actualización, que a la vez controla el factor de olvido y la decisión de actualización de la unidad de estado (desde copiando o ignorarlo completamente).

Asimismo, como lo enuncia (2), tenemos algunos casos donde tenemos secuencias donde los elementos no sólo dependen de los valores anteriores, sino también de los futuros, para éste tipo de secuencias no nos basta una RNN común para poder representarlas, ejemplos de éste tipo de conjuntos son análisis de voz y Natural Language Processing

(NLP) donde en ambos casos un elemento adquiere significado cuando se analizan los elementos pasados y futuros dentro del conjunto; fonemas para voz, y palabras para NLP. Para ello necesitamos recurrir a la variante bidireccional (RRN bidireccional), en la cual no sólo tenemos una unidad recurrente para poder representar los estados ocultos de *forward*'s anteriores, sino que utilizamos otra unidad para los estados de valores futuros, esto es, una unidad que desdoblada va al "contrario del tiempo" (retroalimentando, *backward*, hacia atrás en lugar de hacia adelante), de ésta manera la salida se computa utilizando estados pasados, futuros y la entrada (y según la arquitectura recurrente utilizada, la salida). Esto lo podemos ver en la figura 3.8, donde h es el estado oculto que guarda información del *forward* y g el estado oculto que guarda el estado del *backward*, así pues pasa datos de pasos a adelante hacia atrás, haciendo más robusto el aprendizaje.

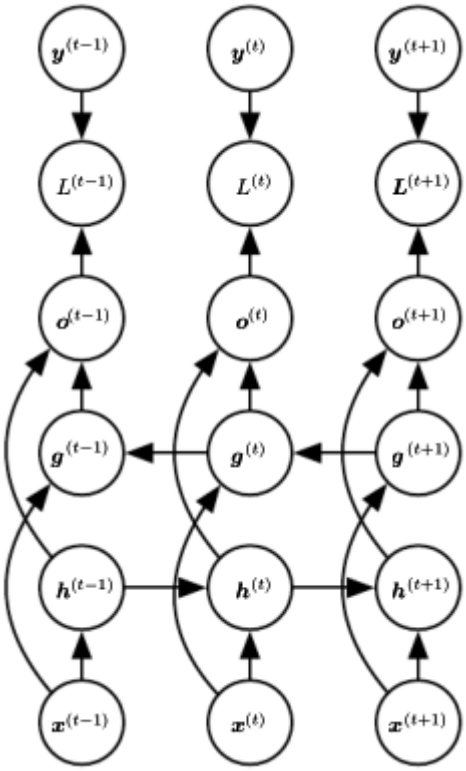

Figura 3.8: Ejemplo de una simple RNN bidireccional. Imagen tomada de (2)

3.2.3.2. Arquitecturas

***AutoEncoders* (AE)** Los *autoencoders* son un tipo de red neuronal simétrica y cuya estructura se puede dividir en dos partes: un codificador y un decodificador. Éstas

redes mantienen una representación intermedia Z que comparten el codificador y el decodificador, siendo salida del primero y entrada del segundo. Viendo el codificador y decodificador como funciones, tenemos $C(X)$ como la función que mapea de una entrada X a la representación intermedia (llamémosla y) y $D(y)$ la función que mapea la entrada y a una salida $X^{'}$, siendo $X^{'}$ la salida final del *autoencoder*. Entonces, la salida del *autoencoder* se puede ver como $X^{'} = D(C(X))$. El objetivo general de un autoencoder es poder generar una reconstrucción $X^{'}$ lo más fidedigna posible de la entrada X, y para esto se minimiza la diferencia de $X^{'}$ respecto a X utilizando algún tipo de pérdida de reconstrucción y retropropagando el error.

Los *autoencoders* pueden utilizarse para diferentes tareas, sin embargo las más utilizadas son las siguientes:

1. Reducción/"eliminación" de ruido en imágenes, como en (31)
2. Generar datos sintéticos utilizando la representación intermedia, como en (32)
3. Reducción de dimensionalidad, como en (33)

En éstos dos últimos se hace uso exhaustivo de la representación intermedia Z, aunque de manera complementaria.
En la generación de datos sintéticos, se entrena para forzar a la representación intermedia a ajustarse a una función de distribución, y ya entrenado el modelo, sólo se genera una entrada de valor aleatorio para obtener una $X^{'}$ sintética.
En la reducción de dimensionalidad seleccionamos una dimensionalidad a la cual queramos que nuestra entrada se reduzca, entrenamos el modelo, y al aplicar el modelo a una entrada X obtendremos una y que sería una proyección de X en un espacio de una dimensionalidad menor.

Derivados de los *autoencoders* tenemos a los *Variational Auto Encoders* (VAE) (34) los cuales son un tipo de modelos generativos que se diferencían de los *autoencoders* en que la salida del codificador se pasa a un vector, el cual a su vez alimenta al decodificador. En éste vector se modela un espacio latente que pueda representar al conjunto de datos de entrada, y al entrenar el modelo se intenta obtener la distribución de probabilidad del espacio latente.

β Variational AutoEncoder (βVAE) Propuesta en (35), ésta arquitectura es una variante de los VAE donde al momento de calcular la similitud marginal de los datos (definida posteriormente en 3.30), inyecta un nuevo hiperparámetro β (de ahí obtiene el nombre) para poder "desenredar" (disentangled en inglés) las variables latentes. Una representación de las variables latentes desenredada es aquella donde las unidades latentes (aquello que nosotros le daríamos significado como una característica intrinseca del conjunto de datos de entrada como tono de voz, timbre, etc..) son menos afectadas (pero no totalmente inmunes) por cambios en más de un factor (variables latentes). Todo esto retomando el trabajo de los *autoencoders* donde en los puntos 2 y 3 se mencionan aplicaciones, y las últimas dos utilizan la representación intermedia de forma

más consciente y directa. Aquí la representación intermedia Z toma el papel principal al ser ésta en la que se enfocan lo esfuerzos.

Partimos de que los VAE se enfocan en aprender la similitud marginal de los datos utilizando la ecuación 3.28 donde x es un conjunto de datos parametrizado por z, el cual es un conjunto de factores generativos latentes. Tenemos aquí a y ϕ que son distribuciones de probabilidad, las cuales corresponden al condificador y decodificador de una VAE, respectivamente.

$$\underset{\phi,\theta}{max}\mathbb{E}_{q_\phi}\left(z|x\right)\left[logp_\theta(x|z)\right] \tag{3.28}$$

$logp_\theta\left(x|z\right)$, que aparece en la ecuación 3.28, lo podemos reescribir como en la ecuación 3.29, donde $D_{KL}(\|)$ es la divergencia Kullback-Leiber la cual se puede ver como una medida de similitud o diferencia entre dos funciones de distribución de probabilidad (2).

$$logp_\theta\left(x|z\right) = D_{KL}\left(q(z|x)\parallel p(z)\right) + \mathcal{L}\left(\theta,\phi;x,z\right) \tag{3.29}$$

Ahora bien, maximizando $L(\theta,\phi;x,z)$ es equivalente a maximizar el límite inferior al objetivo verdadero de la ecuación 3.28, lo cual nos da la ecuación 3.30, la cual es la ecuación de pérdida que hemos de minimizar en las VAEs.

$$logp_\theta\left(x|z\right) \geq \mathcal{L}\left(\theta,\phi;x,z\right) = \mathbb{E}_{q_\phi}\left(z|x\right)\left[logp_\theta(x|z)\right] - \beta D_{KL}\left(q_\phi(z|x)\parallel p(z)\right) \tag{3.30}$$

Aquí es donde entra en juego la variable β, ya que la incrustamos antes de la divergencia Kullback-Leibler (D_{KL}) para controlar la cantidad de información que aporta al cálculo de pérdida. Obteniendo con ello la ecuación de pérdida de las βVAEs en la ecuación 3.31.

$$\mathcal{L}\left(\theta,\phi;x,z,\beta\right) = \mathbb{E}_{q_\phi}\left(z|x\right)\left[logp_\theta(x|z)\right] - \beta D_{KL}\left(q_\phi(z|x)\parallel p(z)\right) \tag{3.31}$$

Ésta ecuación nos recuerda, y de hecho es un principio similar, al principio de cuello de botella de información de la teoría de la información, el cual nos describe una optimización objectivo para maximizar la información mútua entre Z e Y, descartando información reelevante de X. Ésta obtimización se busca con la formula 3.32, donde $I(.;.)$ es la información mútua y β es el multiplicador de Lagrange.

$$max\left[I(Z;Y) - \beta I(X;Z)\right] \tag{3.32}$$

En las βVAE, la priori y posteriori están parametrizadas normalmente a una distribución gausiana normal ($N\left(0,1\right)$) lo cual nos permite reparametrizar las variables latentes en 3.33 lo cual ayuda a la estimación de gradientes con respecto a ϕ durante el entrenamiento, donde cada variable $z_i \sim q_\phi\left(z_i|x\right) = N\left(\mu_i,\sigma_i\right)$ y $\epsilon \sim N\left(0,1\right)$

$$z_i = \mu_i + \sigma_i\epsilon \tag{3.33}$$

También es necesario hacer notar que entre más "desenredamiento" exista entre las variables latentes, la salida tenderá a dar resultados borrosos debido a una limitada capacidad reconstuctiva en z, y lo contrario aplica también, donde variables "enredadas" tienen resultados más atinados. Así que es necesario encontrar un balance entre qué tan atinados queremos los resultados respecto al "desenredamiento" que queremos que tengan las variables. Ya que entre más "desenredamiento" se tenga en z, mejor se puede caracterizar el conjunto de entrada (al tener variables latentes más) y con ello generar datos alterando las variables, donde al parecer las variables "desenredadas" llegan a "aprender" dentro de ellas características que tienen sentido para los humanos como altura de una silla, o ancho de la misma (para una base de datos de sillas).

Generative Adversarial Networks (GAN) Las redes generativas adversarias (GANs), según (36), son un tipo de red neuronales enfocada en la generación de datos mediante el uso de dos modulos: generador (G) y discriminador (D). Mientras que el generador intenta generar muestras lo más parecidas a las muestras con las que se entrenó para engañar al discriminador, este último intenta decidir si la muestra que recibió proveniene del conjunto de datos con el que se le entró o es una falsificación.

Estos dos módulos entran en una competencia, o juego, de dos jugadores minmax (ecuación 3.34) donde se intenta minimizar $1 - logD(G(z))$ y $logD(x)$ de parte del generador y discriminador respectivamente, donde z es un vector que se muestrea de una distribución a priori para el ruido y que se utiliza para poder generar los datos, y x es el vector de entrada.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[logD(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[1 - logD(G(z))\right] \tag{3.34}$$

Éste juego se lleva a cabo, por decirlo de una manera, por turnos, donde primero se entrena el discriminador y posteriormente el generador. El juego termina, idealmente, cuando $p_g = p_{data}$ al llegar un punto donde $D(x) = \frac{1}{2}$, sin embargo (según (2)) esto puede no llegar a suceder, mayormente, cuando G y D están representados por redes neuronales ya que puede no ser convexo 3.35 en éstas situaciones, además de que como ambos están compitiendo por reducir su pérdida, lo hacen a costa del otro. El proceso de entrenamiento de las redes GANs se puede apreciar en el algoritmo 3 donde k es un hiperparámetro.

Conditional Generative Adversarial Network (CGAN) Propuesta en (3), ésta arquitectura nos presenta una variante de las GANs (véase 3.2.3.2). Un problema que surge con éstas redes (las GANs) es que no tenemos control sobre caracteríticas propias del tipo de elemento que se está entrenando en la red. Esto es, si entrenamos una GAN con solamente imágenes de rostros, ésta podrá generar rostros, sin embargo, no será posible tener control respecto a las variaciones de ancho de rostro, color de piel, tipo de labios, etc.

Aquí es donde entran en escena las CGANs, permitiendo tener un cierto control respecto a qué se podrá generar. Esto se realiza tomando en cuenta la etiqueta (y)

Algorithm 3 Algoritmo GANs

for número de iteraciones **do**

for k pasos **do**

Muestresar un mini-batch m de muestras de ruido $z^{(1)}, z^{(2)}, \ldots, z^{(m)}$ de una apriori $p_g(z)$

Muestresar un mini-batch m de ejemplos $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ de una distribución generativa de datos $p_{data}(x)$

Actualizar el discriminador utilizando ascenso de gradiente estocástico

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

end for

Muestresar un mini-batch m de muestras de ruido $z^{(1)}, z^{(2)}, \ldots, z^{(m)}$ de una apriori $p_g(z)$

Actualizar el generador utilizando descenso de gradiente estocástico

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log\left(1 - D\left(-G\left(z^{(i)}\right)\right)\right)$$

end for

del elemento a generar durante el entrenamiento, al hacer que la función de pérdida cambie para ser sensible a la dependencia entre la etiqueta y el elemento de entrada en cuestión. En éste caso cambiamos $D(x)$ por $D(x|y)$ y $D(G(z))$ por $D(G(z|y))$ dentro de la ecuación de pérdida de las GANs (ecuación 3.34), lo que nos lleva a la ecuación 3.35.

De ésta manera es posible generar salidas condicionadas a una etiqueta de entrada, lo cual es súmamente conveniente si tenemos una característica, a veces subjetiva, o una clasificación dentro de los datos de entrada x que alimentarán a la red CGAN.

$$\underset{G}{min}\underset{D}{max}V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[logD(x|y)\right] + \mathbb{E}_{z\sim p_z(z)}\left[1 - logD(G(z|y))\right] \tag{3.35}$$

La estructura de una CGAN simple la podemos ver en la figura 3.9, donde podemos apreciar mejor al vector z que se menciona en el apartado 3.2.3.2. Una vez que tanto el generador como el discriminador son entrenados, podemos tomar el generador y generar datos sintéticos al condicionar la salida a una y específica y generar un z aleatorio basado en su distribución.

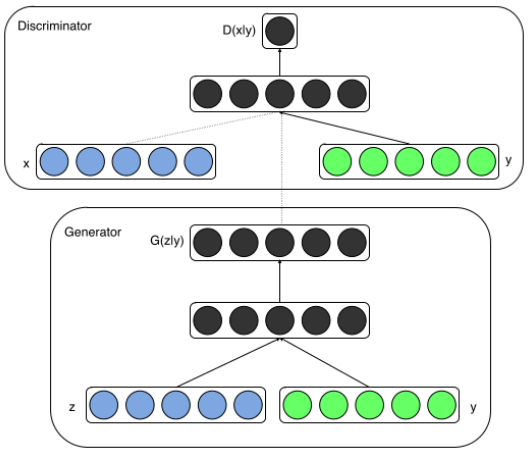

Figura 3.9: Red CGAN simple. Imagen tomada de (3)

Capítulo 4

Metodología

A lo largo de éste capítulo hablaremos sobre los pasos que se llevaron acabo para poder desarrollar éste proyecto. Empezamos con una descripción de los recursos necesarios, tales como la arquitectura T2MS, la cual implementamos de un origen ya exitente mientras explicamos el por qué fue tomado de ahí, así como las bases de datos necesarias para entrenar los modelos, las cuales analizamos obteniendo sus características, posteriormente decidimos cuál de ellas utilizar y por último hablamos de la forma en cómo serían procesados y utilizados los datos dentro del proyecto. Por último, describiremos nuestras propuestas tanto de forma escrita como con ayuda de diagramas a bloques para poder una mejor visualización.

4.1. Implementación de T2MS

Dado que la implementación de T2MS está fuera de los requerimientos de este proyecto, y porque requeriría mucho tiempo para realizarlo, se optó por utilizar una implementación existente. Se encontraron los siguientes candidatos:

1. **ide8**. Implementación realizada por el usuario de github **ide8** cuyo repositorio se puede encontrar en (37). Como característica cabe destacar la inclusión de código para implementar entrenamiento en medio punto flotante (aunque no se activa o desactiva por banderas).

2. **CorentinJ.** Implementación realizada por el usuario de github **CorentinJ** cuyo repositorio se puede encontrar en (38). Como características cabe destacar la inclusión de modelos pre-procesados para los 3 módulos de la arquitectura, cambia WaveNET por WaveRNN y fue entrenada con librispeech.

La implementación **ide8** incluye código para ejecutarse utilizando medio punto flotante, el cual representa datos en coma flotante incluído en el IEEE 754, representando un valor de punto flotante de 16-bits con 1 para signo, 5 para exponente, y 10 para mantiza (39). Al ser un tipo de dato con menos bits pero manteniendo punto flotante,

se pueden realizar cálculos más rápidamente aunque perdiendo precisión. Desafortunadamente, el código no se presta para fácilmente ser modificado y las dependencias necesarias para poder ejecutarlo son considerables, ya que incluso es necesario conseguir una tarjeta de vídeo que tenga características de procesamiento nativas para medio punto flotante, de lo contrario sería necesario simularlo y ello podría tomar más tiempo de procesamiento y configuración de lo necesario, y está fuera del alcance del trabajo actual. Asimismo no se tienen los modelos pre-entrenados de los módulos necesarios.

La implementación **CorentinJ** no tiene los problemas anteriores y presenta una forma simple de instalación y uso, asimismo hace disponibles sus módulos pre-entrenados y funcionales. Por estas razones se eligió esta implementación para ser utilizada.

4.2. Elección de base de datos

Los datos para un algoritmo de inteligencia artificial son una piedra angular, por ello es necesario elegirlos adecuadamente para que cumplan con las necesidades del modelo. En este trabajo se realiza una generación sintética de *embeddings*, pero para obtener estos *embeddings* necesitamos de audios reales, por lo que son necesarios audios de voces humanas. Además de ello, como se generan *embeddings* basándonos en el género de voz, nuestros datos deben de tener etiquetas indicando cuál es el género de la voz que se escucha, y por último, la base de datos debe de tener transcripciones de lo que se dice en los audios. Es necesario, también, considerar que el modelo de T2MS fue entrenado con una base de datos en inglés, y para evitar posibles inconsistencias, lo ideal es utilizar una base de datos que esté en el mismo idioma.

Existen algunas bases de datos que cumplen las características que estamos buscando, entre ellas las más sobresalientes son las siguientes:

	Pros	Contras
Librispeech (40)	▪ Licencia libre (Creative Commons 4, CC by 4.0) ▪ 2500 hablantes ▪ 1000 horas de grabación	▪ Una sola variante de inglés
LJ Speech (41)	▪ Licencia libre (Dominio público)	▪ Una sola variante de inglés ▪ Un solo hablante ▪ 24 horas de grabación
Voxceleb (42)	▪ Licencia libre (Creative Commons 4, CC by 4.0) ▪ Distintas variables de inglés	▪ Requiere registro para obtenerse ▪ 83 horas de grabación ▪ Calidad diferente entre grabaciones al tener distinto origen

Tabla 4.1: Comparacion entre bases de datos utilizadas normalmente para trabajos de IA utilizando voces humanas

De la tabla 4.1, podemos determinar que la más apropiada para éste proyecto es LibriSpeech al ser de libre acceso, tener sólo una sola variante de inglés y tener voces de varios actores de voz de ambos géneros de voz. Además, ésta base de datos se utilizó para entrenar la implementación de T2MS. Por todo esto, se utilizó esta base de datos para entrenar los modelos.

Esta base de datos se compone de diferentes conjuntos de datos: desarrollo, prueba y entrenamiento, los cuales son nombrados de esa forma. Cada uno con una versión de audios 'limpios' y 'sucios', donde la versión con audios limpios se grabó en recintos especiales, como cámaras anecóicas o estudios de grabación, para aislar cualquier otro sonido y otras imperfecciones.

Para éste proyecto se utilizó únicamente la versión 'limpia' ya que no está dentro de los alcances el lidiar ni controlar aspectos como reverberación o ruido, sólo generar *embeddings*. Además, se prevé que utilizar esta base de datos hará que los *embeddings* sean lo más precisos posibles, ya que el ruido y reverberación podrían afectar al mapeo de *embeddings*.

El conjunto de datos que se utilizó es el llamado "dev-clean", el cual es la versión "limpia" del conjunto de desarrollo. Cncontiene aproximadamente 8 horas de grabaciones con audios de 40 personas distintas. Esta cantidad de datos es suficiente ya que es necesario explorar valores de parámetros para varios modelos, así como la arquitectura de los mismos, por lo tanto no es posible utilizar bases de datos más grandes que harían que cada experimento fuese sumamente extenso. Cabe señalar que estas grabaciones sólo

generan 2703 *embeddings* totales (uno por archivo de audio), 2658 si obtenemos iguales cantidades de *embeddings* masculinos y femeninos (ya que hay más voces másculinas que femeninas). Sin embargo, es posible llegar a 21160 *embeddings* totales (21086 al igualar *embeddings* masculinos y femeninos) por medio de, en vez de generar un *embedding* de la grabación completa, generar un *embedding* por cada ventana de audio de la grabación (10 ms de duración con 80 frames de salto y sobrelape de 50).

4.3. Tratamiento de datos

Dado que el objetivo de éste trabajo es generar *embeddings* sintéticos que se basen en un género de voz, se requiere generar una base de datos que contenga solamente *embeddings* de voz y etiquetas del género de voz al que pertenecen. En la figura 4.1 se puede ver el proceso necesario para generarla.

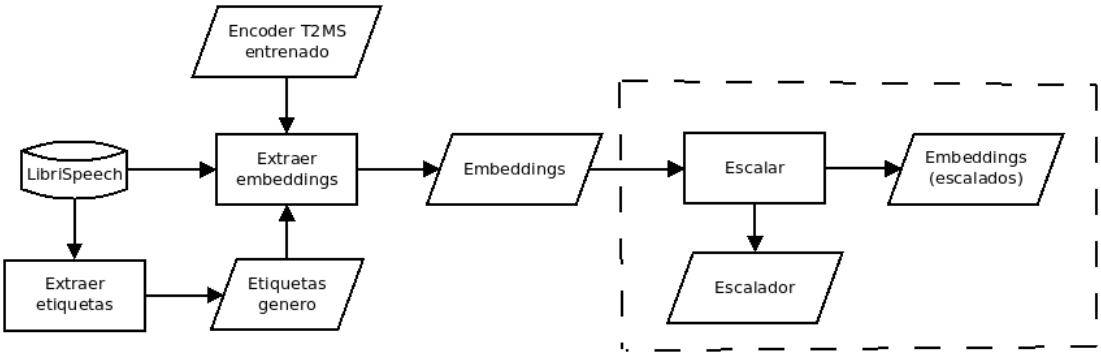

Figura 4.1: Diagrama general de obtención de *embeddings*

Dado que en la base de datos tenemos varios hablantes, existe un archivo por separado que los enlista junto con su género de voz. Éste archivo se encuentra en la raíz del archivo que descarga del sitio oficial de LibriSpeech. Cada hablante tiene múltiples grabaciones, por lo que es necesario asignar cada grabación con un género de voz, para esto se genera un archivo donde se encuentra una relación entre cada ruta de archivo junto con su género de voz, llamado "etiquetas.csv".

Asimismo, el codificador de T2MS genera varios *embeddings* por archivo, dependiendo de cuántas ventanas de 10 ms se encuentran dentro del archivo, permitiendo hacer un aumentado de datos. Se automatiza la generación de la base de datos utilizando la información contenida en 'etiquetas.csv', leyendo uno a uno los registros y generando con el codificador de T2MS los *embeddings* por cada uno archivo, y adjuntando su respectiva etiqueta.

Opcionalmente, se escalan los valores de los *embeddings* para que estén todos en un mismo rango y con ello obtener una nueva base de datos, ahora con los datos escalados. Se escalan para que en todas las dimensiones los valores estén dentro del rango [0, 1], ya que en la base de datos el mínimo es 0 pero no todas las dimensiones tienen el mismo

máximo ya que varía éste valor entre ellas. Asimismo guardamos los datos del escalador para después realizar la opción inversa, donde los datos obtenidos son valores mínimos y máximos de los datos originales. Es importante señalar que este paso proporcionó mejores desempeños con algunos algoritmos, pero con otros no; más adelante se aclarará el uso de este paso en la descripción de cada uno.

Al final, una vez listos los modelos de inteligencia artificial, se generan los audios a voluntad que pueda cumplir nuestro criterio de elección de género de voz. Dada la naturaleza de los modelos, es posible generalizar el uso de los mismos para poder pasar de seleccionar un género de voz a un audio con una voz sintética que sea del género de voz seleccionado. Esto lo podemos realizar siguiendo el diagrama de la figura 4.2. Aquí 'parámetros' dependerá del modelo utilizado, pero donde todos generan un *embedding* sintético (que puede ser escalado), y posteriormente pasarlo a un módulo de generación de audio a partir de *embeddings*.

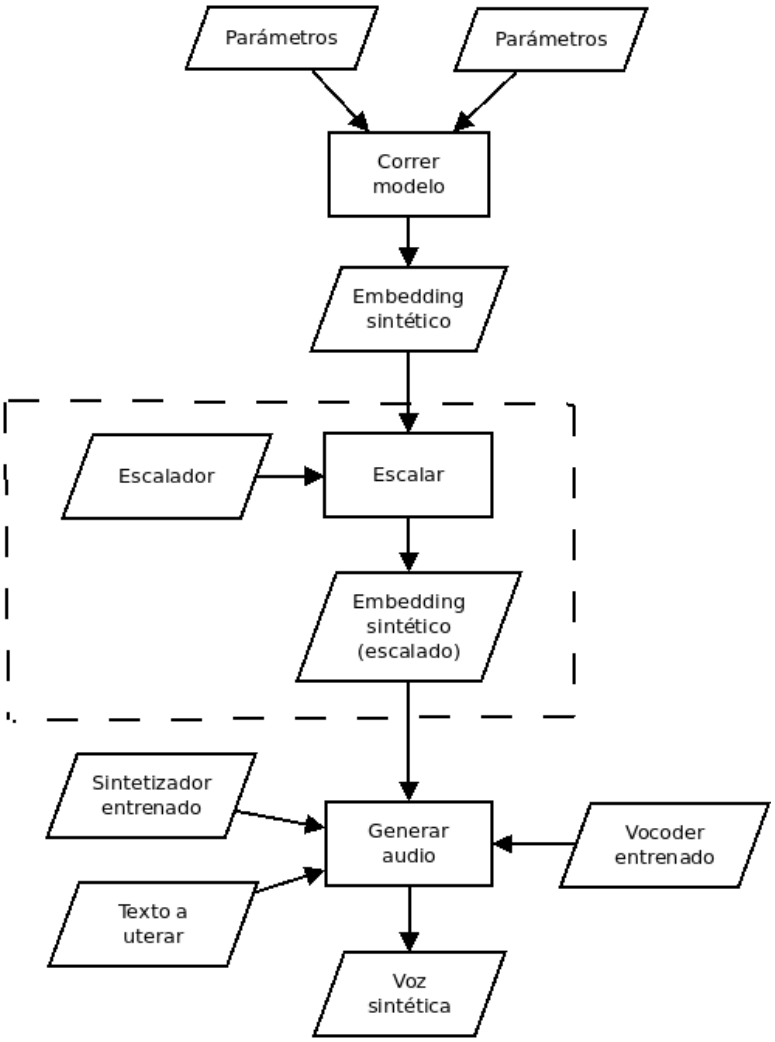

Figura 4.2: Diagrama general de generación de audio en los modelos utilizados en éste proyecto

El último módulo en la figura 4.2 denominado 'Generar audio' es detallado en la figura 4.3. Éste módulo de generación de audio se compone de los módulos *decoder* y *vocoder* de T2MS, por lo que será alimentado con el *embedding* sintético que nosotros le pasemos, el texto a pronunciar, y los modelos pre-entrenados que se obtuvieron de (38)'.

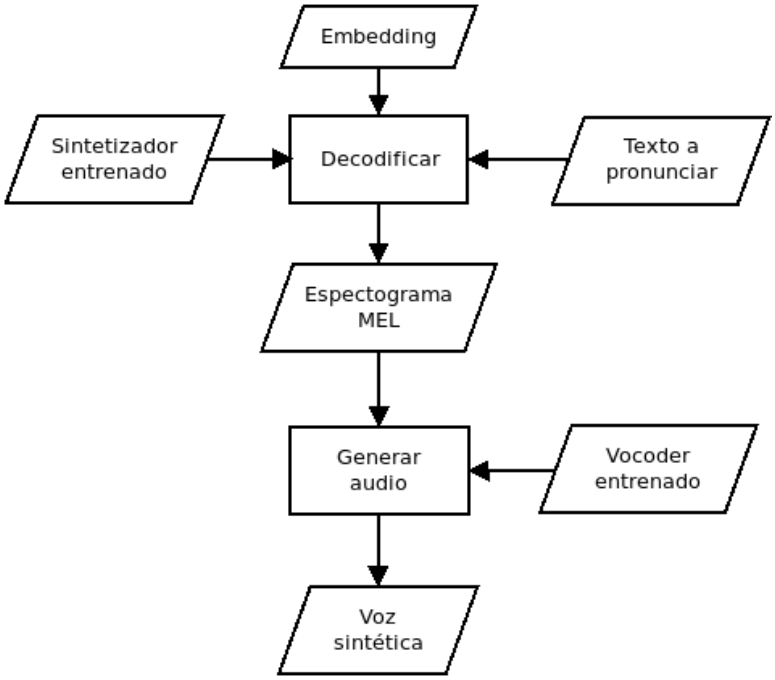

Figura 4.3: Pasos para obtener un audio a partir de un *embedding* usando T2MS

Con todo esto, nos es posible generar audios dado *embeddings* regenerados/reconstruidos utilizando inteligencia artificial donde se podrá elegir explícitamente una etiqueta de género para generarlos, o bien ajustar parámetros (según se prefiera) para generar un *embedding* con más libertad pero manteniendo la premisa de éste proyecto: el decidir el género de voz del cual será la voz que generaremos.

4.4. Propuestas

Considerando lo mencionado en la sección anterior, se generan 3 propuestas que se utilizarán para poder generar *embeddings* de voz sintéticos.

4.4.1. Machine learning

De lo que vimos en el capítulo anterior, sabemos que podemos utilizar diferentes técnicas de reducción de datos para poder visualizar aquellos que tienen alta dimensionalidad en un espacio dimensional más bajo que ayude a que sean entendibles mediante gráficas. Si bien ésta es una de las aplicaciones más utilizadas, también se puede aprovechar ésta característica para poder generar soluciones que no se podían realizar en espacios dimensionales más grandes, o donde simplemente era poco viable (o muy costoso computacionalmente).

Existen otras técnicas que ayudan a la reducción de la dimensionalidad, sin embargo nos enfocaremos solo en dos que justamente fueron presentadas en el artículo (35): PCA y t-SNE. PCA nos ayuda a encontrar los componentes principales en baja dimensionalidad y ayuda a mantener varianza, además de que conviene utilizarlo respecto a ICA (43) ya que al tener los componentes ortogonales, podremos utilizar cada uno de ellos como una dimensión sobre la que actuar. t-SNE se mantiene para realizar pruebas dado que se demostro que puede generar una buena separación de *embeddings* en baja dimensionalidad.

4.4.1.1. PCA

Empezamos con una reducción de dimensionalidad utilizando PCA. Considerando que se pueden hacer visualizaciones más fáciles en 2 dimensiones, y que incluso las reconstrucciones se realizan más fácilmente entre menos parámetros tengan, nos proponemos a utilizar PCA para pasar de 256 dimensiones, que son las nativas de los *embeddings* generados por T2MS, a un espacio de 2 dimensiones.

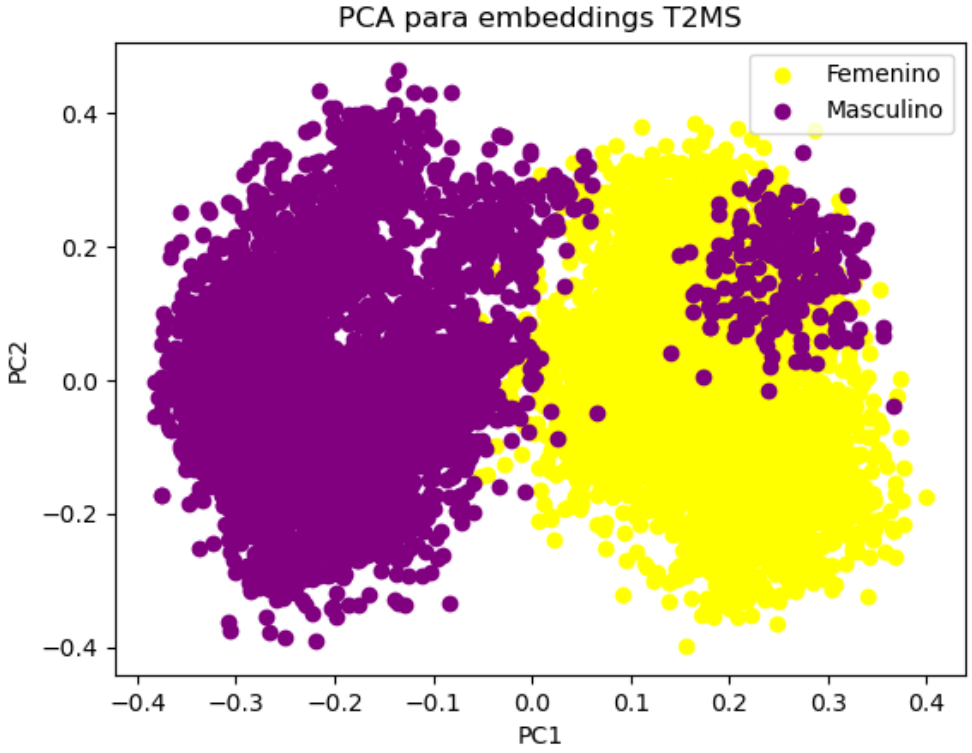

Figura 4.4: Visualización de *embeddings* en un espacio de 2 dimensiones

Lo anterior se puede ver en la figura 4.4, éste espacio de dos dimensiones ayuda

bastante a poder separar los *embeddings* entre el género de voz al que pertenecen, por ello es posible utilizar técnicas que puedan ayudarnos a generar un punto dentro de una distribución de probabilidad donde se genere un punto perteneciente al género de voz preferido. O bien, estimar un hiperplano que separe los *embeddings* para poder generar un punto dentro de un espacio delimitado por el plano y que sea perteneciente al género de voz preferido.

Todo esto nos lleva a dos soluciones: usar SVM para generar el hiperplano, y usar algún tipo de clusterización (en éste caso K-Means debido a la simplicidad del problema).

PCA + SVM. Esta alternativa se basa en utilizar SVM para poder generar un vector de soporte que utilizaremos como hiperplano (en éste caso una línea dado las dimensiones del plano original) para poder dividir el espacio y así obtener dos subplanos dentro del espacio de dos dimensiones, cada uno donde se alojen *embeddings* sobre un género de voz en específico. El proceso completo lo podemos ver en la figura 4.5 donde, para generar el panorama completo, retomamos la reducción de dimensiones y posteriormente utilizamos un clasificador de *embeddings* (SVM) para poder generar el vector de soporte.

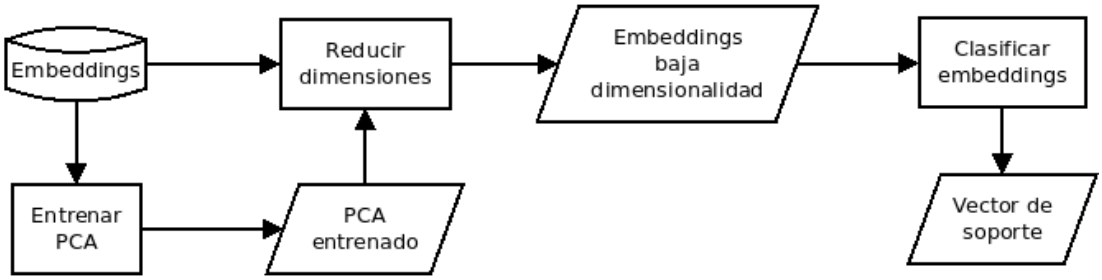

Figura 4.5: Entrenamiento de PCA

Para facilitar las tareas reutilizando código de terceros, utilizamos las implementaciones de PCA y SVM de sklearn (44). Se entrena un módulo PCA con la base de datos de *embeddings* de 256 dimensiones, para después transformarla a una base de datos de 2 dimensiones, a la vez que se mantiene la etiqueta de cada uno de los *embeddings*. Posteriormente se genera un modulo clasificador SVM utilizando ésta última base de datos generada. Una vez entrenado este módulo podemos obtener el vector de soporte que nos servirá para parametrizar en qué sub-espacio están los *embeddings* femeninos y en cuál los masculinos. Los módulos usados aquí ayudarán a la hora de la generación de *embeddings* sintéticos como se puede ver en la figura 4.6.

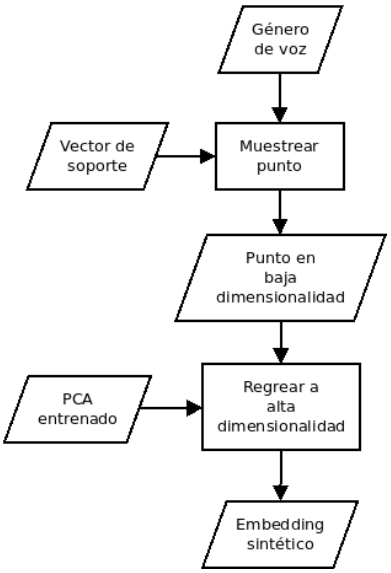

Figura 4.6: Aumento de dimensionalidad utilizando KMeans

Ya que los vectores de soporte (SVM) sólo nos ayudan a dividir el espacio de baja dimensionalidad (2 dimensiones), es necesario regresar a una alta dimensionalidad (256 dimensiones). Para esto utilizamos PCA, apoyándonos del módulo que se entrenó anteriormente. Ya que con PCA lo que hacemos es encontrar una matriz W (2x256) (transpuesta sería W' (256x2)) que, al aplicarse a una matriz de entrada X (1x256), obtenemos una matriz de salida Y (1x2), podemos utilizar la matriz W para hacer la operación inversa. Para esto encontraremos la inversa de ésta matriz y, dado que W es una matriz ortogonal (45), se puede representar como como es mostrado en la ecuación 4.1. Multiplicamos los *embeddings* en 2 dimensiones (1x2) por ésta matriz, de ésta manera encontraremos una representación de *embeddings* en alta dimensionalidad (1x256) que podremos utilizar para generar voz en la arquitectura T2MS.

$$X\cdot = W'^{-1} \cdot Y \tag{4.1}$$

PCA + K-Means. Esta alternativa se basa en utilizar K-Means, implementado en sklearn, para agrupar los *embeddings* según el género de voz al que pertenecen, apoyado de las etiquetas para darle sentido. Esto hace que la variación de K-Means utilizada pasa a ser una técnica de aprendizaje semi-supervisado.

Debido a lo presentado en la figura 4.4, podemos darnos cuenta que tenemos dos grupos de *embeddings*, cada uno siendo un género de voz distinto. Aprovechamos esta característica y encontramos 2 clusters donde los vectores de un género estarán más cerca de un solo centroide del cluster, y estarán todos asignados a ese cluster (idealmente). Así pues, procedemos a encontrar los clusters para posteriormente pasar a una alta dimensionalidad. El proceso completo lo podemos ver en la figura 4.7 donde, para generar el panorama completo, retomamos la reducción de dimensiones y posteriormente

utilizamos un clusterizador para obtener dos clusters (uno por género de voz).

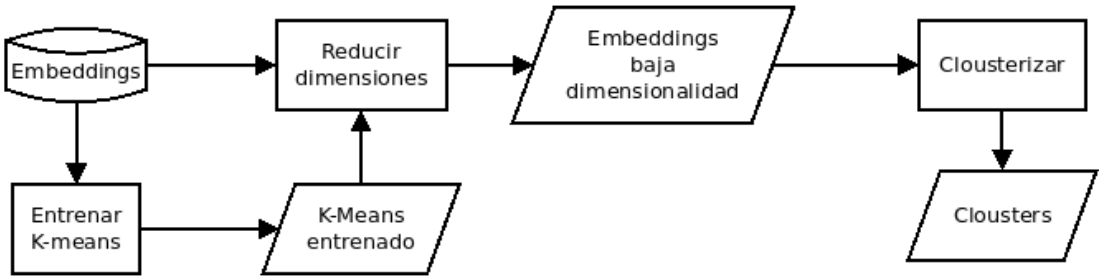

Figura 4.7: Entrenamiento de Kmeans

Una vez obtenidos los clusters, estos se pueden utilizar para muestrear un punto en 2 dimensiones y utilizando PCA inversa regresar a 256 dimensiones, como lo es mostrado en la figura 4.8.

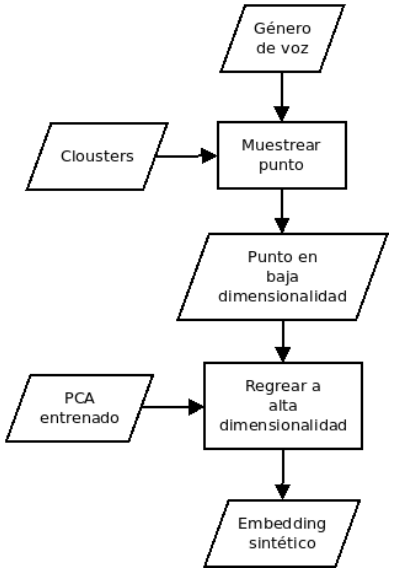

Figura 4.8: Aumento de dimensionalidad utilizando KMeans

‘

4.4.1.2. T-SNE

Al igual que para PCA, se utilizó un módulo T-SNE de sklearn para entrenar un módulo que pudiera reducir la dimensionalidad y de ahí generar algún tipo de clasificación para su posterior generación. Sin embargo, aún cuando existe la división entre *embeddings* de diferentes géneros (sin considerar el pequeño cúmulo de masculinos dentro del femenino que se ve en la figura 4.9), no existe manera natural para pasar de ésta

baja dimensionalidad a una alta como con PCA.

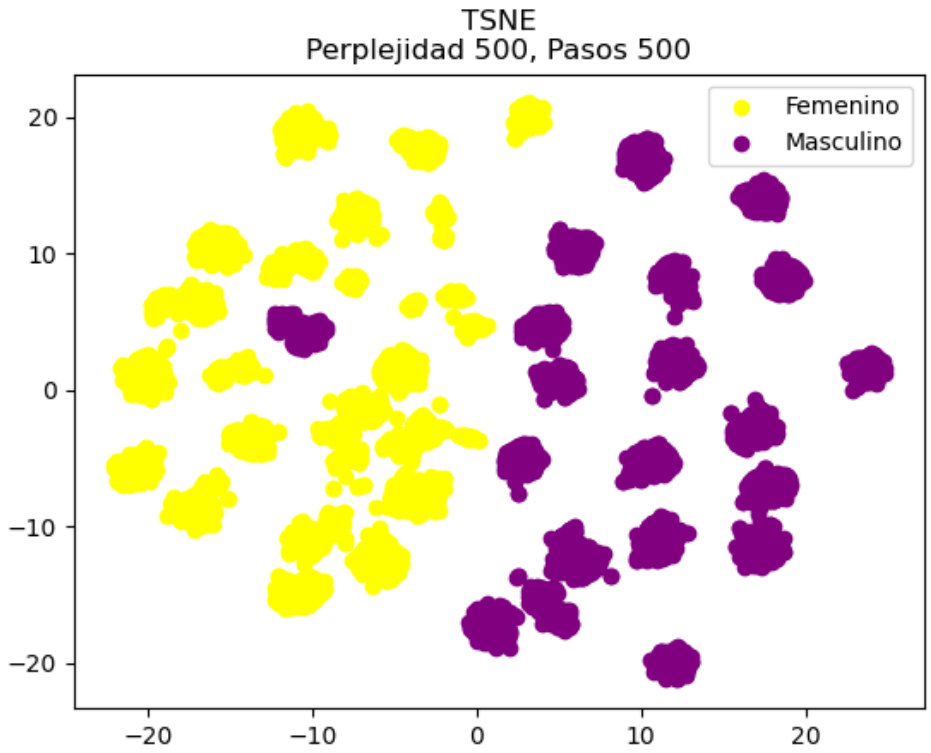

Figura 4.9: Resultados T-SNE

Además existe otro problema más grande que se encontró al hacer pruebas, y es que al nosotros necesitar generar *embeddings* sintéticos muestreando puntos del espacio de *embeddings* en 2D (no necesariamente uno de los puntos con el que se entrenó el modelo) el modelo TSNE se recalcula y la distribución de puntos que teníamos anteriormente ya no existe. Es decir, el espacio de *embeddings* de 2 dimensiones cambia. Por ende, no es viable utilizarlo para generar *embeddings* sintéticos hacia altas dimensionalidades, pero sirve para corroborar la cercanía de los *embedding* de T2MS en bajas dimensiones respecto al género de voz de cada uno.

4.4.2. Deep Learning

Debido a que la naturaleza de los *embeddings* aquí presentados no es artificial/sintética, no tenemos trasfondo de cómo están organizados los campos/columnas dentro de cada *embedding*. Así pues, no podemos aprovechar esta información para poder identificar un modelo y arquitectura exactos que nos ayude a nuestro objetivo. Para tratar

de identificar esto, calculamos la matriz de correlación del promedio de los *embeddings* de nuestra base de datos y podemos verla en la 4.10.

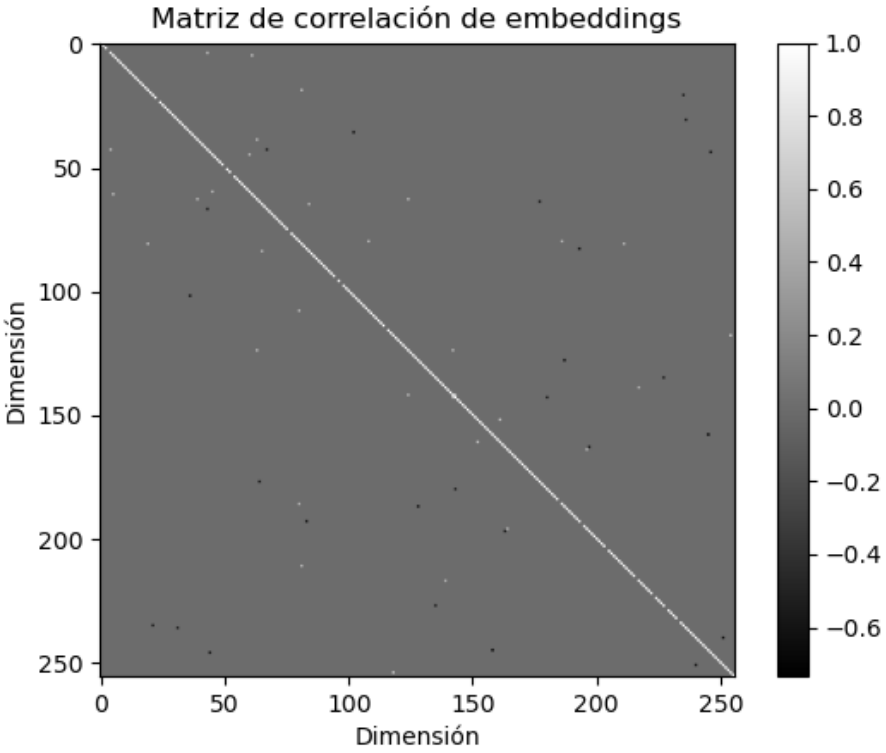

Figura 4.10: Matriz de correlación entre dimensiones

Sin embargo, la matriz no nos ayuda más de lo que ya sabemos. Las variables no están relacionadas directamente una con la otra de forma secuencial (siendo v el vector de *embeddings*: $v[0]$ con $v[1]$, $v[1]$ con $v[2]$, etc.), y las relaciones que presenta no nos dan la suficiente información para poder decidir sobre un modelo específico por lo que no podemos saber la estructura que tienen los datos. Por ello recurrimos a las redes neuronales artificiales, en específico a las generativas, entrenadas para aprender la estructura de los *embeddings* y así generarlos a nuestra conveniencia según el género deseado.

En la figura 4.11 vemos el proceso que se lleva a cabo para entrenar los modelos de deep learning que nos ayudarán a encontrar un modelo que genere *embeddings* sintéticos. Utilizamos βVAEs y cGANs ya que aprenden a generar los datos que se les entregan, además de que ambos nos permiten elegir el tipo de dato que se generará.

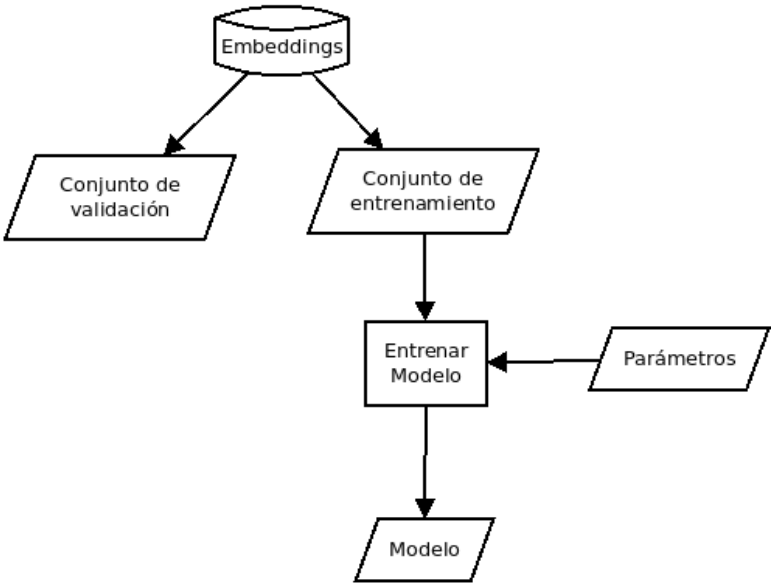

Figura 4.11: Entrenamiento general para modelos de deep learning

Si bien las βVAEs aprenden de una manera no supervisada, encontramos un vector de variables latentes donde una de las variables puede relacionarse con la generación de audio basándose en un género de voz. Las cGANs tienen su aprendizaje supervisado, por lo que la salida generada es directamente la que le pidamos según la entrada eligiendo un género de voz.

A continuación se enlistan las configuraciones propuestas para ambos modelos. Aquí cada uno de las configuraciones mostradas indican los módulos centrales de cada modelo. Así pues, la configuración gru_base para el caso de CGANs cuenta con módulos GRU en medio del generador y discriminador. Para el caso βVAE, los tiene en medio del codificador y decodificador. Las configuraciones se detallan en los apéndices B.2.1 y B.1 para βVAE y CGAN respectivamente.

- Red *fully-connected* (fc)
- Red convolucional de 1 dimensión (conv1d)
- Red convolucional de 2 dimensiones (conv2d)
- Red convolucional de 2 dimensiones utilizando la curva Z de Morton (46) para pasar entre diferentes dimensionalidades (utilizando una secuencia Moser-De Bruijin (47)) y así generar la imagen 2D a aprender (conv2d_morton)
- Red recurrente utilizando GRU (gru_base)
- Red recurrente utilizando GRU bidireccional (gru_bi)
- Red recurrente utilizando LSTM (lstm_base)
- Red recurrente utilizando LSTM bidireccional (lstm_bi)

4.4.2.1. βVAE

Los modelos βVAE utilizados en este trabajo, están basados en la implementación de (48). Para cada uno de ellos se utilizó la versión 'H', que es aquella que utiliza la función de pérdida original de (35). Si bien existe la versión 'B' que fue presentada en (13), para esta versión es necesario encontrar un parámetro que genere el mejor resultado, y eso es necesario hacerlo para cada modelo. Hacerlo para todos consumiría demasiado tiempo y eso queda fuera del alcance de éste proyecto.

Recordemos que el vector de variables latentes se obtiene después de pasar la entrada por el módulo codificador resultando en un vector de z dimensiones. Si se pasa posteriormente por el módulo decodificador, éste genera una salida muy similar a la entrada. Pues bien, el codificador y decodificador suelen ser simétricos, esto es, el decodificador realiza un proceso inverso a la entrada.

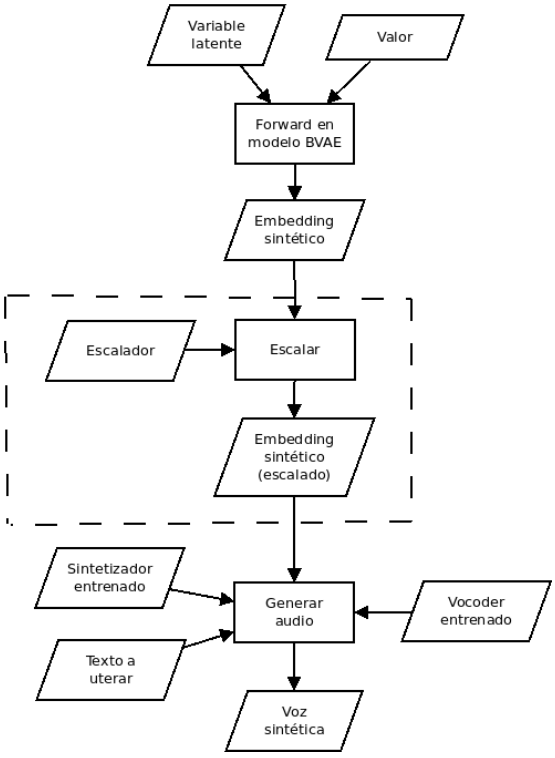

Figura 4.12: Generación de voz utilizando βVAE

Cada uno de los modelos, después de ser entrenado genera un modelo decodificador (que es el que nos interesa) cuyos pesos hacen que, dado un vector de z dimensiones de entrada, genere un *embedding* de voz. La importancia de esto es que ese vector de z dimensiones (**z-dim** a partir de ahora) tiene, idealmente, una dimensión/valor el cual puede parametrizar el género de la voz al ajustar el valor de la misma. Para los demás valores del vector es suficiente utilizar la mediana de los vectores.

El modelo a escoger, y que pueda parametrizar el género de la voz, es aquel que (dentro de uno de los modelos propuestos) tenga los parámetros adecuados para cumplir su objetivo. En éste caso, los parámetros principales, y a explorar, son el valor β, que se utiliza en la función de pérdida, así como la variable z-dim de variables latentes. Por ello, se realiza una búsqueda heurística de cuáles valores son los mejores para β y z-dim, para encontrar cuáles parámetros generan mejores resultados generales.

Los modelos generados tienen la siguiente nomenclatura:

<abreviatura de modelo>_ <learning rate>_ <beta>_ <dimensiones vector latente>

donde *<abreviatura de modelo>* es la abreviatura del modelo como fueron establecidos en la sección 4.4.2, y *<dimensiones vector latente>* es el número de dimensiones del vector latente en el modelo.

Debido a que es súmamente tardado e impráctico escuchar cada uno de los audios generados para poder determinar su género de voz (siguiendo el flujo de la figura 4.12), se utiliza un clasificador de audio entrenado con la base de datos original mencionada en la figura 4.1. Éste fue generado utilizando un SVM en sklearn con parámetro de $c = 5$ utilizando la base de datos de *embeddings* de alta dimensionalidad (256 dimensiones) con precisión de 99.94 % en conjunto de prueba.

Se utiliza un clasificador de *embeddings* y no de audios debido a que los *embeddings* ya están lo suficientemente separados en el espacio de 2 dimensiones, al grado que el clasificador en 2 dimensiones obtuvo más de 90 % de precisión. Además, para realizar las clasificaciones individuales usando un clasificador de audio se tendría que sintetizar un audio a partir del *embedding* y texto de referencia, y luego clasificarlo. El tiempo total de éste proceso es de al menos 5 segundos por audio, lo cual terminaría siendo impráctico si se considera que se deben de probar diversos parámetros (y modelos derivados de ellos).

4.4.2.2. CGAN

Los modelos CGAN utilizados en éste trabajo están basados en la implementación de (49). Al igual que en βVAE, también es necesario probar diferentes configuraciones de CGAN, aunque aquí la diferencia recae en que no tenemos un codificador y decodificador, sino un generador y un discriminador. Sin embargo, la estructura entre ambos suele ser similar (aunque en orden inverso en muchoss casos donde el discriminador es complejo), pero con la diferencia que la entrada del discriminador no sólo es un vector de variables de tamaño de la salida del generador, sino que es éste vector de variables con un vector one-hot concatenado.

Cada modelo, después de ser entrenado, nos generará un modelo generador que genera los *embeddings* de los audios. Este modelo recibe una entrada de un vector de variables (z-dim) concatenado a un vector one-hot que expresa el género de voz que se desea generar.

Éste modelo es más simple que uno βVAE ya que el vector z-dim no necesita ser

la mediana de un conjunto de datos, sino que basta con ser muestreado de una función de distribución normal y con media cero. Así pues, el proceso de generación de voz utilizando cGAN se muestra en la figura 4.13, donde internamente se muestrea un vector de dimensión de z-dim y se concatena al one-hot del género de voz (como se aprecia en la figura 3.9) para así crear un *embedding* que seguirá el proceso que han llevado los demás modelos presentados en éste trabajo.

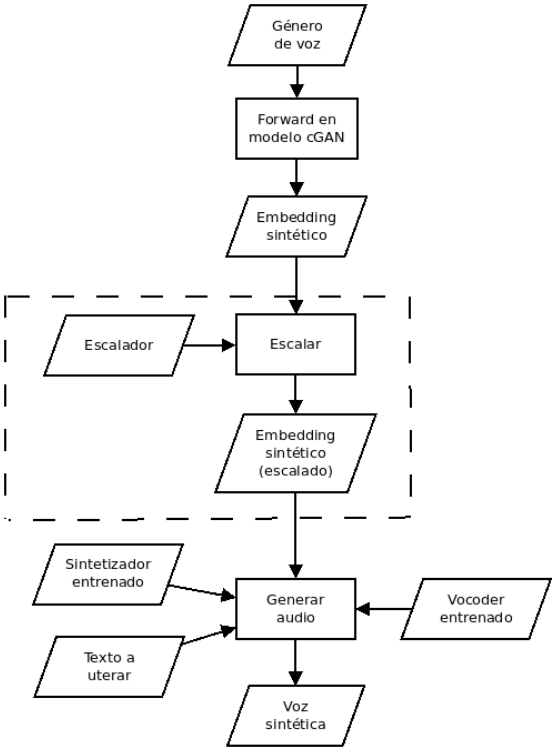

Figura 4.13: Generación de voz utilizando cGAN

Aquí los bloques encerrados en un cuadro con líneas no-continuas significa que puede o no realizarse ese proceso, pero se prueban ambos para encontrar en cuáles sí vale la pena dejarlo o quitarlo.

Los modelos generados tendrán la siguiente nomenclatura:

<abreviatura de modelo>_ <learning rate>_ <dimensiones vector latente>

donde *<abreviatura de modelo>* es la abreviatura del modelo como fueron establecidos en la sección 4.4.2, y *<dimensiones vector latente>* es el número de dimensiones del vector latente en el modelo.

Capítulo 5

Evaluación y resultados

En éste capítulo nos enfocamos en implementar las propuestas mencionadas en el capítulo anterior. Para esto necesitamos una forma de poder saber si nuestras propuestas generan resultados esperados, y para esto lo hacemos de dos maneras:

- Utilizando una interfaz gráfica que nos permita generar y escuchar la salida del modelo al seleccionar el género de voz deseado para ser generado, y cuando aplique, variar los parámetros del modelo.
- Utilizando criterios objetivos tales como valores de exactitud del modelo después de entrenarlo, así como métricas que se puedan adecudar a los modelos utilizados.

Además, presentamos los resultados de nuestras propuestas, tanto con criterios objetivos como subjetivos (cómo se aprecia la salida al escucharla) a la vez que mostramos y comparamos los parámetros que fueron probados dentro de los modelos, y su comparación con otros de la misma propuesta. Y por último discutimos las propuestas basándonos en los resultados obtenidos.

5.1. Interfaz gráfica para pruebas subjetivas

Para la prueba subjetiva de los resultados obtenidos se generó una interfaz gráfica que permite seleccionar una propuesta/prueba y con base en ella, poder generar audios. Las propuestas de machine learning no necesitan cargar modelos ya que sólo existe un modelo por propuesta (el que dio mejor resultado en entrenamiento), pero debido a que existen muchos modelos para las propuestas de deep learning, por la exploración de los valores de los parámetros, se da la opción de seleccionar qué modelo se ha de usar. En todas y cada una de las propuestas, lo que hará cada modelo al darle 'click' en *generar* será correr su función de generación de *embeddings* de alta dimensionalidad (256 dimensiones), que básicamente es el forward del modelo. Posteriormente se pasa ese *embedding* al *decoder* y *vocoder* de T2MS para generar el audio.

5.1.1. Machine Learning

5.1.1.1. PCA + SVM

En la figura 5.1 se puede ver la pantalla para el uso de interfaz gráfica configurada para la propuesta de utilizar PCA + SVM.

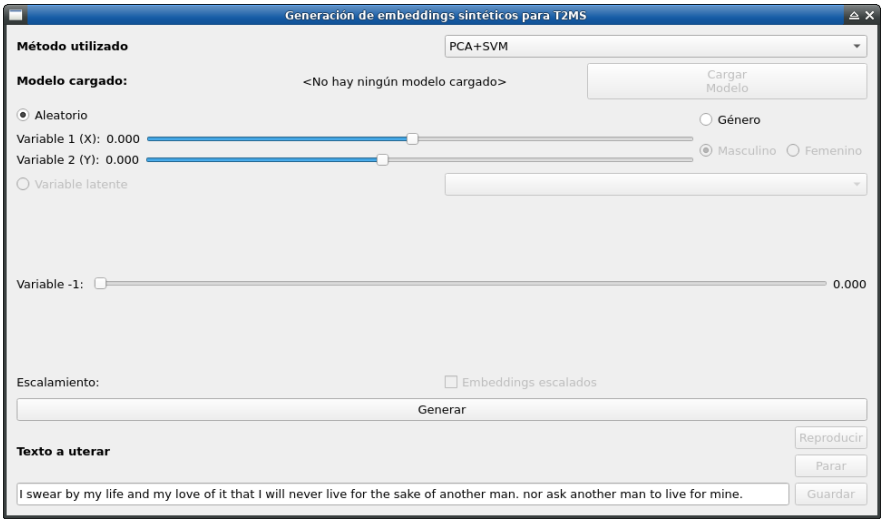

Figura 5.1: Interfaz de prueba de modelos para generación de audio en modo PCA+SVM

En ésta propuesta tenemos dos formas de generar audios sintéticos:

Por medio de parametrización directa. En esta opción podemos cambiar los valores de las variables de las dimensiones donde se alojan los *embeddings* de baja dimensionalidad (2D). Esto es, podemos cambiar el valor en el eje horizontal ('X') y vertical ('Y') que tendrá el punto que generaremos dentro de ese espacio de baja dimensionalidad. Y al presionar 'generar' tomará éste punto generado y realizará la operación de iPCA (PCA inversa, mencionada al final de 4.4.1.1) para generar un *embedding* de alta dimensionalidad.

Por medio del género deseado. Ya conociendo el hiperplano que genera la división entre *embeddings* de distintos géneros de voz, podemos parametrizar la generación de un punto dentro del espacio de baja dimensionalidad. Posteriormente se aplicará iPCA a ese punto generado, y obtendremos un *embedding* de alta dimensionalidad.

5.1.1.2. PCA + K-Means

En la figura 5.2 se puede ver la pantalla para el uso de la interfaz gráfica configurada para la propuesta de utilizar PCA + K-Means. En ésta propuesta sólo tenemos una

forma de generar audios sintéticos, que es el seleccionar directamente el género de voz que se quiere sintetizar. Al nosotros ya conocer el centroide del cluster al cual pertenece cada uno de los géneros de voz, podemos elegir uno de ellos, muestrear un punto que se encuentre dentro del circulo generado dado el radio que se encontró en el plano de dos dimensiones, y posteriormente aplicar iPCA para encontrar el *embedding* de alta dimensionalidad al que pertenece.

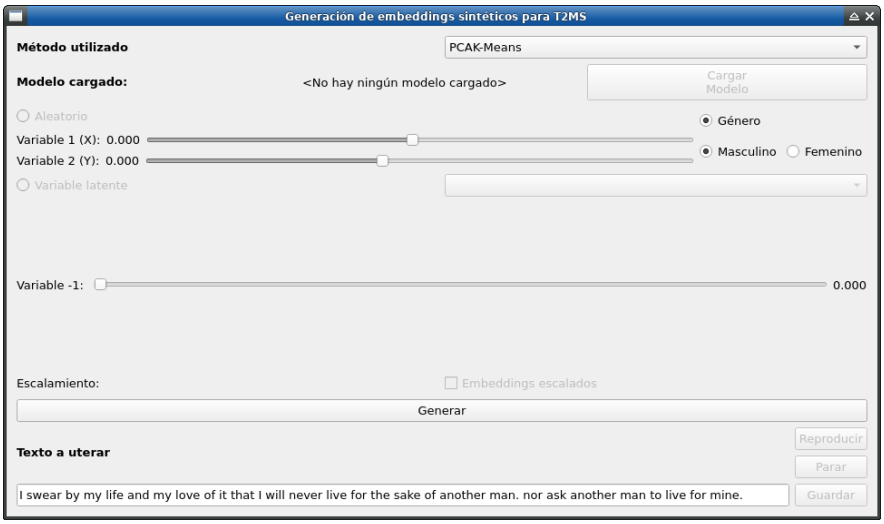

Figura 5.2: Interfaz de prueba de modelos para generación de audio en modo PCA+K-Means

5.1.2. Deep Learning

5.1.2.1. βVAE

En la figura 5.3 se puede ver la pantalla para el uso de interfaz gráfica configurada para la propuesta de utilizar βVAE. En ésta propuesta podemos cargar uno de los múltiples modelos que tenemos, posteriormente seleccionar la variable latente dentro del vector de variables latentes a la que se le modificará el valor, y por último variamos el valor de tal variable latente. Teniendo seleccionado el modelo, variable y valor de la variable, podemos ejecutar el método del modelo que nos permitirá tener un *embedding* de alta dimensionalidad.

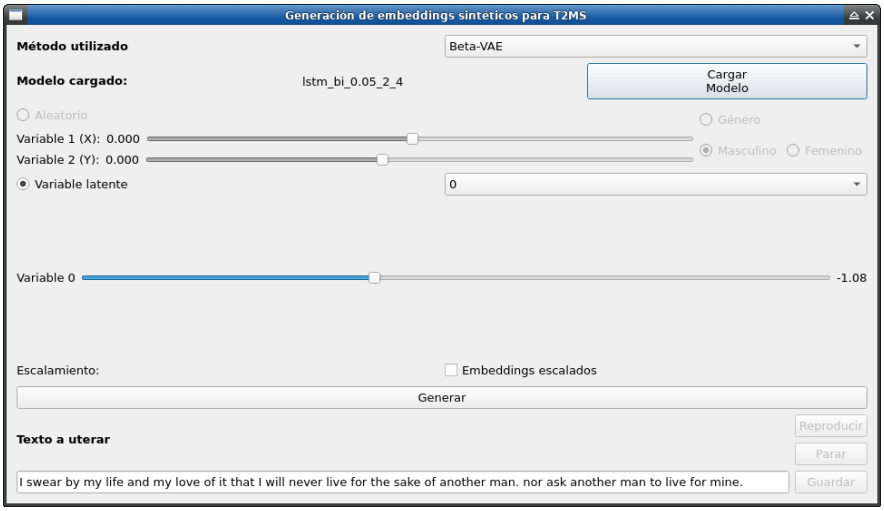

Figura 5.3: Interfaz de prueba de modelos para generación de audio en modo βVAE

5.1.2.2. CGAN

En la figura 5.4 se puede ver la pantalla para el uso de interfaz gráfica configurada para la propuesta de utilizar CGAN. En ésta propuesta podemos cargar uno de los múltiples modelos que tenemos y seleccionar el género de voz que queremos que tenga el audio final. Teniendo seleccionado el modelo y género de voz que queremos que sea generado, podemos ejecutar el método del modelo que nos permitirá tener un *embedding* de alta dimensionalidad.

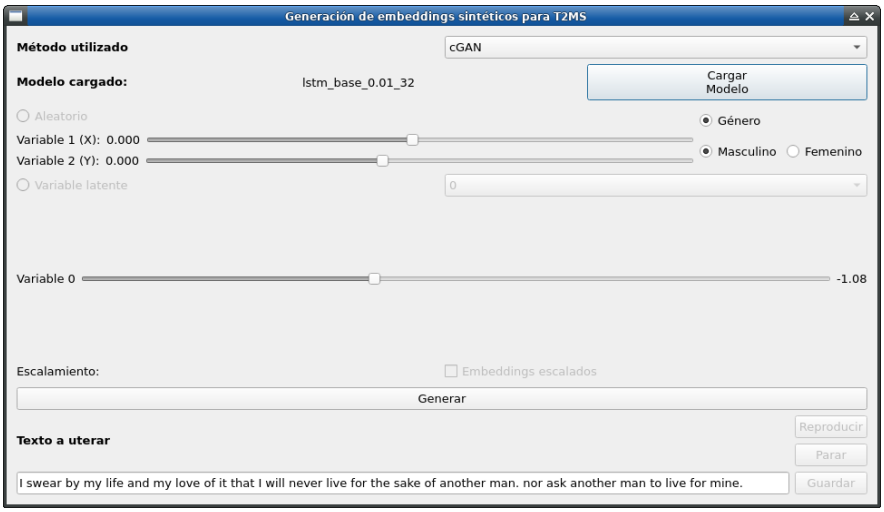

Figura 5.4: Interfaz de prueba de modelos para generación de audio en modo CGAN

5.2. Criterios objetivos

En ésta sección hablamos sobre las métricas probadas para elegir unos modelos sobre otros, para que a los mejores les sean reportados tanto sus parámetros como sus resultados. Aunque para la mayoría de los modelos aquí presentados se utilizó el valor de la exactitud al evaluar los modelos, para otros ésta no fue la única métrica probada, como en βVAE donde se probó la *disentangement metric score*.

5.2.1. Machine learning

Además de los criterios subjetivos mencionados en la sección anterior, se tienen ciertos criterios objetivos para la evaluación de las propuestas aquí descritas. Debido a que todas las propuestas son de naturaleza diferente, podrá o no aplicar el mismo criterio para más de un método.

Para sus dos variantes, SVM y K-Means, al momento de entrenar los algoritmos, podemos obtener una precisión en los conjuntos de entrenamiento y prueba, por lo cual podríamos saber qué tan efectivo será el utilizar esos algoritmos.

5.2.2. Deep Learning

5.2.2.1. βVAE

Las pérdidas de reconstrucción al entrenar los modelos βVAE sólo nos dan una ligera idea de qué tan bien se aprendió a reconstruir los *embeddings* de alta dimensionalidad, no tanto así el hecho de que la representación intermedia nos ayude a encontrar una variable latente que pueda generar un cambio de género de voz al ser alterada. Por ello, es necesario el utilizar una técnica fuera de esto para poder encontrar qué tantos cambios de género de voz se generan al variar una variable latente dentro de los modelos entrenados.

Para esto, necesitamos un clasificador de género de voz para *embeddings* de alta dimensionalidad, el cual se entrenó utilizando los *embeddings* originales de la base de datos elegida. Se utilizó un SVM lineal para este propósito con 30 % de datos de prueba y el resto de entrenamiento, y utilizando un valor de margen $c = 5$, obteniendo un 99.94 % de precisión en el conjunto de prueba (véase tablas 5.1 y 5.2).

Conjunto/ % prueba	**0.1**	**0.2**	**0.3**	**0.4**
Entrenamiento	0.9994	0.9993	0.9992	0.9993
Prueba	0.9986	0.9992	0.9994	0.9990

Tabla 5.1: Tabla comparativa respecto a valores de exactitud en entrenamiento y prueba para el modelo SVM en 256 dimensiones (c=5)

Conjunto/C	0.1	0.5	1	2	5	10	100	1000
Entrenamiento	0.9949	0.9973	0.9981	0.9987	0.9991	0.9992	1	1
Pueba	0.9957	0.9987	0.9989	0.9992	0.9994	0.9992	0.9992	0.9993

Tabla 5.2: Tabla comparativa de exactitud dados valores de c en SVM para 256 dimensiones (30 % de datos de prueba)

Una vez que tenemos una forma de decidir si un *embedding* pertenece a un género de voz específico, podemos revisar en masa los modelos que se entrenaron. Dado que necesitamos encontrar una variable latente, dentro del vector de variables latentes, que al variar su valor genere cambios de género de voz, es necesario probar cada una de las variables latentes y variar su valor para generar un *embedding* y poder clasificarlo. Sin embargo, puede darse el caso de que una variable latente tenga más de un cambio de género de voz, por ello no es adecuado sólo considerar aquellos modelos que tengan un cambio de género de voz. Más bien hay que mantener un registro de los géneros de voz mostrados al variar el valor de la variable latente y así detectar cuántos cambios de género existen. Se puede considerar este proceso como una forma de determinar cuántos puntos de inflexión ocurren donde se cambia de género de voz.

Para poder identificar estas inflexiones, sólo necesitamos variar el valor de una sola variable latente dentro del vector de variables latentes. Sin embargo, quedan otras variables que necesitamos asignarles un valor. Para ello necesitamos el valor medio o mediana de cada una de las variables (todo esto respecto a los valores que tendrían con valores reales). Para ello utilizamos el modelo a analizar y la base de datos de *embeddings* reales, y los pasamos por la parte codificadora del modelo para poder obtener el vector de variables latentes. Esto lo hacemos para cada uno de los *embeddings* obteniendo al final un conjunto de vectores de variables latentes a los que sólo necesitamos obtener las estadísticas de ellos. Así pues, obtenemos las medianas por cada variable latente, así como valores mínimo y máximos que nos sirven tanto para poder realizar la variación de valores en cada variable latente como para la interfaz gráfica. Éste proceso lo podemos ver en la figura 5.5. Se optó por usar las medianas en lugar de las medias dado que generaron mejores resultados empíricamente, lamentablemente no se tiene una métrica cuantitativa para mostrarlo.

Una vez teniendo estas medianas, procedemos a variar el valor de cada una de las variables latentes dentro del modelo que estamos analizando, y cada valor que le asignemos a tal variable genera un embedding, el cual clasificaremos y obtendremos una etiqueta que nos dirá a qué género de voz pertenece. Éste proceso se puede ver en la figura 5.6.

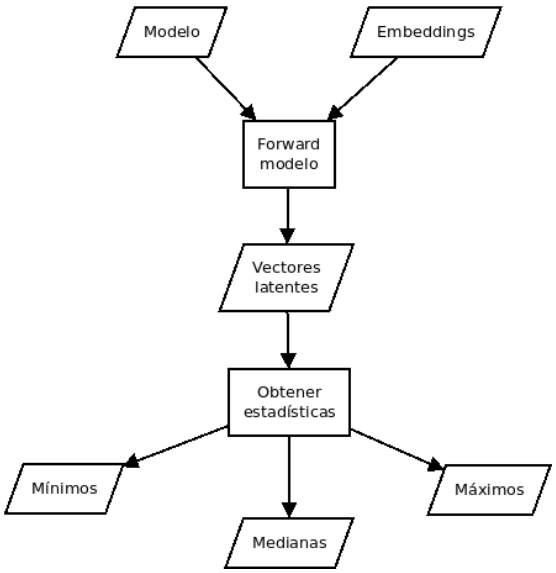

Figura 5.5: Diagrama para la obtención de medias de variables latentes

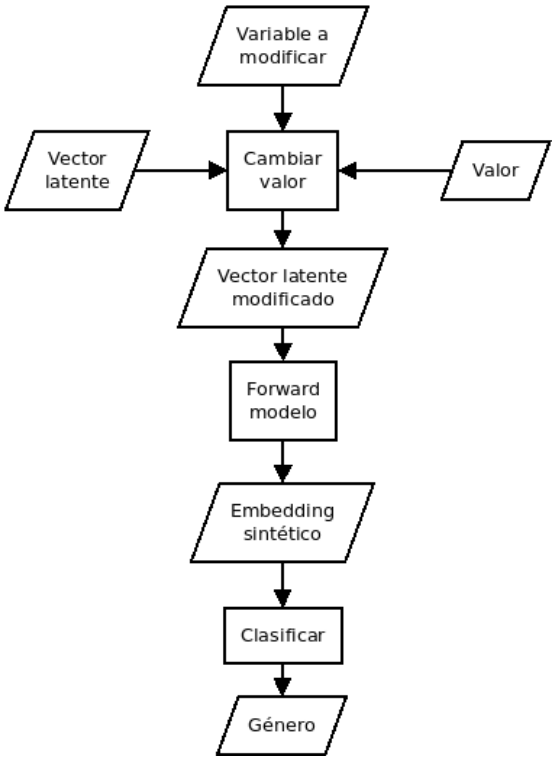

Figura 5.6: Diagrama de generación de *embedding* a partir de una variable latente modificada

Una vez listo el barrido de valores, obtenemos una lista de etiquetas a las que pertenece cada valor, y contamos cuántos puntos de inflexión genera cada modelo. Puede no existir inflexión alguna en el modelo, o pueden existir múltiples inflexiones (véase figura 5.7 para un ejemplo donde existen 9 inflexiones). Mayormente ésto nos indica que la variable analizada no genera cambio de género de voz al variar su valor. La cantidad de inflexiones buscadas es 1, dado que indica que la variable latente que estamos analizando aprendió la representación intrínseca del género de voz dentro. Un ejemplo de lo que se busca se encuentra en la figura 5.8, donde podemos apreciar que no sólo existe una única inflexión, sino que ocurre prácticamente a la mitad de la variación. Esto implica que hay una mayor probabilidad de que esa variable genere variaciones de voz considerables (y no sólo una voz), también garantiza con más fiabilidad el que genera un cambio de género de voz.

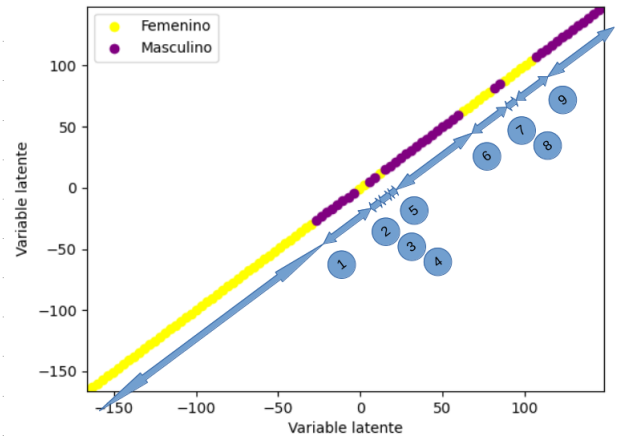

Figura 5.7: Muchas inflexiones en una sola variable

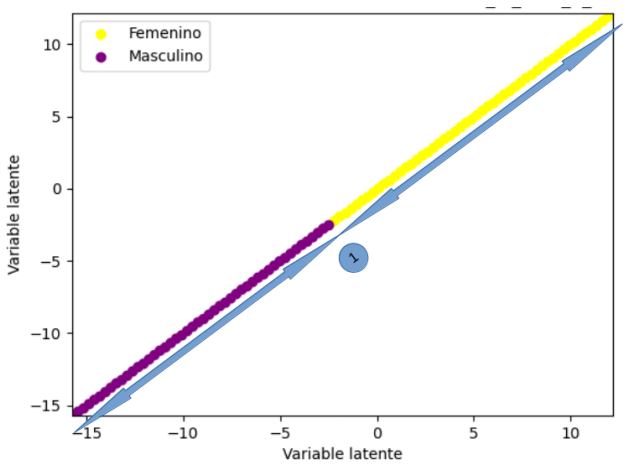

Figura 5.8: Una sola inflexión en una sola variable

Para las arquitecturas utilizando βVAE tampoco tenemos una métrica específica. Sin embargo los creadores de la arquitectura βVAE diseñaron una métrica (*disentanglement metric score*) la cual nos indica qué tan "desenredadas" están las variables latentes encontradas en el momento. Entre mayor sea el valor de la métrica mayor "desenredadas" estarán, pero esto genera datos que podrían cosiderarse "borrozos" (en imágenes, como si se aplicase un filtro de difuminación en partes o toda la imagen) ya que se pierde especificidad por ganar generalidad. Valores "bajos" a "promedio" están de 0 a 0.25 (según la figura 6 dentro de (35)).

5.2.2.2. CGAN

Para este modelo, se utilizaron la pérdida del modelo de generación y del modelo de discriminación como indicadores de cuán buen modelo es para poder tomarlo en cuenta y usarlo en la interfaz gráfica para comprobar su eficacia. Recordemos que el objetivo de las CGAN respecto a las pérdidas entre generación y discriminación es llegar a un balance idealmente con 0.5 en cada uno de ellos.

Para nuestras arquitecturas utilizando CGANs no tenemos una métrica cuantitativa que nos ayude a decidir el mejor modelo, ya que no existe una específica para ésta tarea que involucra síntesis de voz con aspectos subjetivos como el género de voz. Lo más cercano que existe es la métrica Fréchet Inception Distance (FID) (50) la cual ayuda a ver qué tan similares en términos estadísticos son el conjunto de datos original y el sintético generado con un modelo generativo (mayormente utilizado para GANs). Esto lo hace utilizando una red Inception, en específico la última capa antes de la clasificación (básicamente un embedding) donde obtiene estadísticas de los conjuntos de datos original y sintético y con ello obtiene un valor que consideran como una métrica. También se podrían hacer escalaciones y clonación de datos para poder utilizar nuestros *embeddings* como entrada, dado que nuestras dimensiones son [1,256], cuando se requiere dimensiones de [299,299,3] (50), pero se tendrían resultados inexactos y por mucho, ya que tenemos $1x256 = 256$ datos que deberíamos escalar a $299x299 = 89401$ (eso si hacemos que cada una de las 3 capas tengan los mismos datos), teniendo así que re-escalar nuestros datos 349 veces.

5.3. Resultados

5.3.1. Machine learning

5.3.1.1. PCA

Los modelos se entrenaron utilizando la base de datos mencionada anteriormente en la sección 4.2, entrenando el modelo de PCA con todos los datos ya que no es necesario hacer validación y prueba debido a la naturaleza del mismo. Los modelos de SVM y K-Means fueron entrenados con un 60 % de datos para el conjunto de entrenamiento, mientras que el resto fueron utilizados como conjunto de datos de prueba (comparativa en la tabla 5.3 y 5.4).

Conjunto/ % prueba	**0.1**	**0.2**	**0.3**	**0.4**
Entrenamiento	0.9652	0.9651	0.9646	0.9643
Prueba	0.9635	0.9649	0.9660	0.9661

Tabla 5.3: Tabla comparativa respecto a valores de exactitud en entrenamiento y prueba para el modelo SVM de PCA 2 dimensiones

Conjunto/ % prueba	**0.1**	**0.2**	**0.3**	**0.4**
Entrenamiento	0.9615	0.9615	0.039	0.9606
Prueba	0.9530	0.9585	0.0431	0.9608

Tabla 5.4: Tabla comparativa de valores de exactitud en entrenamiento y prueba para modelo K-means de PCA 2 dimensiones

Una vez entrenados los modelos basados en PCA, se obtuvieron los siguientes resultados:

SVM. Se comparó con diferentes valores de c (véase tabla 5.5) obteniendo el mejor desempeño con $c = 0.4$. El vector de soporte y márgenes resultantes son mostrados en la figura 5.9 donde los límites de los ejes son los valores que puede tomar el *embedding* en 2-D. Éstos separan a los *embeddings* de diferente género de voz en dos, identificándolos fácilmente. La recta que separa los géneros tiene una pendiente de *-7.5* y una elevación de *0.04*, con lo que podemos parametrizar los puntos que se muestrearán al generar un punto que sea de género masculino o femenino, y garantizaremos que sea del género deseado.

Conjunto/C	0.1	0.2	0.3	0.4	0.5	1
Entrenamiento	0.9647	0.9651	0.9649	0.9645	0.9643	0.9641
Pueba	0.9653	0.9662	0.9661	0.9663	0.9663	0.9663

Tabla 5.5: Tabla comparativa de exactitud dados valores de c en SVM para 2 dimensiones

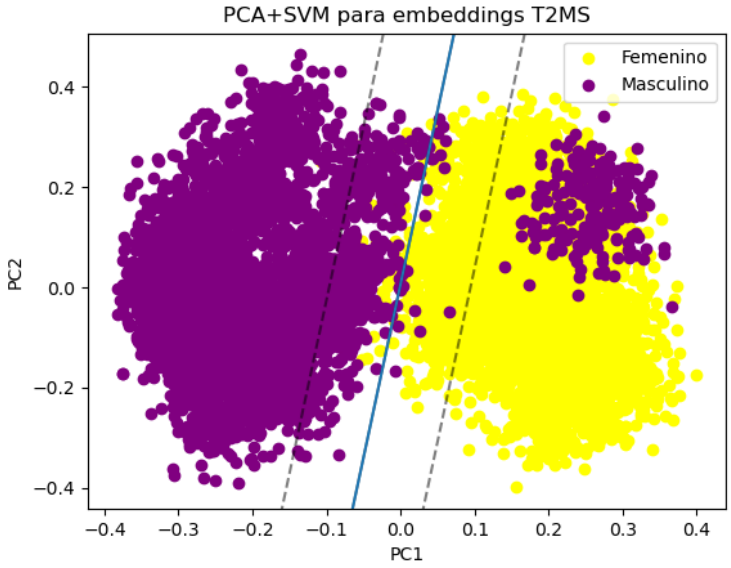

Figura 5.9: Resultados de utilizar SVM en *embeddings* de baja dimensionalidad

Los resultados subjetivos nos indican que sí se genera un cambio de género de voz al variar los valores de los ejes dónde se producirá el género de voz, aunque se nota menos la diferencia si se varía el eje 'X' (sin dejar de existir el cambio entre género de voz al pasar por el eje de soporte), por lo que la parametrización funciona adecuadamente. Asimismo, al generar audios eligiendo el género también produce los resultados deseados.

K-Means. Se obtuvieron dos clusters que pueden verse marcados en la figura 5.10. El punto marcado con una cruz roja (en cada cluster) es el centroide del mismo. Como podemos ver, prácticamente se encuentran a la mitad de cada uno de los clusters. Asimismo, los centroides se muestran en la ecuación 5.1 donde el primer valor se considerará x_0 y el segundo valor y_0.

$$\begin{aligned} \mu_1 &= [-0.2070911, -0.00583817] \\ \mu_2 &= [0.18378669, 0.00418289] \end{aligned} \tag{5.1}$$

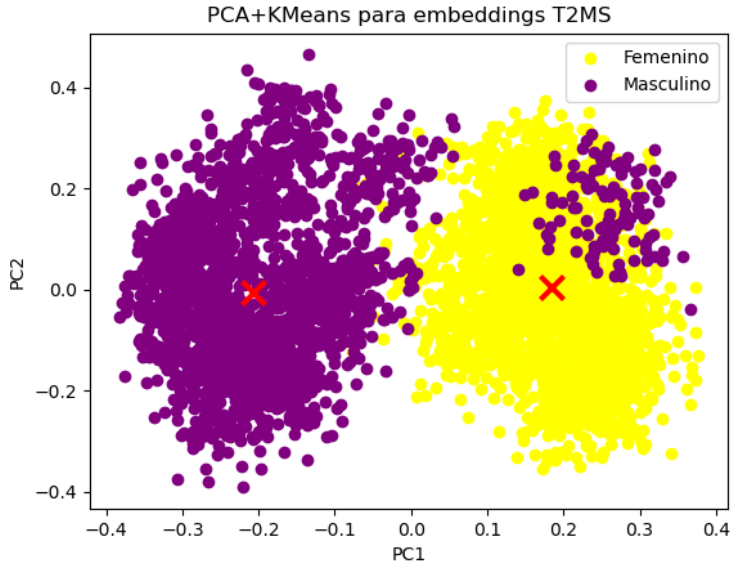

Figura 5.10: Resultados de utilizar K-Means en *embeddings* de baja dimensionalidad

Con esta información es posible generar muestras alrededor de cada uno de estos puntos para así poder obtener un *embedding* que se encuentre dentro de la vecindad de los *embeddings* de cada uno de los géneros de voz. Podemos seleccionar un género de voz, muestrear un punto en la vecindad de su centroide mientras asumimos una circunferencia de dónde se podrán obtener los datos con un cierto radio, y usar ese punto para generar el audio. Este radio es calculado utilizando la ecuación de la distancia de un plano a un punto la cual se muestra en la ecuación 5.2.

$$radio = \frac{|a \cdot x_0 + b \cdot y_0 - c|}{\sqrt{a^2 + b^2}} \tag{5.2}$$

Así, aplicando la ecuación 5.1 y usando como límite la recta obtenida con SVM en el punto anterior, obtenemos los radios de *0.2112* y *0.1776* para cada clase. Es importante mencionar que hacer esto implica dejar a un lado una cantidad de puntos de entrenamiento, pero, como se puede observar en la tabla 5.6) ambas técnicas muestran que son adecuados para la tarea.

Método/Conjunto	Entrenamiento	Prueba
SVM	0.9643	0.9661
K-Means	0.9606	0.9608

Tabla 5.6: Tabla de exactitud para SVM y K-Means para *embeddings* de baja dimensionalidad

5.3.2. Deep Learning

Los modelos aquí descritos se entrenaron utilizando la base de datos mencionada en la sección 4.2 utilizando un 80 % para el conjunto de datos de entrenamiento y el resto para datos de prueba. Se utilizó un optimizador Adam con una taza de aprendizaje diferente según el valor que se estuviese probando para el modelo evaluado. Para βVAE se entrenó con 200 épocas y para CGAN se entrenó con 100 épocas (para compensar la tardanza de entrenamiento al ser arquitecturas más grandes).

5.3.2.1. βVAE

Se probó entrenar modelos con tasas de aprendizaje variadas (como 0.01, 0.001, 0.0001, 0.05, 0.0005, etc...) pero se encontró que la que dio mejor resultados fue 0.05. También se variaron los valores de β desde 0 a 16 pero los mejores fueron [2, 4, 6 y 8], y para zDim (cantidad de variables latentes) se probaron con valores entre 2 y 128 (espaciados no uniformemente), pero los que daban mejores valores fueron [10, 16, 32 y 64] por ello son los que se reportan. Éstos valores se eligieron primeramente viendo los valores de las pérdidas de entrenamiento y se corroboraron sintetizando algunas muestras.

Se analizaron los modelos que se entrenaron utilizando la técnica de encontrar inflexiones descrita en la sección pasada. Los resultados arrojaron muchos modelos sin inflexión alguna y algunos otros con más de una inflexión, siendo una minoría los que generaban una sola inflexión.

Lo deseable es encontrar modelos con variables donde sólo existiese una sola inflexión, esto debido a que eso (idealmente) nos indicaría que sabiendo la variable, en el cuál existe tal única inflexión, y su valor podremos generar audios fácilmente de cada género. Así mismo, otro punto deseable es encontrar un punto de inflexión muy cercano a la mitad del rango de variación de la variable en cuestión, ya que ello (idealmente) permitiría gran variedad, y equidad en cantidad, de audios para cada género de voz.

Se probaron modelos con variables con una única inflexión usando la interfaz gráfica mencionada para generar audios, y se pudo corroborar cualitativamente los resultados que arrojan los valores de puntos de inflexión. Existen algunos modelos que tienen una única inflexión, como el analizado en la figura 5.11, donde se puede apreciar fácilmente el cambio de sonido además de tener audios inteligibles. Sin embargo, existen modelos

como el analizado en 5.12 que pese a tener una única inflexión, sus audios no generan cambio de voz aún cuando se sigue el rango seleccionado.

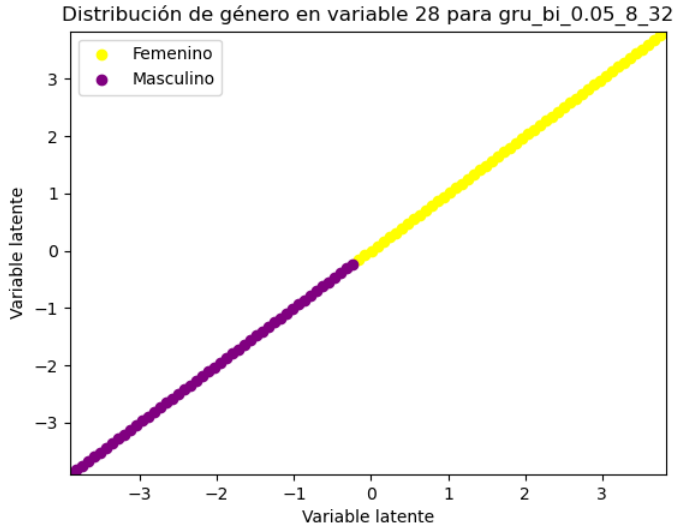

Figura 5.11: Resultados de modelo βVAE con lr = 0.05, beta = 8 y zdim = 32 (valores no escalados)

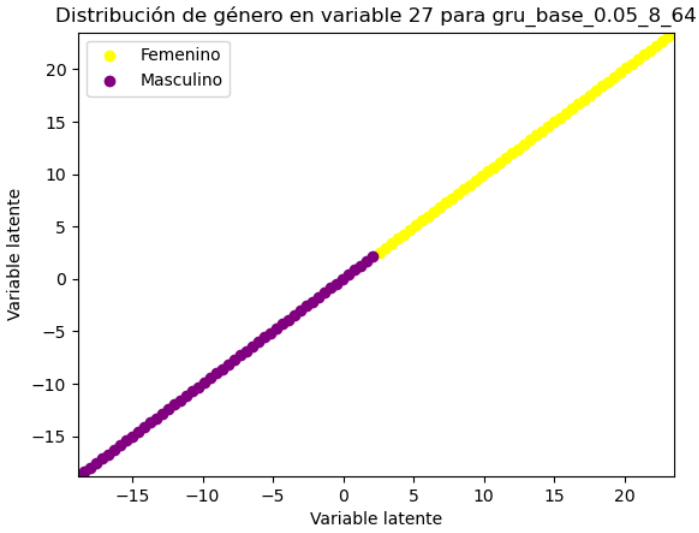

Figura 5.12: Resultados de modelo βVAE con lr = 0.05, beta = 8 y zdim = 64 (valores escalados)

Las tablas de inflexiones para los modelos finales se pueden consultar en el apéndice 1 (véase A).

Cabe recalcar que no todos los modelos de los que se encontró una inflexión resultaron generar adecuadamente voces, y algunos de los que sí las hacían no generaban variaciones de la voz, sino que mantenían una única voz, pero los aquí mostrados sí pudieron y con buenos resultados.

Algunos de los mejores modelos son los siguientes:

- gru_base_0.05_2_64, variable 1 y 16
- gru_bi_0.05_8_16, variable 7
- gru_bi_0.05_2_10, variable 0

Sólo son algunos de los que generaron mejores resultados pero ciertamente sólo fueron modelos recurrentes utilizando GRU. El mejor de estos tiene la siguiente configuración, obteniendo los audios más claros y variados:

- Tipo = GRU
- Learning rate = 0.005
- $\beta = 2$
- dimensiones latentes = 64

Respecto a la métrica cuantitativa, en la tabla 5.7 se comparan valores obtenidos con esta métrica para algunos de los modelos entrenados.

Modelo	Valor entrenamiento	Valor evaluación
gru_base_0.05_2_64	0.2095	0.013
gru_bi_0.05_8_16	0.1145	0.0665
gru_bi_0.05_2_10	0.147	0.101
gru_bi_0.05_8_32	0.105	0.0335

Tabla 5.7: Obteniendo puntaje de *disentanglement metric score* de modelos βVAE

Sin embargo, al escuchar las salidas de cada uno de los modelos, nos damos cuenta que, por ejemplo, "gru_bi_0.05_2_10" tiene un valor más alto en evaluación en ésta métrica, sin embargo no se escucha tan bien como uno de los modelos con mejores resultados subjetivos ("gru_base_0.05_2_64"), así mismo, el que tengan menos valor tampoco significa que sean mejores, ya que el modelo "gru_bi_0.05_8_16" llega a generar mayores variaciones de voz, mientras que también se puede cambiar el género,

y tiene mayor valor en ésta métrica. También otro ejemplo de dónde el valor bajo no significa inteligibilidad ni variación es el de "gru_bi_0.05_8_32" que tiene un valor menor que todos, excepto el primero, los de la tabla 5.7, pero sus audios no suelen generar una voz ni un cambio de género de voz.

Así pues, podemos ver cómo ésta métrica no sirve a nuestros propósitos aún cuando es la recomendada en (35). Sin embargo, se notó que sirve aún menos para los modelos donde se entrenó con los datos escalados, ya que algunos generan audios no inteligibles ("gru_bi_0.05_6_64") y obtienen valores bajos (0.0115), mientras que otros con valores altos (0.1025) son entendibles ("lstm_base_0.05_4_10"), pero no varían voz o género en ninguno de los casos.

Como dato curioso, algunos modelos generan voces con tonos de voz o características con los que no fueron entrenados, como "gru_bi_0.05_8_16" que genera voces que asemejan a personas de la tercera edad y "gru_bi_0.05_2_32" que llega a generar voces con tono de voz como los personajes "el padrino" y "Batman" en las variables 14 y 3 respectivamente.

Los resultados obtenidos, como se mencionó anteriormente con la 5.12 y los valores obtenidos con "disentanglement score metric", al utilizar escalamiento de la bases de datos en éstos modelos no son muy buenos dado que la mayoría de modelos no generaba un cambio de género de voz en sus variables latentes encontradas, aunque los modelos entrenados con éste escalamiento solían ser más inteligibles. Por ésta razón es que la mayoría de modelos reportados aquí no tienen habilitado el escalamiento.

5.3.2.2. CGAN

Se probó entrenar modelos con tasas de aprendizaje variadas (como 0.01, 0.001, 0.0001, 0.05, 0.0005, etc...) pero se encontró que lo que dieron mejores resultados fueron [0.01, 0.005, 0.0005]. También se variaron los valores de zDim (cantidad de variables latentes) usando valores entre 2 y 128 (espaciados no uniformemente), pero los que daban mejores valores fueron [8,32 y 128] por ello son los que se reportan. Éstos valores se eligieron primeramente viendo los valores de las pérdidas de generación y discriminación.

La primera aproximación lógica para elegir qué modelo tendría mejor aptitud para ésta tarea es el valor de la pérdida en el generador y discriminador. Sin embargo, se encontraron modelos con valores de generador y discriminador bajos, pero a la vez cercanos, que no generaron audios que asemejacen a voces. Tal es el caso, por ejemplo, del modelo "conv1d_0.0005_32" que tiene una pérdida de 0.6556 en el generador y de 0.7042 en el discriminador. Aunque hubo un modelo con valores cercanos de pérdida que sí generó voces de audios, y tiene una pérdida de 0.5921 en el generador y 0.6994 en el discriminador.

Sólo se encontró un modelo que genera voces humanas, aunque todas ellas del mismo género de voz y la variación de hablante es prácticamente nula. Sin embargo, es el único que no genera cosas fuera de voces humanas. Éste modelo tiene la siguiente configuración:

- Tipo = *fully-connected*

- Learning rate = 0.0005
- dimensiones latentes = 128

Los entrenamientos se realizaron utilizando valores de *embeddings* escalados y no escalados, sin embargo, éste modelo que sí llega a generar audios fue entrenado con *embeddings* escalados.

5.4. Discusión

Con respecto a los modelos de PCA con SVM y K-Means, los resultados fueron satisfactorios al poder generar un método para poder obtener *embeddings* sintéticos de 256 dimensiones al manipular una baja dimensionalidad. Si bien los porcentajes de entrenamiento y prueba son mejores para SVM respecto al aspecto cuantitativo, el aspecto cualitativo no se queda atrás y también genera voces donde se puede distinguir fácilmente que es una voz de género femenino o de género masculino. Asimismo, ambos generan variaciones de voces dentro del género que se está generando, sin embargo, en éste aspecto la versión de PCA con SVM se aprecia menos variación de voces, probablemente al ser una forma parametrizada y no muestreada dentro de la UI estemos utilizando menos precisión como la que se tendría utilizando un muestreador dentro de un área como lo hace la versión de K-means.

Respecto a los modelos de CGAN, de no ser por un único modelo podría haberse considerada fallida la exploración, pero se encontró y da resultados buenos de síntesis de voz. Sin embargo sólo se generaron voces de un solo género, por lo que no se obtuvieron resultados satisfactorios. El por qué se generaron éstos resultados puede deberse a que la red no fue lo suficientemente profunda en ninguna de las arquitecturas propuestas, los valores de las capas internas no fue el adecuado, la taza de aprendizaje debía ser otro, o simplemente la dimensión del vector de ruido debía haber sido otro. Son muchos factores los que influyen aquí pero para poder encontrar un modelo adecuado hubiese sido necesario realizar una infinidad de pruebas más y hubiese sido más conveniente tener una métrica cuantitativa que nos pudiese ayudar a encontrar el mejor modelo evaluando el audio generado sin necesidad de escucharlo. Sin embargo, todo esto está fuera del alcance de éste trabajo y se puede considerar para trabajo a futuro.

La métrica anterior nos da una pequeña idea pero en última instancia lo que ayuda a determinar si uno modelo es bueno o no, en éste proyecto, son únicamente los valores subjetivos: escuchar las salidas de los modelos. La gran mayoría de los modelos CGAN entrenados generan audios que se asemejan más a una psicofonía, animal salvaje o efecto de sonido de película de terror que a una voz humana. Sin embargo, se encontró una configuración, la mencionada anteriormente, que sí generó voces humanas aunque no logró generar voces del género que se le pida, sino que solamente genera del género masculino, aunque sí existe ligera variación de voz pero más en tono que en persona.

Respecto a los modelos de βVAE, los resultados fueron bastante satisfactorios pero no perfectos. La síntesis en la mayoría de los modelos era la de voces humanas y muy

rara vez era diferente a ésta como fue el caso con los modelos CGAN. Sin embargo (como con los modelos CGAN) hubiese sido conveniente tener una métrica cuantitativa específica para esta tarea ya que como vimos anteriormente, la métrica propuesta por (35) no aplica muy bien para éste trabajo. Asimismo, la variación de voz es diferente respecto al modelo (y variable) que se elija para variar los valores de la variable latente en cuestión, ya que en algunos casos la variación es considerable y es posible escuchar una gran variedad de voces. También en algunos casos la variación fue prácticamente nula subjetivamente. Tal vez utilizar modelos más profundos o ser entrenados los modelos con una gran cantidad de épocas como muchos modelos actuales (30) podría ayudar a generar resultados superiores. Sin embargo, para el alcance y objetivo de éste trabajo, se obtuvieron resultados esperados.

Por último, tenemos que el escalar los datos de entrada (embeddings) antes de entrenar, sólo benefició a la arquitectura CGAN, no así a la βVAE. Esto puede ser debido a que el discriminador de los modelos CGAN tiene como valores de salida entre 0 y 1 en el softmax, por lo que el tener los valores escalados entre 0 y 1 ayudaba a conseguír que los valores estuvieran cercanos en todo el proceso de entrenamiento. Las βVAE fueron probadas utilizando ReLU y LeakyRELU, sin embargo no dio buenos resultados ni con datos escalados ni con no escalados. Las sigmoides dieron mejores resultados, aunque no así en CGANs donde también se probaron.

Capítulo 6

Conclusiones y trabajo a futuro

6.1. Conclusiones

El trabajo realizado en (1) nos sirvió como base para poder encontrar un espacio de *embeddings* de 256 dimensiones que puede representar una voz mientras se distingue de otras. Asimismo, este espacio de alta dimensionalidad puede ser utilizado de distintas formas, y una de ellas es la propuesta en este trabajo: generar *embeddings* sintéticos condicionados al género de voz seleccionado.

En este trabajo nos propusimos aprovechar este espacio de alta dimensionalidad y pasar los *embeddings* a un espacio de menor dimensionalidad para poder manejarlos de forma más fácil en métodos de machine learning como SVM y K-Means. Esto lo realizamos utilizando PCA debido a la facilidad que tiene para encontrar componentes principales que tengan relevancia para los datos. En éste caso, el espacio de baja dimensionalidad que se generó fue adecuado ya que los *embeddings* están cerca a su propio género pero alejados del otro género (recordemos que tenemos más de 90 % de precisión con el conjunto de prueba en ambos métodos). Los resultados encontrados utilizando éstas técnicas nos muestran la viabilidad de poder generar los *embeddings* sintéticos sin dificultad y con buenos resultados cualitativos también, aunque en el caso de la versión parametrizada utilizando SVM no llegamos a tener tanta variedad como en la versión de K-Means.

En este espacio de alta dimensionalidad también encontramos representaciones de vectores latentes con βVAE, y modelos generativos con CGAN. En el caso de βVAE, los resultados fueron satisfactorios cualitativamente y pudimos generar audios que tanto se generan con el género de voz seleccionado como con una variedad de voces al variar la misma variable, aunque no se llegó a generar esto con todos los modelos entrenados. En el caso de CGAN, la mayoría de los modelos no generaban audios de voces humanas, aunque podrían considerarse humanas algunas pronunciaciones que generaban, pero parecían más bien quejidos y lamentaciones que voces. Cabe mencionar que se logró encontrar un modelo que sí las generaba. Sin embargo, las voces que generaba éste modelo no podía variarse su género de voz y la variación que existía entre ellas era más

de tono que de "hablante", aunque considerando los demás modelos de CGAN, ésta es una buena aproximación.

Se mencionó anteriormente (véase la sección 4.4.2) que al no tener correlación directa entre las variables de manera consecutiva no podíamos saber la estructura o poder asegurar qué tipo de red se tendría que utilizar. Considerando los modelos que sí generaron voces y variación, aunque no hayan sido los resultados más satisfactorios (considerando el caso de CGAN), vemos que los modelos que generaron buenos resultados en βVAE fueron de redes recurrentes, mientras que en CGAN fue un modelo de *fully-connected*. Podemos ver cómo se puede generar ésta síntesis con distintas arquitecturas, no sólo con recurrentes. Esto es contra-indicativo de lo que la intuición podría decirnos si nos dejamos llevar con la idea de que los *embeddings* son sacados de una señal de voz que varía a lo largo del tiempo.

Con esto concluímos que sí es posible generar *embeddings* sintéticos de voz basándonos en el espacio de *embeddings* de T2MS, tanto utilizando métodos de machine learning como de deep learning. Éstos últimos son más difíciles de entrenar y con resultados relativamente de menor calidad (salvo que se entrenen con más parámetros, capas y épocas). Sin embargo, como se llegó a observar, con deep learning se pueden obtener representaciones de variables latentes que pueden ser utilizadas para condicionar la salida, y que en este caso se utilizó para generar voces de distintos géneros. Aún así, se podría condicionar (con el análisis y pruebas suficientes) a otros aspectos de la voz, lo cual incluso se nota con los casos curiosos mencionados en la sección 5.3.2.1 donde se encontraron voces de personas de tercera edad. Esto apunta que es posible realizar condicionamientos de éste estilo, o bien encontrarlos en las variables latentes.

6.2. Trabajo a futuro

Continuando con la idea del presente proyecto, se podrían generar modelos de deep learning que generen una mejor representación latente de los *embeddings* así como una mejor variación de voces, lo cual tal vez sería posible aumentando capas, dimensiones y épocas en el entrenamiento.

Dejando de lado la condición por género de voz, se podría condicionar los *embeddings* con otras características como tono, timbre, etc. Será necesario ver si el espacio de *embeddings* actual es adecuado para éstas nuevas condiciones.

Aunque no existen métricas cuantitativas adecuadas para evaluar nuestros modelos, se puede agregar formalidad a los resultados cualitativos obtenidos por medio de la métrica Mean Opinion Score (MOS) (51). Dado que ésta requiere un conjunto de usuarios para calcularse (lo cual implica una gran inversión de tiempo), se propone sólo evaluar los modelos que dieron mejores resultados subjetivamente.

Por último, podrían hacerse pruebas con otras arquitecturas como Transformers que no tienen un uso directo para la generación sintética de datos pero adaptándolo para que realice ésta tarea. O al menos utilizar cabezas de atención en lugar de los módulos FC, Conv y RNN que se utilizaron aquí.

Apéndice A

Tablas de inflexiones

Las tablas de inflexión muestran cuántas variables posee cada uno de los modelos donde existe un único punto de inflexión, que es lo que buscamos.

Aquellos con valor 0 debe entenderse que o no tienen puntos de inflexión, o bien tienen más puntos de inflexión en cada variable latente.

A.1. βVAE sin escalar

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.1: Inflexiones para el modelo FC normal

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.2: Inflexiones para el modelo Conv1d normal

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.3: Inflexiones para el modelo Conv2d normal

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.4: Inflexiones para el modelo Conv2dMorton normal

β / z-dim	10	16	32	64
2	3	5	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.5: Inflexiones para el modelo Lstm normal

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.6: Inflexiones para el modelo BiLstm normal

β / z-dim	10	16	32	64
2	4	2	0	2
4	4	3	0	10
8	0	5	0	2

Tabla A.7: Inflexiones para el modelo Gru normal

β / z-dim	10	16	32	64
2	2	5	14	0
4	3	3	3	33
8	9	3	15	37

Tabla A.8: Inflexiones para el modelo BiGru normal

A.2. βVAE escalado

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.9: Inflexiones para el modelo FC escalado

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.10: Inflexiones para el modelo Conv1d escalado

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.11: Inflexiones para el modelo Conv2d escalado

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.12: Inflexiones para el modelo Conv2dMorton escalado

β / z-dim	10	16	32	64
2	3	3	18	4
4	3	0	2	0
8	0	0	7	0

Tabla A.13: Inflexiones para el modelo Lstm escalado

β / z-dim	10	16	32	64
2	0	0	0	0
4	0	0	0	0
8	0	0	0	0

Tabla A.14: Inflexiones para el modelo BiLstm escalado

β / z-dim	10	16	32	64
2	0	0	0	2
4	0	1	0	10
8	0	0	4	5

Tabla A.15: Inflexiones para el modelo Gru escalado

β / z-dim	10	16	32	64
2	0	0	0	10
4	0	1	3	11
8	9	0	2	2

Tabla A.16: Inflexiones para el modelo BiGru escalado

Apéndice B

Arquitecturas utilizadas

B.1. CGAN

Las arquitecturas aquí expuestas están compuestas de dos modelos: un generador y un discriminador. La base, o partes que se repiten en las demás arquitecturas, del generador es tener un vector de ruido que se concatenará (eso representa el símbolo a donde ambos apuntan) con un vector one-hot que representará a la clase que se quiere generar y generarán el vector latente, y la salida será un *embedding* sintético de 256 dimensiones. Lo que cambia en este módulo entre las variantes que se explicarán a continuación es el intermedio entre el vector latente y el *embedding* sintético. En éste módulo se utiliza la función de activación sigmoide ya que queremos generar *embeddings* y estos están en el rango de -1 a 1, aproximadamente, así que ésta función nos sirve por ello.

La base del discriminador es recibir un embedding, procesarlo y obtener un par de valores de probabilidad que nos indicarán, usando argmax, a qué clase cree el discriminador que pertenece el *embedding*. Cabe señalar que el módulo de argmax y la obtención de clases no se utilizan al entrenar. En éste módulo se utiliza la función de activación LeakyReLU ya que queremos que los valores se conserven arriba de 1, sin embargo, si usamos un ReLU normal los pesos podrían tender mucho a 0, lo cual nos genera peores resultados, mientras que LeakyReLU nos ayuda a evitar esto dejando un cierto margen de valores menores a 0 (aquí 0.2 en la mayoría de los casos salvo que se indique lo contrario) para evitar esto.
Se tomó como base (52) y (53) para considerar la disposición de dónde irían los módulos dropout, función de activación y más.

Nota: Para cada modelo tenemos que $ndim = longitud(\text{vector de ruido}) + longitud(\text{one-hot})$

B.1.1. Usando redes *fully-connected*

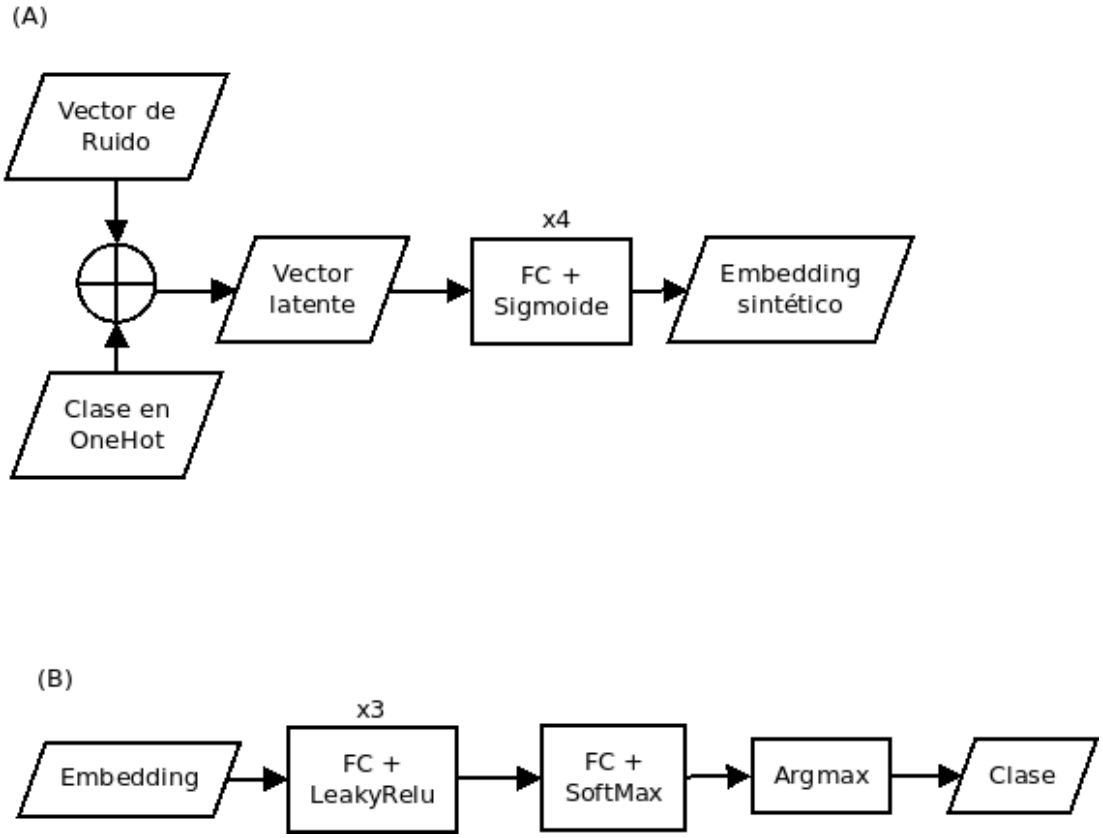

Figura B.1: Arquitectura CGAN para generación de *embeddings* con redes *fully-connected* como base. a) Modelo generador, b) Modelo discriminador

Para el generador: Entre el vector latente y el *embedding* sintético de salida tenemos 4 bloques de *fully-connected* [ndim a (1024-ndim)/2, (1024-ndim)/2 a 1024, 1024 a 512 y 512 a 256] + sigmoide. El vector de entrada va subiendo poco a poco la dimensionalidad, ya que la dimensionalidad del vector latente es menor a la del *embedding* sintético.

Para el discriminador: La entrada la proporcionamos a un conjunto de 3 bloques de *fully-connected* [256 a 512, 512 a 1024, 1024 a (1024-ndim)/2] + LeakyReLU, una de *fully-connected* [(1024-ndim)/2 a ndim] + softmax.

B.1.2. Usando redes convolucionales 1D

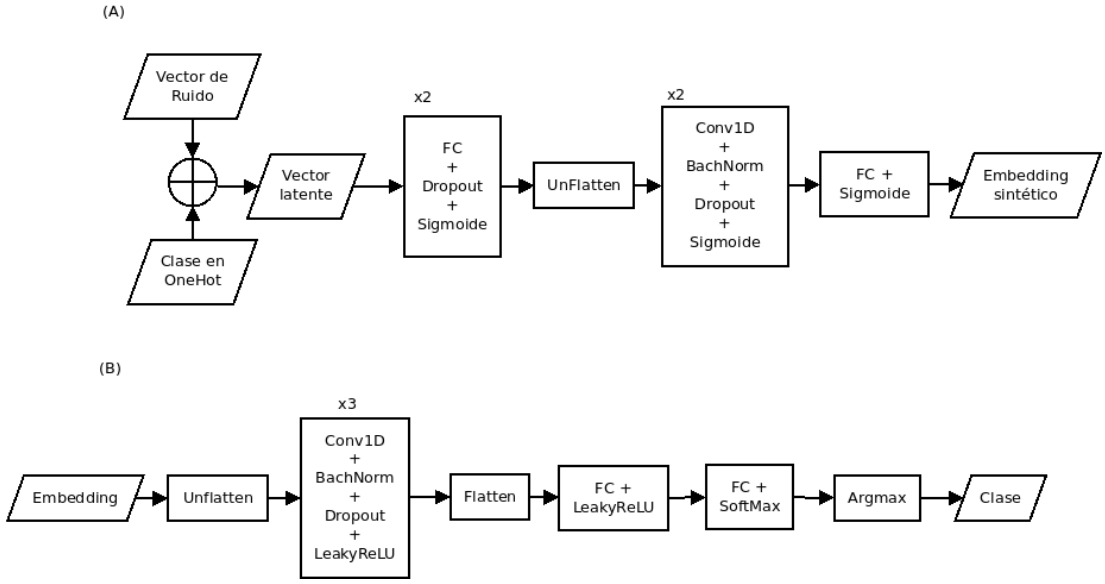

Figura B.2: Arquitectura CGAN para generación de *embeddings* con redes Conv1D como base. a) Modelo generador, b) Modelo discriminador

Para el generador: Entre el vector latente y el *embedding* sintético de salida tenemos 2 bloques de *fully-connected* [256 a 512, 512 a 1024] + dropout + sigmoide, y posteriormente movemos el tensor a uno de mayor dimensión para que tengamos la entrada que solicita Conv1D y aplicamos dos módulos de Conv1D [mismo número de canales de entrada y salida, kernel_size = 2, stride = 0 y padding = 0] + batch normalization + dropout de 0.5 + sigmoide, y por último aplicamos una fully connected [256 a 256] + sigmoide.

Para el discriminador: A la entrada le aumentamos una dimensión para que podamos utilizar Conv1D, y aplicamos 3 bloques Conv1D (kernel_size = 2, stride = 0 y padding = 0, pero el número de entradas y salidas va como sigue [1 a 4, 4 a 8 y 8 a 16]) + Batch normalization + dropout de 0.5 + LeakyReLU de 0.01. La salida de esto se pasa a una red *fully-connected* [512 a 250] con LeakyReLU y por último pasa por una red fully connected [250 a 2] + softmax.

B.1.3. Usando redes convolucionales 2D

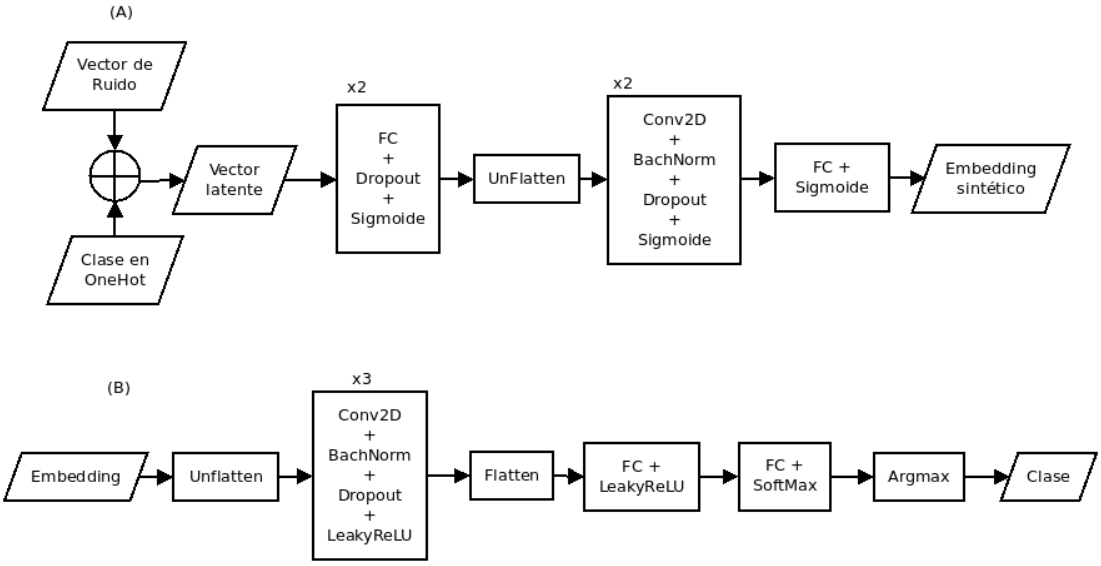

Figura B.3: Arquitectura CGAN para generación de *embeddings* con redes Conv2D como base. a) Modelo generador, b) Modelo discriminador

Para el generador: Entre el vector latente y el *embedding* sintético de salida tenemos 2 bloques de *fully-connected* [256 a 512, 512 a 1024] seguida de un dropout y sigmoide, y posteriormente movemos el tensor a uno de mayor dimensión para que tengamos la entrada que solicita Conv2D y aplicamos dos bloques de Conv2D [pero el número de entradas y salidas va como sigue [1 a 2, 2 a 4], kernel_size = 2, stride = 0 y padding = 0] + batch normalization + dropout de 0.5 + sigmoide, y por último aplicamos una fully connected [256 a 256] + sigmoide.

Para el discriminador: A la entrada le aumentamos una dimensión para que podamos utilizar Conv1D, y aplicamos 3 bloques Conv2D (kernel_size = 2, stride = 0 y padding = 0, pero el número de entradas y salidas va como sigue [1 a 4, 4 a 8 y 8 a 16]) + Batch normalization + dropout de 0.5 + LeakyReLU de 0.01. La salida de esto se pasa a una red fully connected [512 a 250] con LeakyReLU y por último pasa por una red fully connected [250 a 2] + softmax.

Para el modelo Conv2D normal sólo es necesario que la entrada sea una redimensión del *embedding* original a uno cuadrado (1x256 a 16x16), sin embargo, para la versión Conv2D utilizando la z de morton es necesario calcular el valor para cada uno de los (i, j) elementos de la matriz de 16x16 que se ingresará como entrada.

B.1.4. Usando redes recurrentes

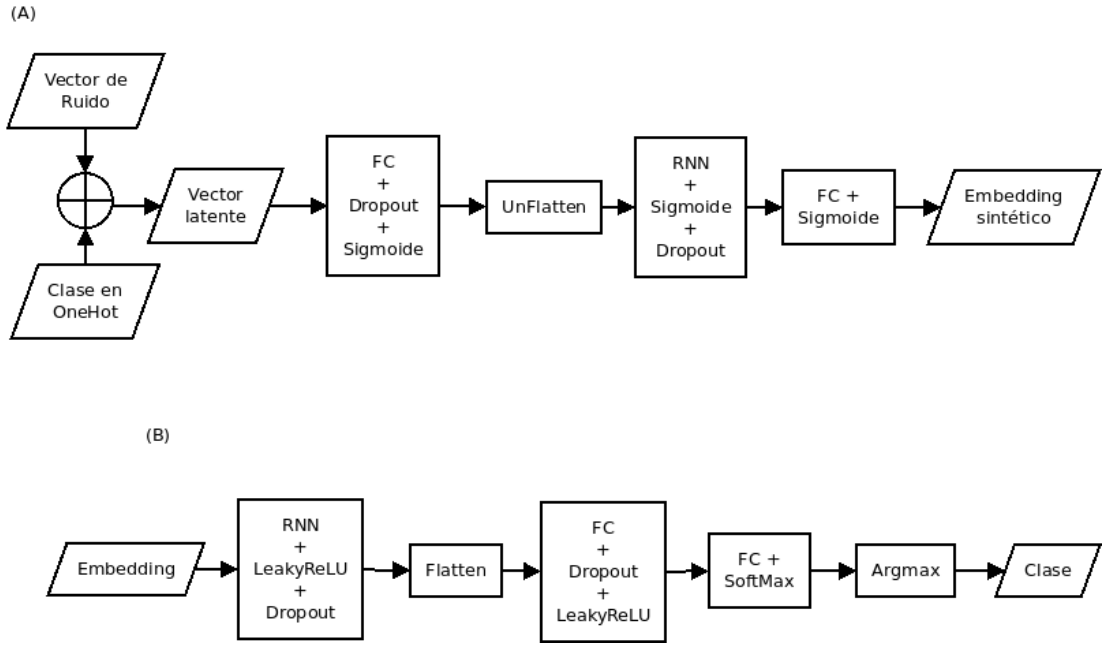

Figura B.4: Arquitectura CGAN para generación de *embeddings* con redes recurrentes como base. a) Modelo generador, b) Modelo discriminador

Para el generador: Entre el vector latente y el *embedding* sintético de salida tenemos 1 bloque de *fully-connected* [256 a 512, 512 a 1024] seguida de un dropout y sigmoide, y posteriormente movemos el tensor a uno de mayor dimensión para que tengamos la entrada que solicitan los módulos RNN y aplicamos un bloque de RNN [con una capa interna, bidireccional si se requiere y, 512 valores de entrada y 256 de estado oculto] + sigmoide + dropout de 0.5, y por último aplicamos una fully connected [256 a 256] + sigmoide.

Para el discriminador: La entrada la pasamos por un bloque de RNN [con una capa interna, bidireccional si se requiere y, 256 valores de entrada y 256 de estado oculto] + sigmoide + dropout de 0.5. La salida de esto se pasa aplana para reducir su dimensionalidad y posteriormente se pasa a un bloque fully connected [512 a 250] + Dropout + LeakyReLU y por último pasa por una red fully connected [250 a 2] + Softmax

Para las redes GRU y LSTM en su versión estándar o bidireccional, el módulo nombrado RNN es reemplazado por estos módulos (*lstm, lstm bidireccional, gru y gru bidireccional*); lo demás se mantiene intacto.

B.2. βVAE

Estos modelos son simétricos, por lo que sólo se explicará la parte del codificador (A) y el decodificador (B) será el mismo proceso pero a la inversa (cambiando tamaños, y capas, de salida por entrada y viceversa) así como que la otra diferencia es que no existe método de reparametrización dentro del decodificador, por lo que la entrada pasará directo al primer módulo listado. Así mismo, si en un módulo se utiliza Conv de codificador, en el decodificador se utilizan convTranspose.

Nota: Para cada modelo tenemos que $nDim$ es la dimensión del vector latente que se busca.

B.2.1. Usando redes *fully-connected*

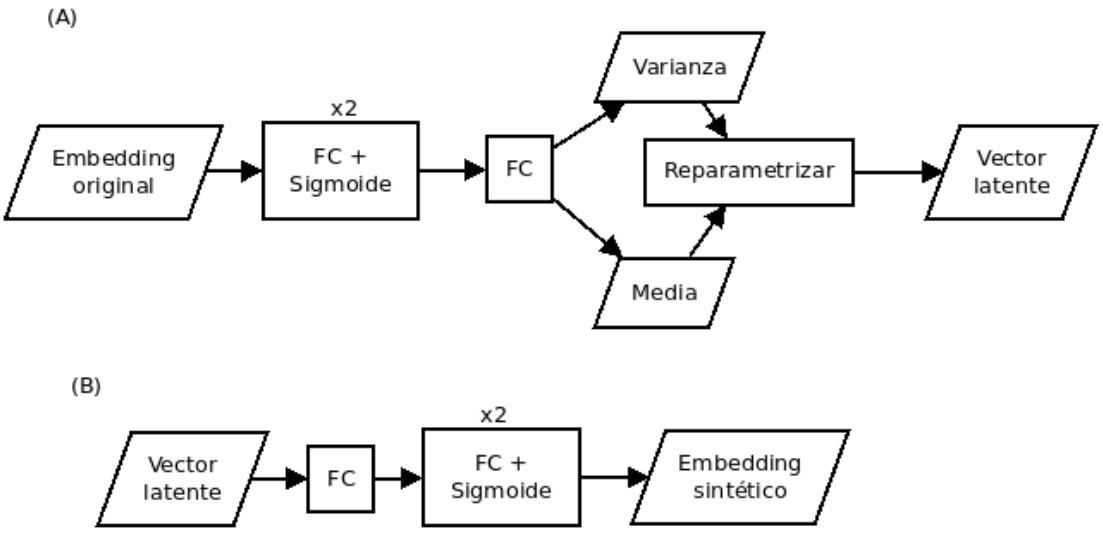

Figura B.5: Arquitectura βVAE para generación de *embeddings* con redes *fully-connected* como base. a) Codificador, b) Decodificador

Codificador: El *embedding* original se pasa a un par de bloques de fully connected [256 a 512, 512 a 1024] + Sigmoide, posteriormente se pasa su salida a una red fully connected [1024 a nDim*2] para obtener media y varianza (ambas de tamaño $nDim$), parametrizar ambas y obtener el vector latente.

B.2.2. Usando redes convolucionales 1D

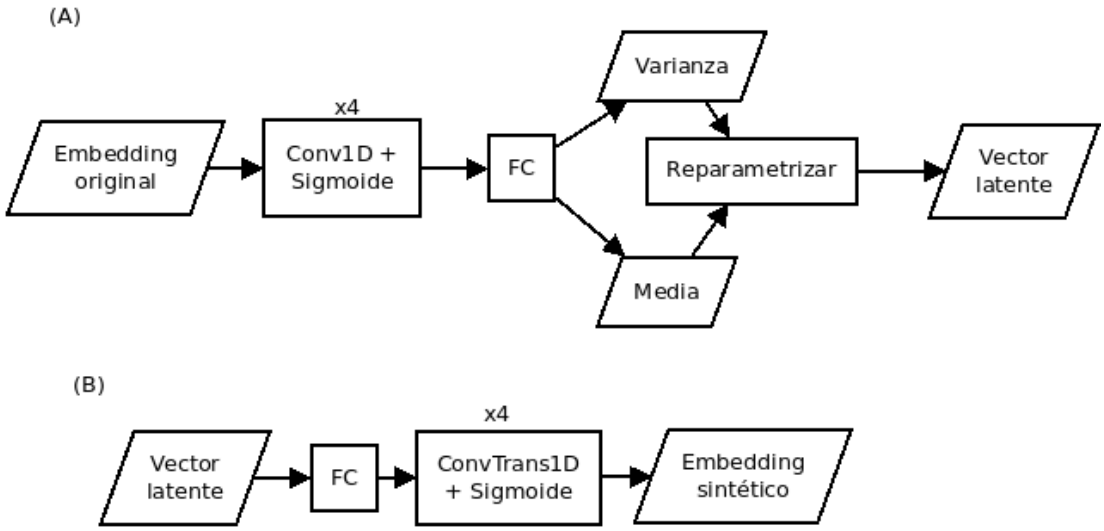

Figura B.6: Arquitectura βVAE para generación de *embeddings* con redes Conv1D como base. a) Codificador, b) Decodificador

Codificador: El *embedding* original se pasa a 4 bloques de Conv1D [kernel_size = 2, stride = 0 y padding = 0, con el mismo valor de canales de entrada y salida] + Sigmoide, posteriormente se aplana su salida y se manda a una red fully connected [16 a nDim*2] para obtener media y varianza (ambas de tamaño $nDim$), parametrizar ambas y obtener el vector latente. En las pruebas realizadas obtuvieron mejores resultados, al menos basándonos en pérdidas, al no utilizar dropout, por ello no aparece listado.

B.2.3. Usando redes convolucionales 2D

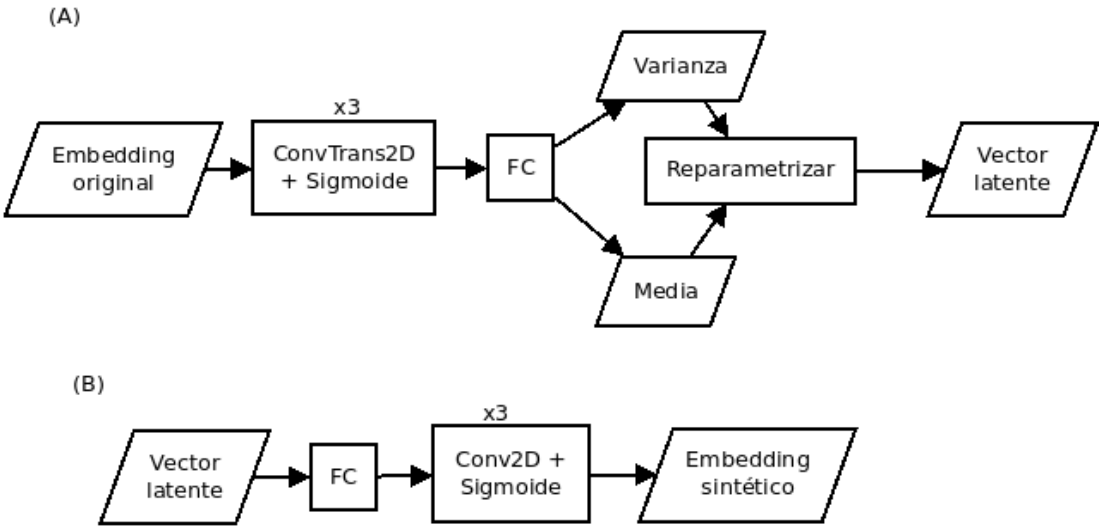

Figura B.7: Arquitectura βVAE para generación de *embeddings* con redes Conv2D como base. a) Codificador, b) Decodificador

Codificador: El *embedding* original se pasa a 3 bloques de ConvTranspose2D [kernel_size = 2, stride = 0 y padding = 0, con el mismo valor de canales de entrada y salida] + batch normalization + Sigmoide, posteriormente se aplana su salida y se manda a una red fully connected [128^2 (por la aplanda) a nDim*2] para obtener media y varianza (ambas de tamaño $nDim$), parametrizar ambas y obtener el vector latente.

Aquí para el modelo de Z de Morton, se obtienen los valores (i, j) pero se aplanan, ya que dentro del codificador se realiza el cambio de dimensiones a 16x16 justo antes de entrar a los bloques de Conv2D.

B.2.4. Usando redes recurrentes

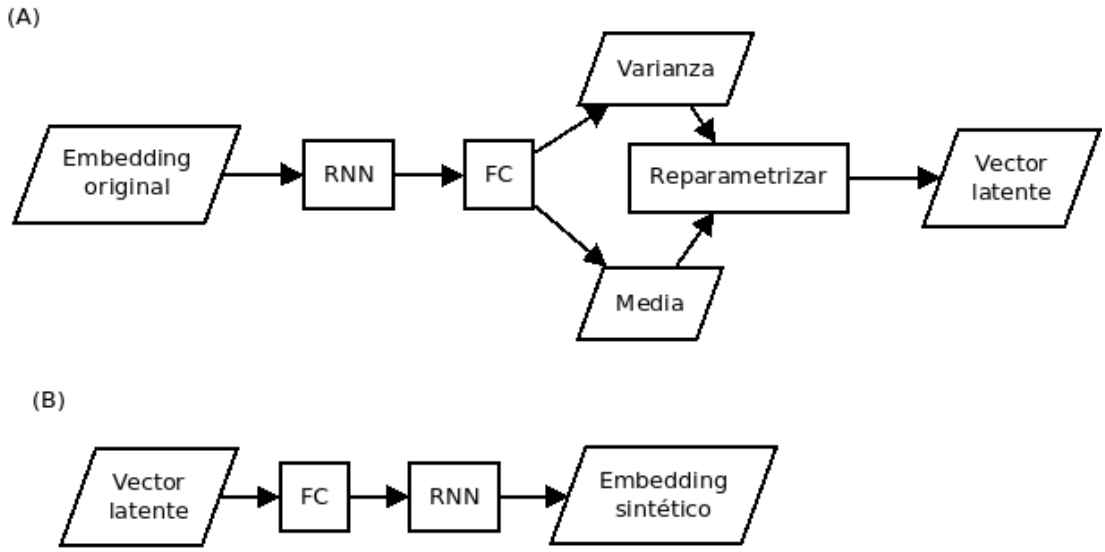

Figura B.8: Arquitectura βVAE para generación de *embeddings* con redes recurrentes como base. a) Codificador, b) Decodificador

Codificador: El *embedding* original se pasa a un bloque RNN, posteriormente se pasa su salida a una red fully connected [128 a nDim*2] para obtener media y varianza (ambas de tamaño $nDim$), parametrizar ambas y obtener el vector latente. Para las redes GRU y LSTM en su versión estándar o bidireccional, el módulo nombrado RNN es reemplazado por éstos módulos (*lstm, lstm bidireccional, gru y gru bidireccional*); lo demás se mantiene intacto.

Referencias

[1] Y. Jia, Y. Zhang, R. J. Weiss, Q. Wang, J. Shen, F. Ren, Z. Chen, P. Nguyen, R. Pang, I. L. Moreno, and Y. Wu, "Transfer learning from speaker verification to multispeaker text-to-speech synthesis," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), p. 4485–4495, Curran Associates Inc., 2018. xiii, 1, 2, 5, 7, 71

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Adaptive Computation and Machine Learning series, MIT Press, 2016. xiii, xiii, 25, 26, 27, 29, 30

[3] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *CoRR*, vol. abs/1411.1784, 2014. xiii, 30, 32

[4] D. Howard, "On voice synthesis," *IESIS Journal of Engineering, Paper No. 1658*, vol. 153, pp. 24–30, 01 2013. 1, 6

[5] W.-N. Hsu, Y. Zhang, R. Weiss, H. Zen, Y. Wu, Y. Cao, and Y. Wang, "Hierarchical generative modeling for controllable speech synthesis," in *International Conference on Learning Representations*, 2019. 2

[6] R. Habib, S. Mariooryad, M. Shannon, E. Battenberg, R. J. Skerry-Ryan, D. Stanton, D. Kao, and T. Bagby, "Semi-supervised generative modeling for controllable speech synthesis," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, OpenReview.net, 2020. 2

[7] S. Arik, J. Chen, K. Peng, W. Ping, and Y. Zhou, "Neural voice cloning with a few samples," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018. 5

[8] G. J. Ramsay, "Mechanical Speech Synthesis in Early Talking Automata," *Acoustics Today*, vol. 15, no. 2, p. 11, 2019. 5

[9] Y. Wang, R. J. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, and Q. Le, "Tacotron: Towards end-To-end speech

synthesis," *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, vol. 2017-Augus, pp. 4006–4010, 2017. 6

[10] S. Ö. Arık, M. Chrzanowski, A. Coates, G. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, A. Ng, J. Raiman, S. Sengupta, and M. Shoeybi, "Deep voice: Real-time neural text-to-speech," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 195–204, PMLR, 06–11 Aug 2017. 6

[11] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan, R. A. Saurous, Y. Agiomvrgiannakis, and Y. Wu, "Natural TTS Synthesis by Conditioning Wavenet on MEL Spectrogram Predictions," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2018-April, pp. 4779–4783, 2018. 6

[12] S. O. Arik, G. Diamos, A. Gibiansky, J. Miller, K. Peng, W. Ping, J. Raiman, and Y. Zhou, "Deep voice 2: Multi-speaker neural text-to-speech," *Advances in Neural Information Processing Systems*, vol. 2017-Decem, no. Nips, pp. 2963–2971, 2017. 6

[13] C. P. Burgess, I. Higgins, A. Pal, L. Matthey, N. Watters, G. Desjardins, and A. Lerchner, "Understanding disentangling in β-vae," *arXiv preprint arXiv:1804.03599*, 2018. 8, 47

[14] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," in *Arxiv*, 2016. 9

[15] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimber, A. Van Den Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," *35th International Conference on Machine Learning, ICML 2018*, vol. 6, pp. 3775–3784, 2018. 9

[16] R. Prenger, R. Valle, and B. Catanzaro, "Waveglow: A flow-based generative network for speech synthesis," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3617–3621, 2019. 9

[17] B. Z. J. Leitner and S. Thornton, "Audio recognition using mel spectrograms and convolution neural networks," 2019. 9

[18] K. Pearson, "On lines and planes of closest fit to systems of points in space," *Phil. Mag.*, vol. 6, no. 2, pp. 559–572, 1901. 11

[19] R. Billon, A. Nédélec, and J. Tisseau, "Gesture recognition in flow based on pca analysis using multiagent system.," in *Advances in Computer Entertainment Technology* (M. Inakage and A. D. Cheok, eds.), vol. 352 of *ACM International Conference Proceeding Series*, pp. 139–146, ACM, 2008. 12

[20] G. Golub, C. Van Loan, P. Van Loan, and C. Van Loan, *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 1996. 12

[21] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. 13

[22] J. Mccarthy, "What is artificial intelligence?." http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html, 2007. 14

[23] S. Suthaharan, *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*. Integrated Series in Information Systems, Springer US, 2015. 14, 16, 17

[24] M. Mohammed, M. Khan, and E. Bashier, *Machine Learning: Algorithms and Applications*. 07 2016. 16

[25] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, 2014. 16, 19, 20, 23, 24

[26] K. Murphy, *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning series, MIT Press, 2012. 16

[27] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001. 18

[28] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning," *arXiv preprint arXiv:2106.11342*, 2021. 21, 23

[29] N. Ketkar, *Deep Learning with Python: A Hands-on Introduction*. Apress, 2017. 24, 25, 26

[30] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. 24, 69

[31] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, 12 2010. 28

[32] D. Lee, H. Yu, X. Jiang, D. Rogith, M. Gudala, M. Tejani, Q. Zhang, and L. Xiong, "Generating sequential electronic health records using dual adversarial autoencoder," *Journal of the American Medical Informatics Association*, vol. 27, pp. 1411–1419, 09 2020. 28

[33] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006. 28

[34] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013. 28

[35] "beta-vae: Learning basic visual concepts with a constrained variational framework | openreview." 28, 40, 47, 59, 67, 69

[36] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 2672–2680, 2014. 30

[37] I. Didur, "Tacotron 2." https://github.com/ide8/tacotron2. 33

[38] J. Corentin, "Real Time Voice Clonning." https://github.com/CorentinJ/Real-Time-Voice-Cloning. 33, 38

[39] A. Raveendran, S. Jean, J. Mervin, D. Vivian, and D. Selvakumar, "Risc-v half precision floating point instruction set extensions and co-processor," in *VLSI Design and Test* (A. Sengupta, S. Dasgupta, V. Singh, R. Sharma, and S. Kumar Vishvakarma, eds.), (Singapore), pp. 482–495, Springer Singapore, 2019. 33

[40] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 5206–5210, IEEE, 2015. 35

[41] K. Ito and L. Johnson, "The lj speech dataset." https://keithito.com/LJ-Speech-Dataset/, 2017. 35

[42] A. Nagrani, J. S. Chung, and A. Zisserman, "Voxceleb: a large-scale speaker identification dataset," in *INTERSPEECH*, 2017. 35

[43] A. Hyvärinen and E. Oja, "Independent component analysis: Algorithms and applications," *Neural Netw.*, vol. 13, p. 411–430, may 2000. 40

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine

learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. 41

[45] I. Jolliffe, *Principal Component Analysis*. Springer Verlag, 1986. 42

[46] G. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966. 46

[47] N. G. de Bruijn, "Some direct decompositions of the set of integers," *Mathematics of Computation*, vol. 18, no. 88, pp. 537–546, 1964. 46

[48] AntixK, "Implementación bvae de antixk." https://github.com/AntixK/PyTorch-VAE/blob/master/models/beta_vae.py. 47

[49] YixinChen-AI, "Cvae-gan-zoos-pytorch-beginner/cgan.py at master · yixinchen-ai/cvae-gan-zoos-pytorch-beginner · github." https://github.com/YixinChen-AI/CVAE-GAN-zoos-PyTorch-Beginner/blob/master/CGAN/CGAN.py. 48

[50] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," *Advances in neural information processing systems*, vol. 30, 2017. 60

[51] I. Recommendation, "Vocabulary for performance and quality of service," *International Telecommunications Union—Radiocommunication (ITU-T), RITP: Geneva, Switzerland*, 2006. 72

[52] Z. Liu and X. Yin, "Lstm-cgan: Towards generating low-rate ddos adversarial samples for blockchain-based wireless network detection models," *IEEE Access*, vol. 9, pp. 22616–22625, 2021. 79

[53] triwave33, "Gan/cgan_mnist.py at master · gan · github." https://github.com/triwave33/GAN/blob/master/GAN/cgan/cgan_mnist.py. 79