

UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**APRENDIZAJE NO-SUPERVISADO CON MODELOS
GENERATIVOS PROFUNDOS**

Fernando Arribas Jara
Tutor: Daniel Hernández Lobato

Enero 2018

APRENDIZAJE NO-SUPERVISADO DE MODELOS GENERATIVOS PROFUNDOS

AUTOR: Fernando Arribas Jara
TUTOR: Daniel Hernández Lobato

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero de 2018

Resumen

En una sociedad en constante cambio como en la que vivimos la tecnología se ha convertido en una pieza fundamental en nuestras vidas, cambiando nuestro estilo de vida de manera permanente. Una de las áreas científicas artífice de esta incesante metamorfosis a la que está sometida la sociedad actual es sin duda la Inteligencia Artificial (IA).

Los avances en Inteligencia Artificial son posibles gracias, entre otras cosas, a la gran cantidad de datos que manejamos y a la potencia computacional de la que disponemos en la actualidad. Nos hallamos en la era de los datos (Big Data) y conseguir explotarlos es el objetivo de técnicas como el Aprendizaje Automático que se ocupa de obtener patrones estructurales y predecir hechos en base a los datos observados.

Dentro del Aprendizaje Automático, la rama con mayor auge es el Aprendizaje Profundo, que habitualmente se conoce como redes neuronales profundas y tiene como objetivo principal orientar el proceso aprendizaje de forma muy similar a como aprendemos los seres humanos, simulando el funcionamiento de nuestro cerebro.

Los modelos generativos profundos suponen una amplia área de investigación dentro del aprendizaje profundo, concretamente una nueva técnica, los **Autoencoders Variacionales (VAE)**.

Los Autoencoders Variacionales son una técnica de aprendizaje profundo no-supervisado perteneciente al grupo de los Autoencoders, que al igual que estos tienen como objetivo reconstruir los datos de entrada. A diferencia de los Autoencoders típicos, estos aprenden representaciones latentes de los datos observados, con la finalidad de entender mejor los datos observados y usar estas representaciones para tareas generativas.

Palabras clave

Aprendizaje Automático, Redes Neuronales, Inteligencia Artificial, Perceptrón Multicapa, Decodificador, Codificador, Variables Latentes, Inferencia

Abstract (English)

In a society in constant change like the one in which we live, technology has become a fundamental piece in our lives, changing our way of life permanently. One of the creative scientific areas of this incessant metamorphosis to which today's society is subjected is undoubtedly Artificial Intelligence (AI).

The advances in Artificial Intelligence are possible thanks, among other things, to the large amount of data that we manage and the computational power that we have at present. We are in the time of data (Big Data) and get to exploit them is the goal of techniques such as Automatic Learning that deals with obtaining structural patterns and predict facts based on the observed data.

Within the Automatic Learning, the branch with greater height is the Deep Learning, that usually is known like deep neuronal networks and has like main objective to orient the learning process of form very similar to how we learn the human beings, simulating the operation of our brain.

The deep generative models suppose a wide area of investigation within the deep learning, concretely a new technique, the Variational Autoencoders (VAE).

The Variational Autoencoders are a deep non-supervised learning technique belonging to the group of Autoencoders, which, like these, aim to reconstruct the input data. Unlike the typical Autoencoders, they learn latent representations of the observed data, in order to better understand the observed data and use these representations for generative tasks.

Keywords

Machine Learning, Neural networks, Variational Autoencoder, Artificial Intelligence, Multilayer Perceptron, Decoder, Encoder, Latent Variables, Inference, Black-box Inference...

Agradecimientos

A todos los profesores que he tenido a lo largo de todo este tiempo, por todos los valores y nuevos conocimientos que me han aportado. En especial, a mi tutor, Daniel Hernández Lobato, por darme la oportunidad de sumergirme en este mundo tan apasionante de las redes neuronales y permitirme realizar las tutorías vía Skype debido a mi lesión grave de Tendón de Aquiles. Dándome apoyo y respuestas a mis problemas de la mejor manera posible, dadas las circunstancias.

A la Escuela Politécnica Superior por darme la oportunidad de formarme académicamente y desarrollarme como persona. Y por conocer personas increíbles que formarán parte de mi vida siempre.

A mis amigos por todo el apoyo que me han dado, aportándome la fuerza necesaria para afrontar las adversidades

A mis padres por darme la oportunidad de poder formarme durante 25 años e inculcarme unos valores y principios que hacen la persona que soy ahora. Sin ellos hubieran sido imposible conseguir llegar hasta aquí.

A mis hermanos y novia por ser junto a mis padres, la parte fundamental de mi vida. Por su continua insistencia y apoyo durante este tiempo. Por eso y por mucho más. GRACIAS

INDICE DE CONTENIDOS

1	Introducción.....	7
1.1	Motivación.....	10
1.2	Objetivos,.....	11
1.3	Organización de la memoria.....	12
2	Estado del arte	13
2.1	Redes neuronales	13
2.1.1	Cronología de las redes neuronales[12]	13
2.1.2	Analogía entre Neurona Artificial y Neurona Biológica.....	15
2.1.3	Funcionamiento de una Neurona Artificial	16
2.1.4	Estructura de las Redes neuronales	18
2.1.5	Propagación de la Información y proceso de aprendizaje en redes neuronales.....	19
2.1.6	Modelos generativos: Autoencoders Variacionales	21
3	Autoencoders Variacionales	23
3.1	Funcionamiento de un Autoencoder Variacional	23
3.2	Estructura de la Red Neuronal.....	24
3.3	Fundamentos matemáticos	25
3.3.1	Inferencia Variacional	28
3.3.2	Optimización y Función de pérdida.....	30
3.3.3	Truco de la Re-parametrización	31
4	Decisiones de Diseño de la red neuronal.....	33
4.1	Arquitectura de la red	33
4.1.1	Encoder.....	34
4.1.2	Decoder.....	35
4.1.3	Espacio Latente.....	36
4.2	Inicialización de pesos y bias	36
4.3	Funciones de activación.....	37
4.4	Función de Pérdida	37
4.5	Optimizador	38
5	Implementación	40
5.1	Código	40
5.1.1	Importación de módulos y dataset	40
5.2	Parámetros de configuración del modelo	41
5.3	Promesas del modelo: Placeholder	41
5.4	Encoder.....	42
5.5	Espacio Latente.....	43
5.6	Decoder.....	43
5.7	Función de pérdida	44
5.8	Optimizador.....	44
5.9	Entrenamiento.....	44
6	Experimentos y resultados.....	46
6.1	Entrenamiento del modelo.....	47
6.1.1	Experimentos con MNIST.....	47
6.1.2	Experimentos con Frey Face	51
6.2	Clasificador Lineal: Regresión logística multinomial	53
6.2.1	Clasificador Semi-supervisado con Autoencoder Variacional.....	53
6.2.2	Clasificador supervisado con imágenes originales.....	54

6.2.3 Resultados.....	54
7 Visualizaciones.....	55
7.1 Jugando con la dimensionalidad.....	55
7.2 Dispersión del espacio latente	56
7.2.1 Espacio Latente 2-D mostrado en dos dimensiones MNIST.....	56
7.2.2 Espacio Latente 2-D mostrado en tres dimensiones MNIST	57
7.2.3 Dispersión del espacio latente Frey Face	58
7.3 Manifold	59
7.3.1 Manifold MNIST.....	59
7.3.2 Manifold Frey Face	60
7.4 Espacio latente en 2D interactivo	61
8 Conclusiones y trabajo futuro.....	62
8.1 Conclusiones.....	62
8.2 Trabajo futuro	63
Referencias	65
Glosario	67
Anexos.....	LXIX
A Optimizador Adam	LXIX
B Manual de instalación.....	LXX
C Clasificador Semi-Supervisado con VAE	LXXII
D Clasificador supervisado con imágenes originales.....	LXXIV
E Dataset MNIST [45]	LXXV

INDICE DE FIGURAS

FIGURA 1 NUBE DE PALABRAS RELACIONAS CON IA	7
FIGURA 2 COMPARACIÓN ALGORITMOS: DEEP LEARNING VS ALGORITMOS MACHINE LEARNING..	9
FIGURA 3 RECONSTRUCCIÓN DE IMAGEN AUTOENCODERS	9
FIGURA 4 ANÁLISIS DE IMÁGENES MÉDICAS[10]	10
FIGURA 5 CRONOLOGÍA DE REDES NEURONALES.....	14
FIGURA 6 NEURONA BIOLÓGICA[14]	15
FIGURA 7 ANALOGÍA ENTRE NEURONA BIOLÓGICA Y NEURONA ARTIFICIAL.[15]	15
FIGURA 8 ESTRUCTURA DE UNA RED NEURONAL SIMPLE: PERCEPTRÓN MULTICAPA (MLP).....	18
FIGURA 9 ALGORITMO: DESCENSO DE GRADIENTE	20
FIGURA 10 COMPARACIÓN DE TASAS DE APRENDIZAJE	20
FIGURA 11 GENERATIVE ADVERSIAL NETWORKS	22
FIGURA 12 INFERENCIA SOBRE VARIABLES LATENTES Y GENERACIÓN	24
FIGURA 13 ARQUITECTURA DE RED: AUTOENCODERS VARIACIONALES.....	25
FIGURA 14 ARQUITECTURA DE RED DESDE UNA PERSPECTIVA PROBABILÍSTICA[30].....	26
FIGURA 15 MAXIMIZACIÓN DEL ELBO.....	30
FIGURA 16 TRUCO DE LA RE-PARAMETRIZACIÓN	31
FIGURA 17 FUNCIÓN DE MUESTREO RE-PARAMETRIZADA [47]	32
FIGURA 18 ARQUITECTURA DEL AUTOENCODERS IMPLEMENTADO	33
FIGURA 19 DISEÑO DEL ENCODER (CODIFICADOR)	34
FIGURA 20 DISEÑO DEL DECODER (DECODIFICADOR).....	35
FIGURA 21 INICIALIZACIÓN DE PESOS Y BIAS.....	36
FIGURA 22 SHAPE PLACEHOLDER DE ENTRADA DE DATOS	41
FIGURA 23 FUNCIÓN DE PERDIDA EXPERIMENTO 1.....	47
FIGURA 24 PROCESO DE APRENDIZAJE: RECONSTRUCCIÓN DE IMAGEN ALEATORIA (EXPERIMENTO 1).....	47

FIGURA 25 FUNCIÓN DE PERDIDA EXPERIMENTO 2	48
FIGURA 26 PROCESO DE APRENDIZAJE: RECONSTRUCCIÓN DE IMAGEN ALEATORIA (EXPERIMENTO 2).....	48
FIGURA 27 FUNCIÓN DE PERDIDA EXPERIMENTO 3	49
FIGURA 28 PROCESO DE APRENDIZAJE: RECONSTRUCCIÓN DE IMAGEN ALEATORIA (EXPERIMENTO 3).....	49
FIGURA 29 FUNCIÓN DE PERDIDA EXPERIMENTO 1 (FREY FACE)	51
FIGURA 30 PROCESO DE APRENDIZAJE: RECONSTRUCCIÓN DE IMAGEN ALEATORIA (EXPE 1 FREY FACE).....	51
FIGURA 31 FUNCIÓN DE PERDIDA EXPERIMENTO 2 (FREY FACE)	52
FIGURA 32 PROCESO DE APRENDIZAJE: RECONSTRUCCIÓN DE IMAGEN ALEATORIA (EXPE 1 FREY FACE).....	52
FIGURA 33 ARQUITECTURA DE LA RED DEL CLASIFICADOR LINEAL SEMI-SUPERVISADO CON VAE	53
FIGURA 34 ARQUITECTURA DE LA RED CLASIFICADOR LINEAL SUPERVISADO.....	54
FIGURA 35 WIDGET INTERACTIVO PARA COMPARAR CLASIFICADOR LINEAL NORMAL Y CLASIFICADOR CON VAE.....	54
FIGURA 36 MUESTRAS ALEATORIAS DEL MODELO GENERATIVO DE MNIST CON DIFERENTES DIMENSIONES DEL ESPACIO LATENTE	55
FIGURA 37 ESPACIO LATENTE 2-D MOSTRADO EN DOS DIMENSIONES.....	56
FIGURA 38 ESPACIO LATENTE 2-D MOSTRADO EN TRES DIMENSIONES	57
FIGURA 39 ESPACIO LATENTE DE 2-D DEL DATASET FREY FACE MOSTRADO EN DOS DIMENSIONES	58
FIGURA 40 MANIFOLD MNIST.....	59
FIGURA 41 MANIFOLD FREY FACE.....	60
FIGURA 42 WIDGET INTERACTIVO PARA MOVERSE POR EL ESPACIO LATENTE DE DOS DIMENSIONES	61
FIGURA 43 IMÁGENES MNIST [46]	LXXV
FIGURA 44 REPRESENTACIÓN DE UN DIGITO DEL DATASET MNIST EN UN ARRAY DE NÚMEROS	LXXV

INDICE DE TABLAS

TABLA 1. FUNCIONES DE ACTIVACIÓN MÁS UTILIZADAS	17
---	----



Figura 1 Nube de palabras relacionas con IA

La inteligencia artificial (de aquí en adelante IA) es la disciplina que se encarga de crear algoritmos que replacen o simulen al ser humano en sus funciones intelectuales. Es decir, su objetivo es dotar a las máquinas de capacidad de observar, entender y actuar de una manera inteligente.

“¿Qué somos las personas sino máquinas muy evolucionadas?”
Marvin Minsky (Padre de la IA)

Todavía tendremos que esperar algunos años para poder comparar o intercambiar las capacidades intelectuales de las máquinas y de los humanos en su totalidad. Tema que no se trata en esta memoria debido a que en este momento existe un gran dilema y una amplia controversia. Si bien es cierto que, a día de hoy, existe un intenso debate sobre si la Inteligencia Artificial constituirá o no, una amenaza real en un futuro no muy lejano. Este miedo se debe, principalmente, al crecimiento exponencial que estamos viviendo de la IA que nos hace temer por una “*La rebelión de las máquinas*” al más estilo de Terminator 3[1].

Por otro lado, el aprendizaje automático es un campo de la IA que se especializa en proporcionar a los sistemas, habitualmente informáticos, la capacidad de aprender por sí mismos sin que nadie los programe explícitamente para ello. Los algoritmos de Aprendizaje Automático tienen, como fuente de conocimientos, los datos. De ahí que el Machine Learning este estrechamente relacionado con otro de los temas de la actualidad, el Big Data.

Las aplicaciones de la IA y el Machine Learning en la actualidad son incontables, ya que casi todos los dispositivos electrónicos que nos rodean están utilizando la IA sin que nos demos cuenta. El último informe de IDC (International Data Corporation) revela que *“Para 2019, el 40% de las iniciativas de transformación digital emplearán servicios de inteligencia artificial; para el año 2021, el 75% de las aplicaciones empresariales comerciales usarán IA”*[2] .

En este trabajo nos centraremos específicamente en un tipo de aprendizaje automático, el Aprendizaje Profundo o Deep Learning.

Deep Learning es una rama del Machine Learning, que habitualmente se conoce como redes neuronales profundas y tiene como objetivo principal orientar el proceso aprendizaje de forma muy similar a como aprendemos los seres humanos, simulando el funcionamiento de nuestro cerebro. Al igual que en el cerebro humano, la unidad básica de trabajo de un algoritmo de Deep Learning es la Neurona.

Los algoritmos de Deep Learning se componen de muchas neuronas artificiales que forman una red neuronal, donde las diferentes capas de neuronas interactúan entre sí del mismo modo que lo hacen las neuronas de nuestro cuerpo. Este aprendizaje no requiere de participación directa de un humano en el proceso.

A modo de ejemplo, el aprendizaje profundo es análogo al proceso de aprendizaje de un niño. A un niño nadie le enseña lo que es un coche, pero durante su proceso de aprendizaje va sacando características comunes que poseen los coches y crea un concepto coche.

El Deep Learning se define pues, de forma más formal como el conjunto de algoritmos que tienen como objetivo el modelaje de abstracciones de alto nivel, usando redes neuronales de varias capas de procesamiento.

El aprendizaje automático puede ser de dos tipos: supervisado y no-supervisado. El aprendizaje supervisado se caracteriza porque se conocen todos los datos, tanto los de entrada como los de salida. El aprendizaje no-supervisado es aquel en el que solo se dispone de los datos de entrada. Esta última variante de aprendizaje automático, el no-supervisado es el que está relacionado con el Deep Learning.

Este enfoque nos lleva a un cambio drástico en la manera de orientar los problemas, ya que ahora el científico de datos no tiene que preocuparse de especificar él mismo las características. Por ejemplo, si contiene un coche o no, sí el coche es amarillo o es verde... Además, esta visión de aprendizaje a la que estamos haciendo referencia, logra conseguir resultados que antes eran inimaginables con algoritmos de aprendizaje automáticos supervisados.

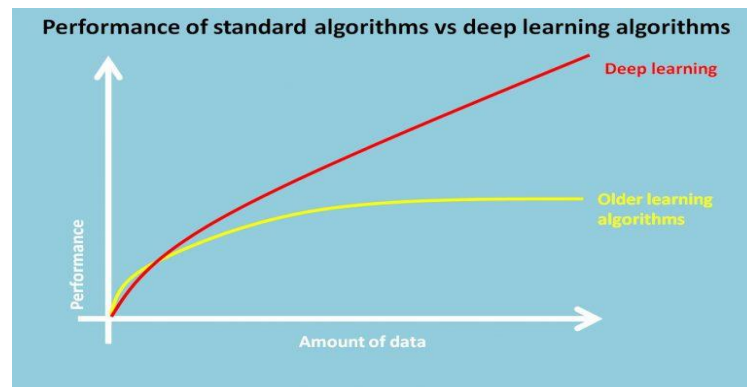


Figura 2 Comparación algoritmos: Deep Learning vs Algoritmos Machine Learning

Estos resultados son posibles gracias, entre otras cosas, a la gran cantidad de datos que manejamos en la actualidad y a la potencia de computación de la que disponemos hoy en día.

El abanico de posibilidades que nos ofrece el uso de esta disciplina abarca infinidad de temas. Como pueden ser temas de salud, bancarios, relacionados con el marketing y la publicidad, transporte...

Actualmente se han conseguido avances importantes en el campo de la Medicina con técnicas de Deep Learning [4].

Una vez hecha una pequeña introducción al Deep Learning, vamos a centrarnos en el tema específico que abarca este trabajo de fin de grado, los modelos generativos profundos concretamente los **Autoencoders Variacionales (VAE)**.

Los Autoencoders Variacionales son un tipo de red neuronal perteneciente al grupo de los Autoencoders, que al igual que estos tienen como objetivo reconstruir los datos de entrada. Para ello disponen de un codificador y un decodificador.



Figura 3 Reconstrucción de imagen Autoencoders

Además, los Autoencoders Variacionales permiten ser modelos generativos con resultados muy satisfactorios, a diferencia de los Autoencoders tradicionales que no funcionan tan bien para funciones generativas.

Este Trabajo de Fin de Grado se centra en estas redes neuronales que están teniendo tanto impacto en la Comunidad Científica, en el área de la Inteligencia Artificial en general, y en la del Machine Learning en particular.

1.1 Motivación

El aprendizaje profundo está viviendo una gran explosión en los últimos años. Sin lugar a duda ha roto con todos los techos existentes hasta el momento, con métodos tradicionales de Machine Learning.

Con la aplicación del aprendizaje no-supervisado, usando redes neuronales, se han conseguido avances en muchos ámbitos, que antes eran impensables. Hecho que está revolucionando, por ejemplo, la sanidad [11], con la aparición de técnicas de Deep Learning aplicadas directamente en los entornos de trabajo del personal sanitario. Estas técnicas ayudan a los médicos a realizar diagnósticos más rápidos y más precisos. Así como predecir enfermedades a tiempo para poder tratarlas.



Figura 4 Análisis de imágenes médicas[10]

Este proyecto de fin de grado se ha realizado con el objetivo de crear un algoritmo de aprendizaje no-supervisado con técnicas de Deep Learning que sea un modelo generativo con buenos resultados con el fin de poder utilizarlos en tareas reales muy diversas.

Los modelos generativos, y en especial los Autoencoders Variacionales, se están utilizando para dibujar imágenes, para trabajar de forma colaborativa con sistemas de recomendación[5], para modelar las reacciones de la audiencia a las películas[6], para clasificación de textos de forma semi-supervisada[7], para recuperar imágenes de teledetección en alta resolución [8] y para analizar imágenes médicas[9].

Estas son solo algunas de las aplicaciones que tiene los Autoencoders Variacionales en la actualidad.

Por esta razón este trabajo se centrará en los VAE puesto que es un tipo de red neuronal muy utilizado en la actualidad y que está consiguiendo resultados muy prometedores en varios entornos de nuestra sociedad.

1.2 Objetivos,

El propósito de este trabajo es implementar un Autoencoder Variacional, que arroje buenos resultados con la finalidad de ser utilizado con éxito en tareas de la vida real.

De forma detallada, los objetivos de este proyecto son:

- Definir sobre qué conjunto de datos se va a trabajar. En este caso se utilizarán con los dataset MNIST y FREY FACE
- Investigar los diferentes tipos de Autoencoders/Autoencoders Variacionales que hay en la actualidad, sus características de modelaje, así como sus resultados.
- Investigar líneas de mejora de los algoritmos ya existentes.
- Diseñar modelo.
- Implementar modelo.
- Realizar pruebas exhaustivas sobre el modelo creado.
- Evaluar los resultados obtenidos.
- Sacar conclusiones.

1.3 Organización de la memoria

La memoria consta de las siguientes secciones:

- **Estado del arte**
En este apartado se hará una breve introducción al concepto de redes neuronales y su analogía con el cerebro humano. Se realizará un repaso a la historia y actualidad de las redes neuronales. Se finalizará profundizando en la actualidad de los modelos generativos y en particular en los Autoencoders Variacionales.
- **Diseño**
En este apartado se detallará de forma minuciosa el diseño elegido para implementar el modelo generativo de tipo VAE.
- **Desarrollo**
En este apartado se describirá detalladamente como se ha realizado la implementación de la red neuronal dando lugar a un Autoencoder Variacional totalmente funcional.
- **Integración, pruebas y resultados**
En este apartado se integrarán todas las partes del desarrollo y se realizarán pruebas exhaustivas para obtener resultados para un gran número de configuraciones diferentes. También se analizar los resultados obtenidos tanto de forma visual como escrita.
- **Visualizaciones de los datos**
En este apartado se utilizarán herramientas gráficas que servirán para comprender los datos desde una perspectiva visual
- **Conclusiones y trabajos a futuro**
En este apartado se llevará a cabo una reflexión sobre los resultados obtenidos, así como sus posibles aplicaciones. Se finalizará proponiendo nuevas líneas de investigación para el futuro.

2 Estado del arte

2.1 Redes neuronales

Las redes neuronales son modelos computacionales que simulan el comportamiento del cerebro humano. Las unidades básicas se llaman, Neuronas. Estas funcionan de forma conjunta, interconectadas entre sí para resolver problemas que no tienen un algoritmo definido para convertir una entrada en una salida deseada.

2.1.1 Cronología de las redes neuronales[12]

A lo de la historia han sido numerosos los científicos que han perseguido el sueño de construir maquinas capaces de realizar tareas con inteligencia, simulando el funcionamiento del cerebro humano. Fueron los autómatas la primera aproximación a esas máquinas que aspiraban hacer cosas de forma inteligente simulando alguna función del ser humano. La persecución de este sueño ha guiado a muchos científicos en su trabajo hasta la actualidad y hoy en día lo denominamos **Inteligencia Artificial**.

Pese a que las primeras explicaciones sobre el cerebro datan de la época de Heron de Alejandría (100 a.C), Platón (427-347 a.C) y Aristóteles (348.422 a.C). No fue hasta el 1936 cuando Alan Turing empezó a interesarse por el cerebro como forma de ver la computación.

Cronología de las redes neuronales:

- **Alan Turing-1936**
Fue el primero en interesarse por el cerebro como forma de ver la computación.
- **Warren McCulloch y Walter Pitts**
Fueron los primeros en dar teorías de la computación neuronal. Creando una red neuronal simple mediante circuitos eléctricos
- **Donald Hebb-1949**
Fue el primero en explicar los procesos de aprendizaje. Aún hoy está presente en la mayoría de las redes neuronales con la regla de aprendizaje de Hebb[13].
- **Karl Lashley-1950**
En sus ensayos, descubrió que la información en nuestro cerebro no está de forma centralizada, sino que está distribuida.
- **Congreso de Dartmouth-1956**
En este congreso se empieza hablar del nacimiento de la Inteligencia artificial.
- **Frank rosenblatt-1957**
Empezó el desarrollo del Perceptrón. Se trata de la red neuronal más antigua.
- **Bernard Widrow y marcial Hoff-1960**
Desarrollaron el modelo Adaline que fue la primera red neuronal utilizada en la vida real para eliminar el eco en las líneas telefónicas.
- **Karl Steinbeck y Die Lernmatrix-1961**
Desarrollaron una red neuronal que simulaba la memoria asociativa.
- **Stephen Grossberg-1967**

Creo la red neuronal llamada Avalancha, que permitía el reconocimiento continuo de habla y el aprendizaje de los brazos de un robot.

- **Marvin Minsky y Seymour Papert-1969**
Estos dos investigadores van a para el crecimiento de las redes neuronales por unas fuertes críticas al perceptrón hasta el 1982. A grandes rasgos, detractaron esta red neuronal porque no era capaz de resolver problemas simples como funciones no lineales o el XOR-exclusive.
- **Paul Werbos-1974**
Empezó a desarrollar la idea del aprendizaje de propagación hacia atrás.
- **Stephen Grossberg-1977**
Teoría de la resonancia adaptada. Es una arquitectura de red que simula la memoria a corto y largo plazo del cerebro.
- **Kunihiko Fukushima-1980**
Crea una red neuronal para reconocer patrones visuales.
- **John Hopfield-1982**
En su persona se reconoce el renacimiento de las redes neuronales con su obra “*Computación neuronal de decisiones en problemas de optimización*”
- **A partir de 1986**
El crecimiento de las investigaciones sobre redes neuronales fue en aumento.
- **David Rumelhart y G.Hinton-1986**
Definieron el algoritmo de aprendizaje de propagación hacia atrás (Backpropagation).
Aparece por primera vez la idea de los Autoencoders.
- **Diederik P. Kingma y Max Welling-2013**
Aparecen los Autoencoders Variacionales
- **Actualidad**
Son abundantes los estudios e investigaciones que se publican cada año sobre redes neuronales. Así como empresas que se suman a esta transformación digital utilizando redes neuronales.

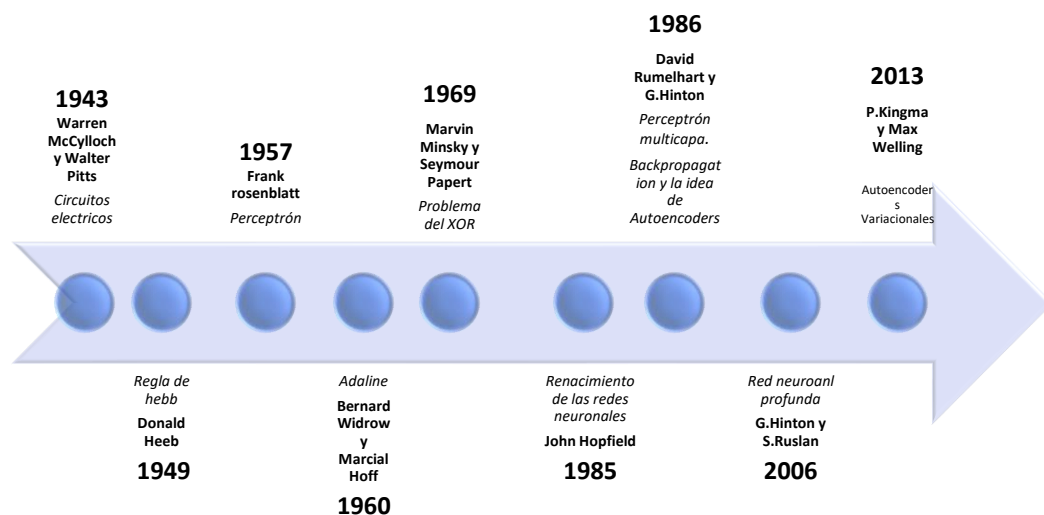


Figura 5 Cronología de redes neuronales

2.1.2 Analogía entre Neurona Artificial y Neurona Biológica

De forma análoga al concepto de neurona biológica en el cerebro humano. Las neuronas artificiales son la unidad básica dentro de una red neuronal (cerebro humano). Estas tienen como propósito simular el comportamiento de las neuronas biológicas en el cerebro.

El cerebro está formado de millones de neuronas que se conectan entre sí, transmitiendo y recibiendo información de una a otra a través de la sinapsis.

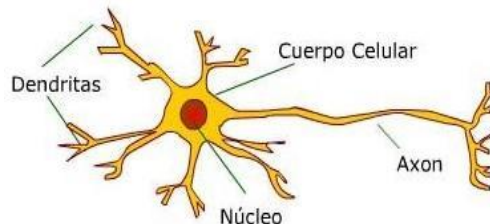


Figura 6 Neurona Biológica[14]

La sinapsis es el proceso por el cual se relacionan dos neuronas y se produce cuando una neurona transmite un impulso nervioso, a través de su axón, a otra neurona receptora por sus dendritas. Durante el proceso de la sinapsis se liberan una sustancia química que se denominan neurotransmisores.

Esta sustancia química actúa como una llave que regula el impulso eléctrico. Facilitando su paso (Neurotransmisores excitatorios) en algunos casos o dificultándolo en otros (Neurotransmisores inhibitorios). La neurona receptora del impulso recibe múltiples señales de diferentes sinapsis que al unir las todas, marcan el nivel de activación (Potencia postsináptica). El nivel de activación es el que marca la intensidad con que la neurona transmitirá el impulso a una nueva neurona.

Una vez hecha una breve y simplificada explicación del funcionamiento de una neurona biológica podemos realizar una analogía con las neuronas artificiales.

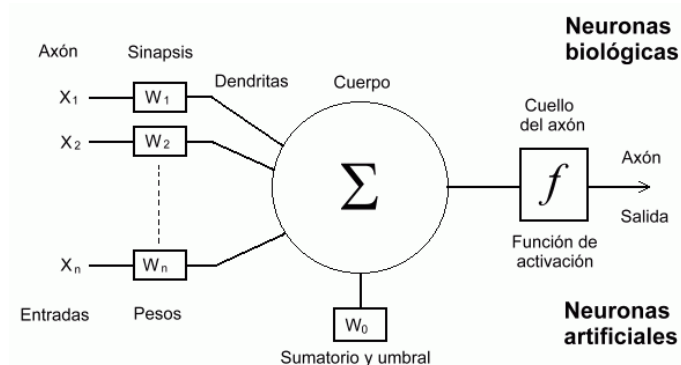


Figura 7 Analogía entre neurona biológica y neurona artificial.[15]

Las entradas de la neurona artificial serían los impulsos que envían una neurona biológica a través de su axón. Los pesos serían la intensidad de la sinapsis (Eficacia sináptica). El signo de los pesos serían los neurotransmisores. El producto de los pesos por las entradas da lugar a la entrada de la función de activación que sería la potencia postsináptica. La salida (ej.: salida de la función sigmoide) sería la respuesta al estímulo que ha recibido por su entrada y sería análogo al axón.

2.1.3 Funcionamiento de una Neurona Artificial

Una única neurona por sí misma no puede resolver problema complejo, pero al combinarse con miles o millones de ellas son capaces de resolver problemas muy complejos con resultados eficaces.

Las neuronas artificiales están interconectadas las unas con las otras, siendo las salidas de unas las entradas de otras. Es **la función de red o función de ponderación** la que se encarga de transformar las entradas de una neurona en una única entrada global a ella, para esto necesita combinar las entradas con los pesos. Las entradas se multiplican por los pesos, estos últimos actúan como un potenciómetro, regulando la intensidad de influencia de esa entrada dentro de la neurona. Es por eso por lo que los pesos están en continuo ajuste (por ejemplo: Backpropagation) durante el proceso de aprendizaje. Un ejemplo de función de red podría ser:

$$funcion\ de\ red = \sum_{i=1}^n x_i \cdot w_i$$

Existen otros operadores aplicables a la función de red en lugar de la sumatoria como pueden ser el máximo, mínimo, el productoria...

Una vez se ha calculado la función de red, la salida de esta, se conecta con **la función de activación** que se encarga de calcular el nivel de activación de una neurona. Una neurona puede estar activa o inactiva, de la misma forma en que una neurona biológica está excitada o inhibida.

Para calcular dicho nivel de activación, esta función transforma la entrada global que previamente ha calculado la función de red, en un estado de la neurona, o bien activa o bien inactiva.

La salida de la función de activación se combina con un valor umbral, que se trata de un parámetro que regula si la neurona emite una señal o no. Si la salida de la función de activación es mayor que el valor del umbral, la neurona enviará la señal, en caso contrario la neurona no emitirá señal.

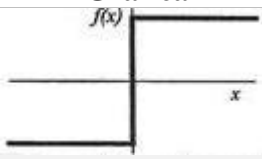
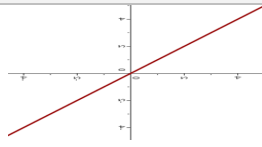
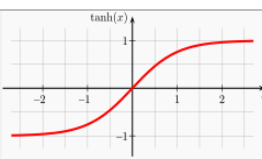
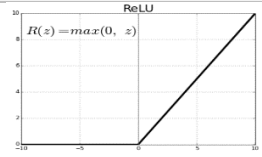
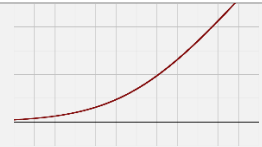
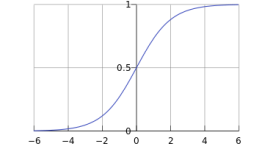
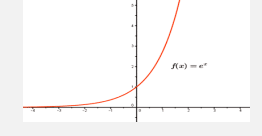
Funciones	Definición	Dominio	Gráfica
Binaria o Escalón	$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	$(0,1)$	 <p>Función Binaria</p>
Identidad	$f(x) = x$	$(-\infty, \infty)$	 <p>Función Identidad</p>
Tangente Hiperbólica	$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$(-1,1)$	 <p>Función Tangente Hiperbólica</p>
Relu (Rectificadora)	$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$	$(0, \infty)$	 <p>Relu (Rectificadora)</p>
Softplus	$f(x) = \ln(1 + e^x)$	$(0, \infty)$	 <p>Función Softplus</p>
Sigmoide	$f(x) = \frac{1}{1 + e^{-x}}$	$(0,1)$	 <p>Función sigmoide</p>
Exponencial	$f(x) = e^x$	$(0, \infty)$	 <p>Función exponencial</p>

Tabla 1. Funciones de Activación más utilizadas

2.1.4 Estructura de las Redes neuronales

Las neuronas artificiales se organizan dentro de la red neuronal en capas.

Generalmente las redes neuronales constan de tres capas: una capa de entrada, una o varias capas ocultas y una capa de salida.

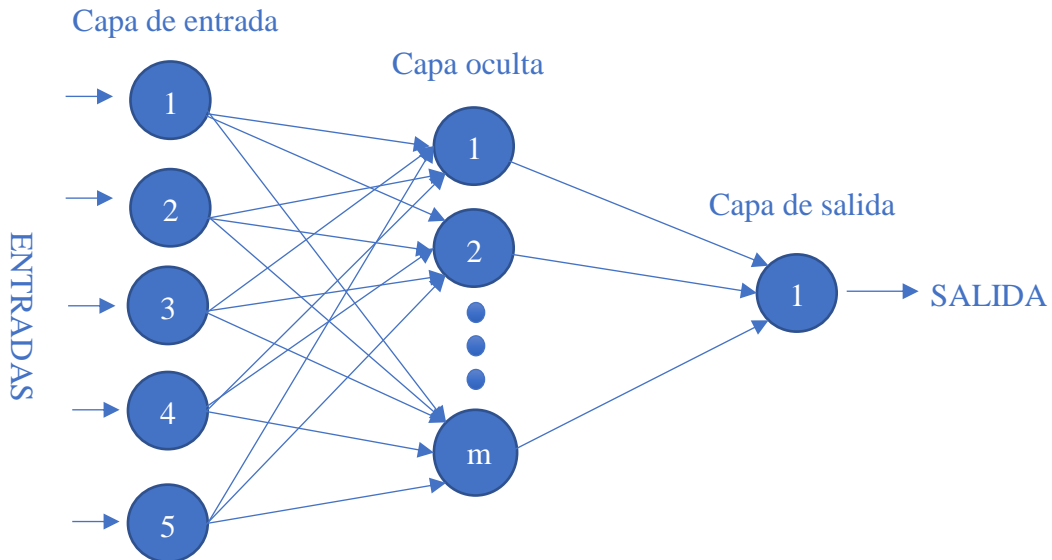


Figura 8 Estructura de una red neuronal simple: Perceptrón multicapa (MLP)

La capa de entrada es la que recibe los datos del exterior mientras que las capas ocultas procesan la información y se conectan con otras capas. La capa de salida se encarga de enviar información ya procesada al exterior.

2.1.5 Propagación de la Información y proceso de aprendizaje en redes neuronales

Una red neuronal aprende durante su fase de entrenamiento, que no es más que ajustar los pesos y el umbral de la red para conseguir los resultados deseados.

La búsqueda de estos parámetros es un proceso adaptativo e iterativo, por el cual la red va adquiriendo mejores resultados, ajustando los valores de estos parámetros y midiendo la diferencia entre los resultados obtenidos y los resultados deseados.

La actualización de los pesos y el umbral¹ se lleva a cabo mediante propagación de la información a lo largo de toda la red. Existen dos tipos de propagación: Propagación hacia delante y propagación hacia atrás o retropropagación (Backpropagation).

Algoritmos de propagación

- **Propagación hacia delante:** En esta fase la red transmite los valores de la capa entrada hacia delante a lo largo de toda la red.
- **Retropropagación:** Esta fase corresponde con la fase de aprendizaje, que tiene como objetivo propagar los errores medidos en la salida de la red hacia atrás con el fin de actualizar los parámetros de la red (Pesos y el umbral).

Las funciones que miden el error entre la salida obtenida y la salida deseada, se denominan función de pérdida o función de error. Hay diversas funciones de pérdida como el error medio cuadrático[18], la entropía cruzada[19], divergencia de Kullback-Leibler[21], máxima verosimilitud...

El objetivo del aprendizaje de una red neuronal es por lo tanto minimizar la función de pérdida. Aunque cabe destacar que en ocasiones se desea, en su lugar, maximizar una función de pérdida como es el caso de la función de pérdida del Lower Bound (ELBO) que se tratará más adelante.

Para conseguir minimizar la función de pérdida se utilizan algoritmo de optimización que van cambiando poco a poco ([Tasa de aprendizaje](#)) los parámetros de la red (Actualización de pesos y bias) en dirección donde se minimice la pérdida o error.

Uno de los algoritmos más utilizados es el **descenso de gradientes**[16] que, de forma simplificada, se trata de un algoritmo iterativo que va dando pequeños saltos (Dependiendo de una tasa de aprendizaje) en dirección negativa al gradiente² en busca del mínimo de la función a optimizar.

¹ El parámetro umbral pasa a llamarse “bias” en el proceso de entrenamiento y se considera como un peso sináptico (w_0) más asociado a la neurona.

² El gradiente de forma sencilla es la pendiente que representa, la tangente en un punto de la función

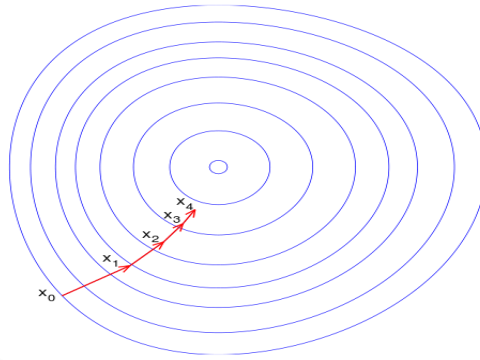


Figura 9 Algoritmo: Descenso de Gradiente

La velocidad a la que se actualizan los pesos de una red viene dada por la tasa de aprendizaje (Learning rate). El valor de la tasa de aprendizaje suele estar entre 0 y 1.

Los valores muy próximos a 0 hacen que los pesos cambien lentamente, por lo tanto, la red tardará más tiempo en encontrar el mínimo (Convergencia), mientras que los valores próximos a 1 hacen que los pesos se actualicen rápidamente y, en consecuencia, la red encuentre el mínimo con más rapidez. En ocasiones utilizar tasas de aprendizajes grandes puede provocar oscilaciones que dificulten o incluso imposibiliten encontrar el mínimo y como consecuencia los resultados serán pésimos.

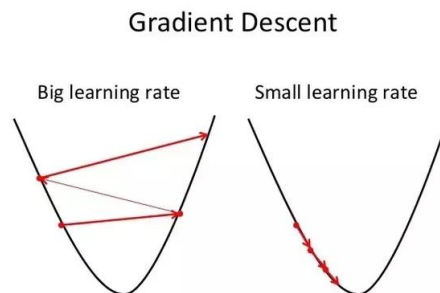


Figura 10 Comparación de tasas de aprendizaje

2.1.6 Modelos generativos: Autoencoders Variacionales

Existen numerosos tipos de redes neuronales y enfoques dentro del mundo del Machine Learning y del Deep Learning, abordar todos ellos en este trabajo de fin de grado, sería muy extenso y nos desviaríamos del objetivo del mismo, que no es otro que profundizar en un tipo concreto, los **modelos generativos profundos**.

Los modelos generativos profundos representan un área amplia dentro del Machine Learning, son un tipo de modelos de aprendizaje no-supervisado que tiene como objetivo generar datos.

Algunos de los problemas que han presentado los modelos generativos a lo largo de su historia son, por ejemplo, los que han requerido fuertes suposiciones sobre la estructura de los datos. Además, han supuesto problemas computacionales sobre todo en procesos de inferencia costosos como Monte Carlo.

Últimamente se están consiguiendo avances muy prometedores en este sentido, resolviendo el problema de las suposiciones fuertes sobre la estructura de datos y consiguiendo un entrenamiento rápido, usando técnicas de Backpropagation. Una de las redes neuronales más populares en esta dirección, es **el Autoencoder Variacional**, tema en el cual se centra este trabajo de fin de grado.

Los Autoencoders Variacionales son un tipo de redes neuronales, relativamente modernas (2013), y en constante evolución. Pertenecen a la familia de los Autoencoders (1986), tienen una estructura muy similar a un perceptrón multicapa y, se utilizan principalmente en aprendizaje no-supervisado. Lo que significa que solo necesita datos de entrada sin etiquetar. Su objetivo principal es reconstruir los datos de entrada, sin olvidar su interés generativo. Para ello disponen de un codificador y un decodificador.

Gracias al pequeño error que tiene en sus aproximaciones, son consideradas como uno de los enfoques más populares dentro del aprendizaje no-supervisado, en concreto de distribuciones de datos complicadas. Esta cualidad le está permitiendo seguir ganando adeptos.

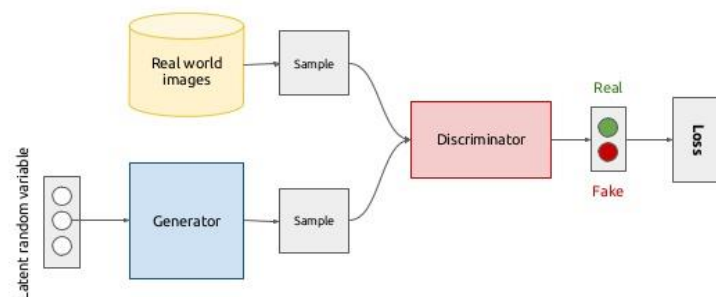
Actualmente se relaciona este tipo de modelos generativos con el arte y los diseñadores gráficos. Ya que permite generar más ejemplos de los que se poseen inicialmente, pero con algunas diferencias. Un ejemplo aplicado al sector de los videojuegos sería generar a partir de una base de datos de modelos 3D de árboles, un bosque repleto de ellos y siendo todos similares, pero no iguales. También se podría tomar un texto escrito a mano y generar más texto. Otro uso muy generalizado es la eliminación de ruido para reconstruir datos dañados, especialmente en imágenes.

Los avances más notables con este tipo de modelo generativo vienen desde ámbito del arte, un ejemplo de ello se ha producido en el 2017, en el que se ha conseguido generar música mediante un Autoencoder Variational[25]. Pese a ello cabe destacar que, en otras áreas, tan importantes como la de la salud, se han conseguido avances muy prometedores al respecto. Uno bastante relevante ha sido la extracción de un espacio latente biológicamente relevante, de transcriptomas de cáncer con Autoencoders Variational.[26]

En la actualidad, hay otro modelo generativo confrontado con los Autoencoders Variacionales por ser el dominador en este terreno tan candente: las redes generativas adversas (GAN³).

Los **GAN** están formados por dos redes neuronales: un generador y un discriminador que compiten entre sí en un juego de suma cero⁴. Una analogía muy usada para su explicación en este campo [28] es considerar un falsificador (Generador) que trata de generar dinero falso, mientras, por otro lado, un detective (Discriminador) intenta detectar si el dinero es falso o es real. Estas dos personas están enfrentadas, de ahí el nombre de estas redes (Generative Adversarial Networks). Para salir victorioso, el falsificador tiene que generar dinero falso tan real como sea posible, en cambio, el detective tiene que ser muy bueno distinguiendo el dinero falso del real. Es fácil ver gracias con esta analogía, que la red va aprendiendo acorde el falsificador y detective (Generador y discriminador) van aprendiendo.

Generative adversarial networks (conceptual)



5

Figura 11 Generative Adversial Networks

³ GAN: Generative Adversial Networks

⁴ Juego de suma cero [27]: Lo que unos jugadores ganan otros lo pierden en la misma proporción. Es decir, si damos un valor positivo a las ganancias y un valor negativo a las perdidas, la suma total es cero.

3 Autoencoders Variacionales

Como se ha introducido en el apartado anterior, en la actualidad hay dos modelos generativos confrontados por dominar en este campo: las redes generativas adversas y los Autoencoders Variacionales.

Ambos modelos presentan diferencias en su arquitectura de red, así como en la manera que tienen de entrenar los modelos. Los modelos basados en redes generativas adversas están basados en la teoría del juego (Juegos de suma cero) y su propósito principal es encontrar el equilibrio entre la red discriminadora y la red generadora. Por otra parte, los Autoencoders Variacionales se basan en técnicas de inferencia bayesiana.

Los Autoencoders Variacionales son una técnica de aprendizaje profundo no-supervisado que aprenden representaciones latentes de los datos observados, con la finalidad de entender mejor los datos observados y usar estas representaciones para generar datos adicionales. De manera más formal, son modelos que generan ejemplos X , siguiendo una distribución desconocida $P(X)$. La finalidad es, por lo tanto, aprender un modelo del cual podamos tomar muestras que sean tan similares a $P(X)$ como sea posible.

“What I cannot create, I do not understand”

Richard Feynman

3.1 Funcionamiento de un Autoencoder Variacional

Para dar una aproximación a los Autoencoder Variacionales, supongamos por un momento, que deseamos generar rostros humanos. Lo primero que debemos hacer es imaginar qué características tienen los rostros humanos, a partir de los que ya hemos visto alguna vez. Tiene ojos, tiene boca, está sonriendo, tiene los ojos azules, es un rostro masculino o femenino... Una vez que tenemos una idea firme de cómo son los rostros humanos podemos tomar muestras y generar otros rostros humanos.

Dado que nuestra imaginación ha creado un vector de variables latentes a partir de todos los rostros que hemos visto alguna vez con esta forma [ojos azules, barbilla afilada, nariz redonda, boca pequeña, pómulos afilados, cejas rubias y voluminosas...]. A partir de este vector de variables latentes podemos generar otros rostros simplemente tomando muestras de este vector.

Un Autoencoder funciona a grandes rasgos de forma análoga, todos los rostros de los seres humanos de los que tenemos constancia serían análogo a nuestros datos de entrada, nuestra imaginación sería análogo al espacio latente o variables latentes⁵, que se obtienen de los

⁵ Variable latente: Son variables que no son observadas explícitamente, sino que son inferidas por datos observados directamente.

datos de entrada, y los rostros generados por nuestra imaginación sería los datos devueltos por nuestra red.

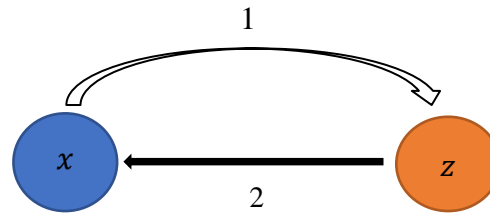


Figura 12 Inferencia sobre variables latentes y generación

De forma muy simplificada el funcionamiento de un Autoencoder se representa en la Figura 19. Los X son los datos observados mientras que las Z son variables latentes. Nosotros queremos inferir las variables latentes desde puntos observados (1). Y posteriormente con las variables latentes se pueden generar datos (2). En realidad, nuestras variables latentes son una distribución de probabilidad, hecho en el que más tarde se profundizará.

3.2 Estructura de la Red Neuronal

Como se ha introducido en el apartado anterior un Autoencoders variacional es una red neuronal que está formada por un codificador, un decodificador y un espacio latente. El codificador corresponde a una capa, o varias, de convolución que codifican la entrada en un espacio latente, es decir, estas capas de convolución extraen características de alto nivel. El decodificador, sin embargo, corresponde a una o varias capas de deconvolución que a partir de las características de alto nivel genera nuevos datos.

Podemos decir que los Autoencoders Variacionales son una composición de redes neuronales, que como ya se puede deducir, está compuesta por dos redes que son el codificador y el decodificador; ambas tienen como punto de conexión el espacio latente o variables latentes. Aunque a menudo veamos esta composición de redes como una sola debemos tener presente esta connotación. Ambas redes neuronales (codificador y decodificador) tienen una arquitectura igual que la de un perceptrón multicapa (MLP).

La clave del funcionamiento de un Autoencoder variacional reside en sus tres componentes principales: El codificador, el espacio latente y el decodificador.

- **Codificador**

Es la parte de la red que se encarga de capturar información de alto nivel sobre los datos de entrada, estos datos de alto nivel son las variables latentes. Además, realiza una representación comprimida de los datos de entrada, fenómeno que se denomina reducción de la dimensionalidad. Su cometido es fundamental a la hora de obtener

buenos resultados en la fase de decodificación, ya que debe crear variables latentes ricas en información.

- **Espacio latente**

Son la representación de alto nivel de las características extraídas de los datos de entrada. Puesto que las variables latentes son en realidad una distribución de probabilidad, necesitaremos tomar muestras de este espacio latente para generar datos. Son la entrada del decodificador.

- **Decodificador**

Parte de la red que, a partir de las variables latentes aprendidas por el codificador, reconstruye las entradas.

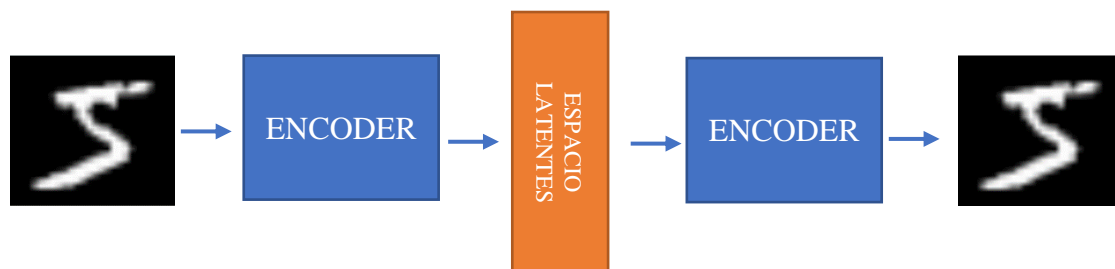


Figura 13 Arquitectura de red: Autoencoders Variacionales.

Los Autoencoders Variacionales [20] se diferencian de los Autoencoders típicos en que los primeros aprenden una distribución de probabilidad del espacio latente y, por lo tanto, se pueden utilizar para tareas generativas. Es decir, el codificador genera variables latentes que siguen una distribución, normalmente gaussiana. Esta restricción es la que lo diferencia de un Autoencoder típico, en el que su código latente presenta aleatoriedad.

3.3 Fundamentos matemáticos

Una vez dada una explicación de los Autoencoders, desde un punto de vista más teórico, en esta sección abarcaremos una explicación desde una perspectiva matemática.

Todas las redes neuronales están fundamentadas en unos principios matemáticos que las explican. En el caso de los Autoencoders Variacionales se encuentra bajo un marco claramente probabilístico. Los VAE⁶ son un modelo probabilístico gobernado por los datos de entrada X , y las variables latentes Z .

En el prolegómeno de la explicación teórica de los VAE se ha expuesto que estos constan de tres partes fundamentales: el codificador, el decodificar y las variables latentes. Pues bien, desde un punto de vista matemático vamos añadir un cuarto elemento fundamental, la

⁶ VAE: Autoencoders Variational

función de pérdida, que será la encargada de medir el error de nuestra red, la cual explicaremos en breve.

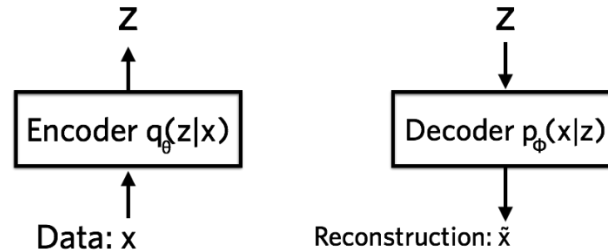


Figura 14 Arquitectura de red desde una perspectiva probabilística[30]

Dada esta representación de la red desde una visión probabilista, se pueden redefinir las partes de esta en términos probabilísticos. El codificador aprende una representación a alto nivel de los datos que, traducido a términos probabilístico, consiste en aprender una distribución de probabilidad (normalmente con una gaussiana) de los datos x , esto se representa en un espacio latente. Por este motivo el codificador se puede expresar como $Q_{\theta}(z|x)$, donde θ representa los pesos y sesgos del codificador.

Las variables latentes Z , en términos probabilísticos se denotan de la misma manera. Por otra parte, el decodificador se puede definir como $P_{\phi}(x|z)$. Esta toma como entrada la representación de alto nivel dada por las variables latentes y reconstruye las entradas siguiendo una distribución de probabilidad, normalmente una distribución de Bernoulli.

Para clarificar este tema, de la misma manera que cuando se aprende un lenguaje de programación nuevo se realiza el típico ejemplo “Hola mundo”, en redes neuronales se suele utilizar el dataset MNIST⁷ para explicar modelos a modo de ejemplo.

Tenemos una imagen de entrada X , del dataset MNIST con 784 dimensiones (28x28), el codificador aprende características de alto nivel de esa imagen en un espacio latente Z que es considerablemente más pequeño en dimensión que X . Este fenómeno se conoce como reducción de la dimensionalidad. El decodificador toma este espacio latente Z como entrada y devuelve una imagen de 784 dimensiones que representa la reconstrucción de la imagen de entrada X dada su espacio latente Z .

Para medir cuantitativamente como de bien, o de mal, está aprendiendo nuestra red, debemos medir, por un lado, la capacidad que tienen el codificador de representar las características de alto nivel de la imagen X en un espacio latente z , siguiendo una distribución $Q_{\theta}(z|x)$. Y, por otra parte, medir la competencia del decodificador para reconstruir una imagen de

⁷ MNIST: es un dataset de imágenes de dimensión 28x28 (784 Píxeles), de dígitos escritos a mano comprendidos entre el 0 y el 9.

entrada x dados su espacio latente z , siguiendo una distribución $P_{\theta}(x|z)$. A esta medida nos referimos como función de pérdida.

Ahora que se han redefinido los componentes de un VAE en termino de probabilidad, podemos explicar los fundamentos matemáticos detrás de los VAE con una nomenclatura probabilística.

Retomando la analogía que introducíamos en la [sección 3.1](#) entre nuestra imaginación y los Autoencoders Variacionales, se puede ir un paso más allá, y definir esta analogía en términos probabilísticos:

- x : serían todos los rostros humanos.
- $P(X)$: sería la distribución de probabilidad que siguen nuestros datos, es decir, las características de alto nivel que se extraen de los rostros humanos.
- z : sería análogo a nuestra imaginación.
- $P(x|z)$: sería la distribución de los datos generados dada la variable latente, es decir, el rostro generado a partir de nuestra imaginación.

Atendiendo puramente a temas probabilísticos un VAE es un modelo regido por los datos x y las variables latentes z . De tal forma que la probabilidad conjunta del modelo viene definida como:

$$P(x, z) = P(x|z) P(z)$$

Supongamos un $P(z)$ muy simple del que podemos tomar muestras, podemos generar x similares usando una distribución $P(x|z; \theta)$. Por ejemplo, siendo $P(x|z; \theta)$ una distribución gaussiana⁸ factorizada con parámetros dados por un perceptrón multicapa.

$$P(x|z; \theta) = \prod_{d=1}^D \text{Normal}(x_d | \mu_d(z; \theta), \sigma_d^2(z; \theta))$$

Nuestro objetivo es encontrar un θ que maximice $P(z|x)$ para crear variables latentes z ricas en información, con la finalidad de usar estas variables latentes aprendidas para tomar muestras.

Lamentablemente $P(z|x)$ es intratable, ya que la derivada del marginal likelihood (Evidencia) $P(x)$, lo es.

$P(x)$ se obtiene de la expresión que calcula la probabilidad posterior real, $P(z|x)$ por el teorema de Bayes⁹:

$$P(z|x) = \frac{P(x|z)P(z)}{P(x)}$$

⁸ Distribución Gaussiana, también denominada normal es una distribución muy utilizada en estadística, como en el caso de la inferencia estadística. [31]

⁹ Teorema de Bayes: se trata de la probabilidad condicionada por un evento aleatorio. [32]

Si marginamos las variables latentes podemos calcular $P(x)$ como:

$$P(x) = \int P(x|z) P(z) dz$$

Como hemos dicho calcular esta integral es intratable, ya que requiere tiempo exponencial para ser calculada, por consiguiente $P(z|x)$ es intratable. Además, existe otro inconveniente y es que los dataset pueden ser muy grandes y no caben en memoria.

Es aquí donde aparece el Autoencoder Variacional solucionando el problema de la intratabilidad, que presenta la probabilidad posterior real, $P(z|x)$. Además, soluciona el problema de dataset muy grandes usando minibatches¹⁰ en lugar del dataset completo.

Para ello los VAE añaden una red de reconocimiento $Q_\phi(z|x)$, con la finalidad de aproximar esta distribución, a la distribución posterior real $P(z|x)$. En otras palabras, un VAE añade una red que sea capaz de aproximar (Inferir) la distribución posterior real $P(z|x)$, usando una distribución más simple y tratable, representada por $Q_\phi(z|x)$. Esta red no es otra que nuestro ya conocido codificador.

Para inferir la distribución posterior real $P(z|x)$, en los Autoencoders Variacionales se usa un método denominado, Inferencia Variacional (VI). Que aproxima la distribución posterior real con una familia de distribuciones $Q_\phi(z|x)$.

3.3.1 Inferencia Variacional

La Inferencia Variacional es uno de los métodos más utilizados de inferencia bayesiana. Otros métodos muy populares son Metropolis-Hastings [33] y Monte Carlo [34]. Tanto Metropolis-Hastings como Monte Carlo son métodos basados en técnicas de muestreo [34]. Los métodos de muestreo presentan algunas deficiencias, pese a que son capaces de encontrar una solución óptima con el tiempo suficiente es difícil saber si la solución que dan es buena debido al tiempo limitado que tiene en la práctica.

Las diferencias principales entre las técnicas de muestreo y la inferencia variacional residen en que los métodos de muestreo son capaces de encontrar soluciones óptimas con suficiente tiempo y práctica, mientras que con técnicas Variacionales es difícil encontrar esta solución. Pero sin embargo con técnicas Variacionales sabemos cuándo han convergido y habitualmente son más adecuados para técnicas de optimización del gradiente estocástico.

La idea principal de inferencia variacional reside en transformar un problema de aproximación a una distribución condicional compleja en un problema de optimización.

En nuestro caso tenemos una distribución de probabilidad intratable $P(z|x)$ que denotaremos como P . El método de inferencia variacional tratará de resolver un problema de optimización encontrando una distribución q , dentro de una familia de distribuciones Q_ϕ tratables y simples, que sea muy cercana a la distribución de probabilidad P . Resumiendo,

¹⁰ Minibatches: son porciones de menor tamaño del dataset completo.

el problema de optimización consiste en encontrar una distribución simple y tratable que sea muy similar a P .

Para poder usar $q \sim Q$ en lugar de P , obteniendo así una solución aproximada.

El parámetro variacional ϕ depende de la familia de distribuciones. Si por ejemplo q es una gaussiana, el parámetro ϕ corresponde con los parámetros de una gaussiana, la media y la varianza de las variables latentes para cada punto de los datos.

$$\phi = x \sim N(\mu, \sigma^2)$$

Para formular la inferencia como un problema de optimización demos medir la similitud entre q y P , es decir, como de bien se aproxima q a P . Para medir la diferencia entre dos distribuciones se utiliza la divergencia de Kullback-Leibler (KL), que mide la información que se pierde cuando se usa q aproximando P .

La divergencia de Kullback-Leibler [35] es una medida de la diferencia entre dos distribuciones de probabilidad, por lo tanto, es siempre positiva o igual a cero. Se define como:

$$D_{KL}(q_\phi(z|x) || p_\theta(z|x)) = E_q[\log q_\phi(z|x)] - E_q[\log p_\theta(x, z)] + \log p(x)$$

Nuestro cometido es por lo tanto resolver este problema de optimización:

$$q_\phi(z|x) = \arg \min D_{KL}(q_\phi(z|x) || p_\theta(z|x)) \text{ siendo } q \sim Q$$

Podemos observar que nuevamente aparece el problema de la intratabilidad en la divergencia KL, debido a la evidencia $P(x)$, que como ya sabemos es intratable.

Como no podemos calcular la divergencia KL directamente, optimizamos un objetivo alternativo llamado límite inferior variacional (ELBO), que si es tratable:

$$ELBO(\phi, \theta) = E_q[\log p_\theta(z, x)] - E_q[\log q_\phi(z|x)]$$

El termino ELBO proviene del logaritmo del marginal likelihood de los datos y se trata del negativo de la divergencia mas $\log(P(x))$, es decir:

$$ELBO(\phi, \theta) = -D_{KL} + \log(P(x))$$

y se comprueba que:

$$E_q[\log p_\theta(x|z)] = \log(P(x)) \geq ELBO$$

Por este motivo este término se le conoce como límite inferior en la evidencia “Lower Bound”. A menudo este término, en la jerga de Autoencoders Variacionales se menciona como “Stochastic Gradient Variational Bayes” (SGVB)

Ahora podemos reescribir la evidencia como:

$$\log p(x) = ELBO(\phi, \theta) + D_{KL}(q_\phi(z|x) || p_\theta(z|x))$$

Observando esta expresión y sabiendo que el KL es siempre positivo o igual a cero. Podemos ver que maximizando el ELBO respecto de ϕ y θ hacemos que la divergencia KL sea muy pequeña y consecuentemente mejora la evidencia $\log p(x)$. Minimizar la divergencia KL es por consiguiente equivalente a maximizar el ELBO respecto de ϕ y θ

Si una vez maximizado el ELBO, este se aproxima a la distribución de los datos $P(x)$ la distancia entre las dos distribuciones es cercana a cero y habremos minimizado la distancia entre $q_\phi(z|x) || p(z|x)$ es decir minimizado la divergencia KL.

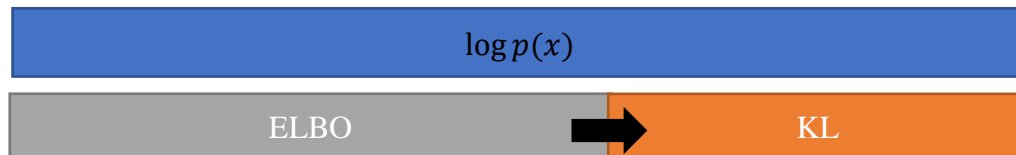


Figura 15 Maximización del ELBO

3.3.2 Optimización y Función de pérdida

La función de pérdida como ya se comentó anteriormente en la [sección 2.1.5](#) es la encargada de medir el error entre la salida obtenida y la salida deseada.

El objetivo del aprendizaje de una red neuronal es siempre minimizar la función de pérdida. Sin embargo, en nuestro caso como se ha detallado en la sección anterior, nuestro objetivo es maximizar la función de pérdida, concretamente el ELBO.

Dicho esto, nuestra función de pérdida quedaría:

$$\text{funcion de perdida} = -ELBO$$

Siendo el ELBO:

$$ELBO(\phi, \theta) = -D_{KL} + \log(P(x))$$

Nuestra función de pérdida (ELBO) está compuesta por dos términos:

- **Pérdida latente**
Se mide con la divergencia de Kullback-Leibler y representa la pérdida de información latente cuando aproximamos la distribución posterior real con la distribución variacional.
- **Pérdida generativa**
Mide con que precisión se reconstruyen las imágenes de entrada respecto de las imágenes de salida de la red. Se utiliza la entropía cruzada[19] para medir el error.

Para conseguir minimizar la función de pérdida se utilizan algoritmo de optimización que van cambiando poco a poco los parámetros de la red (Actualización de pesos y bias) en dirección donde se minimice la pérdida o error.

El descenso de gradientes es uno de los algoritmos de optimización más utilizados que, de forma simplificada, se trata de un algoritmo iterativo que va dando pequeños saltos (Dependiendo de una tasa de aprendizaje) en dirección negativa al gradiente en busca del mínimo de la función a optimizar. Una vez más, como el objetivo es maximizar el ELBO,

en lugar de ir en dirección opuesta al gradiente para buscar un mínimo, vamos en el mismo sentido que el gradiente, utilizando técnicas de **ascenso de gradiente estocástico** para buscar un máximo de la función.

Para poder aplicar métodos de optimización basadas en el gradiente, necesitamos que toda la red sea diferenciable, ya que estas técnicas se basan en el cálculo de derivadas.

Aquí nos encontramos ante uno de los problemas que presentan los Autoencoders Variacionales que para poder utilizar el ascenso de gradiente tenemos que ser capaces de calcular gradientes en todos los nodos de la red. Pero las variables latentes presentan aleatoriedad por lo que no son diferenciables. Por ejemplo, podríamos muestrear z con una distribución gaussiana cuyos parámetros son las salidas del codificador, pero esta operación de muestreo no tiene gradiente.

Para solucionar este problema se usa el “Truco de Re-parametrización”.

3.3.3 Truco de la Re-parametrización

El truco de la Re-parametrización surge porque para entrenar un Autoencoder variacional se necesita realizar retropropagación a través de toda la red para utilizar técnicas de descenso/ascenso de gradiente. Para ello necesita poder calcular derivadas en todos los nodos de la red. Esta condición se rompe cuando queremos muestrear las variables latentes z , por ejemplo, a partir de una distribución gaussiana cuyos parámetros son la salida del codificador.

Este truco es muy simple y consiste en desviar la operación no diferenciable fuera de la red, aunque siga estado presente, ya no interfiere en el entrenamiento de la red. En nuestro caso, convierte la variable aleatoria z no diferenciables en una función diferenciable de x , desacoplada de aleatoriedad.

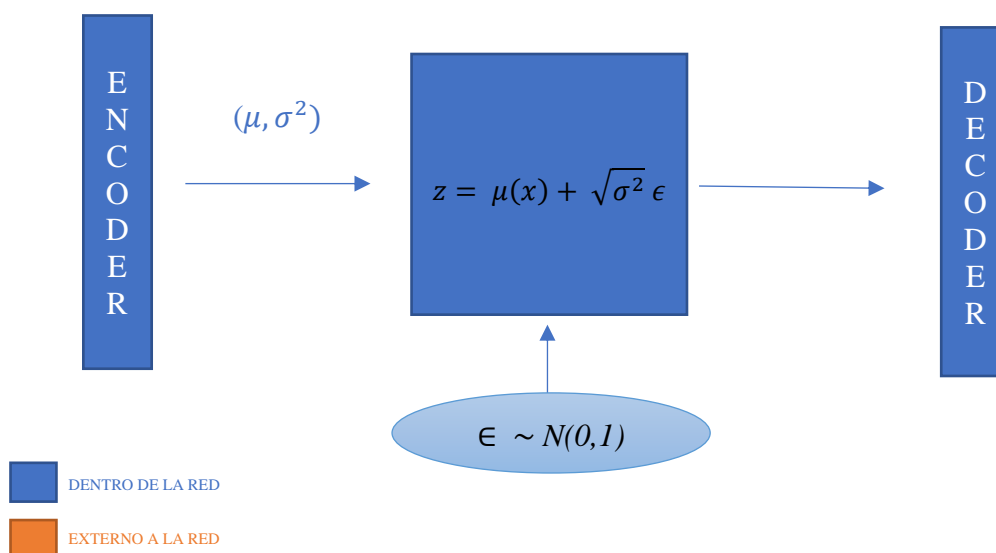


Figura 16 Truco de la Re-parametrización

Supongamos el ejemplo de utilizar una distribución gaussiana cuyos parámetros son dados por la salida del codificador, que además de ser el diseño elegido en mi propio Autoencoders variacional, es el más utilizado. Si por otro lado tomamos muestras de una gaussiana estándar

con media 0 y varianza 1, podemos convertirla en cualquier gaussiana que nos imaginemos siempre y cuando conozcamos la media y la varianza. Dicho esto, una función de muestreo de $z = g(\epsilon, x)$ siendo $x \sim \text{Normal}(\mu, \sigma^2)$ podría ser:

$$z = \mu(x) + \sqrt{\sigma^2} \epsilon \quad \text{siendo } \epsilon \sim \text{Normal}(0,1)$$

Una vez dada esta función de muestreo, a partir de aquí en lugar de muestrear z como $z \sim q_\theta(z|x)$. Z es ahora una función que toma como parámetros un ϵ y una media y una varianza (μ, σ^2) que son las salidas del codificador. Ahora para utilizar técnicas de descenso/ascenso de gradiente solo necesitamos derivadas parciales respecto de μ y σ^2 mientras que ϵ no se tiene en cuenta a la hora de calcular derivadas, ya que esta fuera de la red.

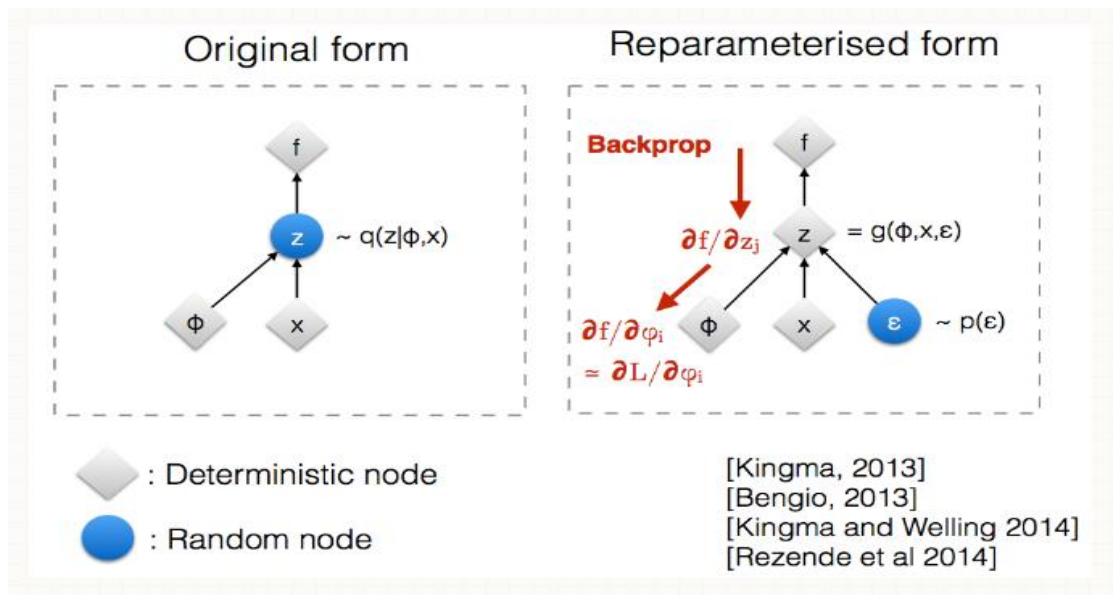


Figura 17 Función de muestreo Re-parametrizada [47]

Una vez hemos aplicado este truco de la Re-parametrización, ya estamos en condiciones de poder entrenar nuestra red. El proceso de entrenamiento de esta red funciona de la misma forma que le resto de redes neuronales. Se optimiza la función de pérdida, donde los parámetros ϕ y θ de la red, son optimizados de manera conjunta con el Autoencoders Variacional.

El parámetro ϕ corresponde con la distribución condicional $q_\theta(z|x)$ que es la aproximación a la distribución posterior real $p_\theta(z|x)$. Y el parámetro θ corresponde con el likelihood $p_\theta(x|z)$. Es otras palabras, ϕ es el parámetro de la red codificadora y θ es el parámetro de la red generativa.

4 Decisiones de Diseño de la red neuronal

Una vez tenemos una visión global de que son los Autoencoders Variacionales, como funcionan y sus fundamentos matemáticos. En este apartado se detallarán las decisiones de diseño elegidas para implementar un Autoencoder Variacional.

Las decisiones de diseño de una red definen aspectos como, cuántas capas se han usado nuestra red neuronal, qué funciones de activación se han usado, cómo se han inicializado los pesos y sesgos, qué optimizador se ha elegido...

4.1 Arquitectura de la red

Como se ha mencionado anteriormente, un Autoencoder Variacional consta de tres partes fundamentales: el codificador, el decodificador y el espacio latente. Dos de ellas, como son el codificador y el decodificador son redes neuronales en sí mismas, que están interconectas por el espacio latente.

Los Autoencoders Variacionales tienen muchas de formas diferentes de ser diseñados, atendiendo a su estructura de red. El diseño elegido por mí y con el que he obtenido mejores resultados es el siguiente:

- Red completamente conectada (Fully-connected)
- El Encoder es un perceptrón multicapa (MLP¹¹), con una capa de entrada y dos capas ocultas.
- El Decoder es un perceptrón multicapa, con dos capas ocultas y una capa de salida.

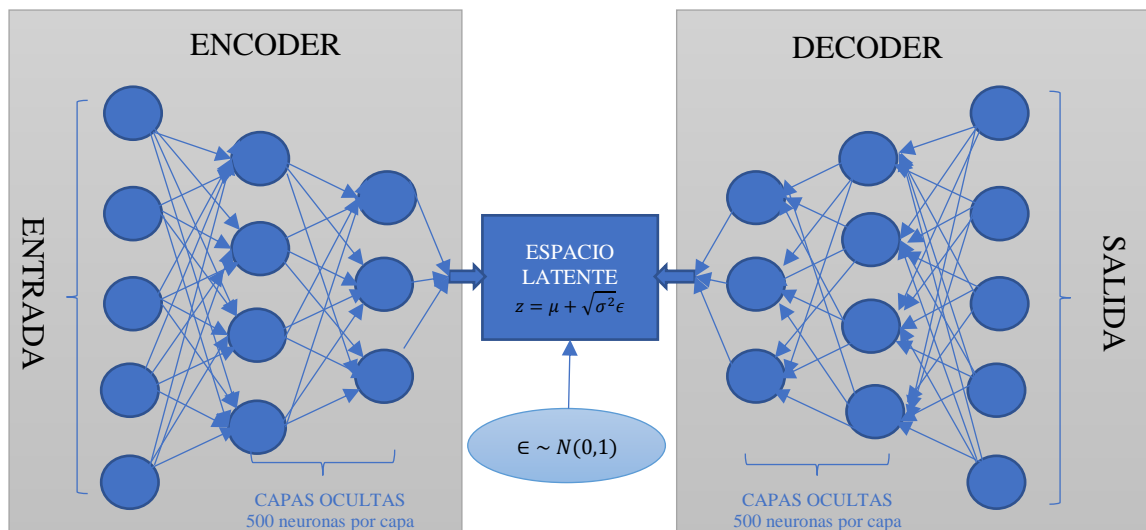


Figura 18 Arquitectura del Autoencoders implementado

¹¹ MultiLayer Perceptron [38]

4.1.1 Encoder

El Encoder como ya sabemos es una red neuronal dentro de un Autoencoder Variacional. Para ellos he utilizado una red neuronal simple, completamente conectada, llamada perceptrón multicapa, con dos capas ocultas y 500 neuronas cada una.

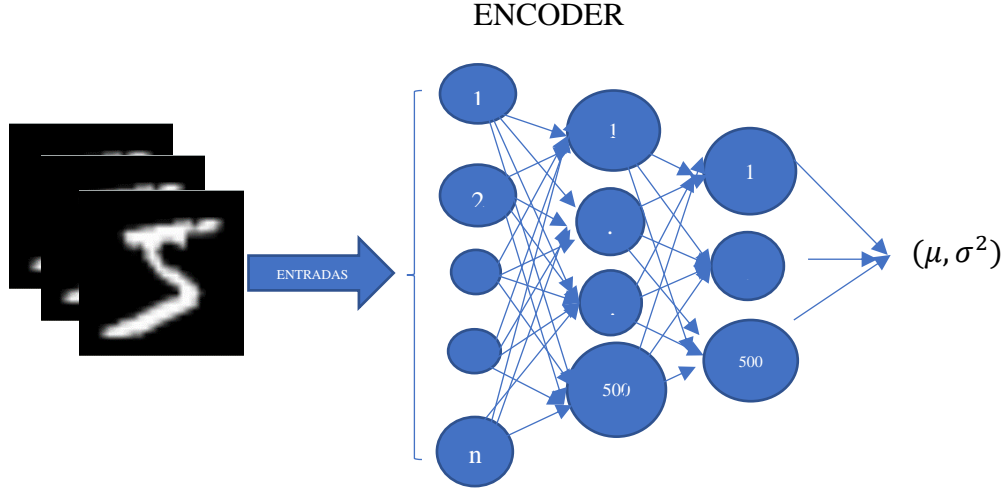


Figura 19 Diseño del Encoder (Codificador)

Como aproximación posterior variacional $q_{\phi}(z|x)$, se he utilizado una distribución gaussiana multivariante con una estructura de covarianza diagonal:

$$\log q_{\phi}(z|x) = \log N(\mu, \sigma^2)$$

Donde la media μ y la varianza σ^2 son las salidas del codificador. Tal que:

$$encoder_{out} = W_{capa2} \cdot \text{relu}(W_{capa1}Z + b_{capa1}) + b_{capa2}$$

$$\mu = W_{media} \cdot encoder_{out} + b_{media}$$

$$\sigma^2 = W_{varianza} \cdot encoder_{out} + b_{varianza}$$

Y donde $\phi = W_{capa1}, W_{capa2}, W_{media}, W_{varianza}, b_{capa1}, b_{capa2}, b_{media}, b_{varianza}$ son los pesos y las bias de la red neuronal.

4.1.2 Decoder

El Decoder tiene la misma estructura que el Encoder, es una red neuronal simple, completamente conectada, MLP, con dos capas ocultas y 500 neuronas cada una.

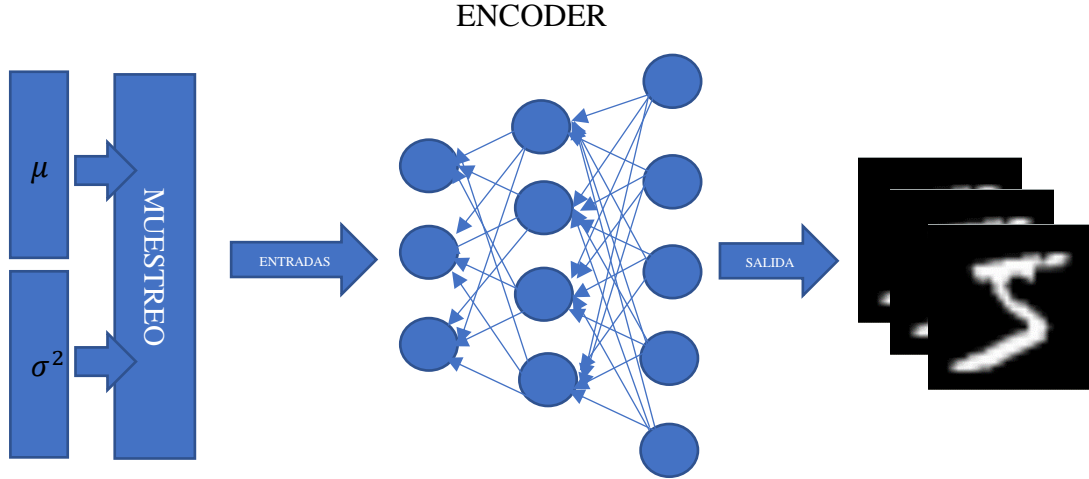


Figura 20 Diseño del Decoder (Decodificador)

En el caso de la probabilidad, $p_{\theta}(x|z)$ se ha utilizado una distribución de Bernoulli multivariante para aproximar esta probabilidad.

$$\log p_{\theta}(x|z) = \sum_{i=1}^D x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i)$$

Donde x es el valor de entrada real e y es:

$$y = \text{funcionSigmoide}(W_{out} \text{relu}(W_{capa2} \text{relu}(W_{capa1}Z + b_{capa1}) + b_{capa2}) + b_{out})$$

Y donde $\theta = W_{out}, W_{capa2}, W_{capa1}, b_{capa1}, b_{capa2}, b_{out}$ son los pesos y las bias de la red neuronal.

4.1.3 Espacio Latente

Dado que las variables latentes son en realidad una distribución de probabilidad, necesitaremos tomar muestras de este espacio latente para generar datos con el decodificador.

El espacio latente como se ha explicado en la sección 3.3.3 presenta aleatoriedad por lo que es preciso aplicar el truco de la Re-parametrización para que todos los nodos de la red sean diferenciables y por lo tanto derivables. Una vez aplicado el truco de la Re-parametrización estamos en condiciones de muestrear nuestro código latente para generar nuevos datos.

Un muestreo muy común cuando se aproxima la posterior variacional $q_\theta(z|x)$ como, $\log q_\theta(z|x) = \log N(\mu, \sigma^2)$ es el siguiente:

$$z = \mu(x) + \sqrt{e^{\sigma^2}} \epsilon \quad \text{siendo } \epsilon \sim \text{Normal}(0,1)$$

Tomando muestras de esta función de muestreo podemos generar nuevos datos pasando estas muestras como entrada al decodificador.

Una de las características del espacio latente es que reduce la dimensionalidad respecto de la entrada. Esta reducción de la dimensionalidad viene dada por el tamaño latente que se elija. Yo he realizado pruebas con tamaños latentes de dimensiones 2, 5, 10, 20 y 50.

4.2 Inicialización de pesos y bias

Uno de los aspectos importantes en una red neuronal, es la forma en la que se inicializan los pesos y las redes. El proceso de aprendizaje que tiene una red con retropropagación es su capacidad de adaptar los pesos y bias para aprender una relación entre las entradas y las salidas. Por lo tanto, realizar una buena inicialización ayudara a este proceso y a que el algoritmo de optimización converja antes.

En mi caso para inicializar los pesos y las bias me he basado en este paper [39] que plantea utilizar pesos aleatorios tomados de una distribución normal uniforme con media $\mu = 0$ y desviación $\sigma = m^{-1/2}$ siendo m el total de conexiones que alimentan el nodo.

Initializing Weights

Assuming that:

1. the training set has been normalized, and
2. the sigmoid from Figure 4b has been used

then weights should be randomly drawn from a distribution (e.g. uniform) with mean zero and standard deviation

$$\sigma_w = m^{-1/2} \tag{16}$$

where m is the fan-in (the number of connections feeding *into* the node).

Figura 21 Inicialización de pesos y bias

4.3 Funciones de activación

Las funciones de activación elegidas en toda la red se basan nuevamente en este paper [39] con el objetivo de mejorar el aprendizaje de la red. Funciones que han sido explicadas en la sección 2.1.3.

Funciones de activación elegidas

- Encoder
Para las capas ocultas de la red neuronal codificadora he utilizado la función de activación RELU. Por otro lado, se he utilizado una función de activación para las salidas de las varianzas, la exponencial.
- Decoder
Para las dos primeras capas de la red neuronal decodificador he utilizado la función de activación RELU. Mientras que para la capa de salida he utilizado una función sigmoide ya que como en este trabajo se han utilizado dataset de imágenes, las x son pixeles que valen o bien 0 o bien 1

4.4 Función de Pérdida

Como ya sabemos de la sección 3.3.2, la función de pérdida utilizada en los Autoencoders Variacionales es:

$$funcion\ de\ perdida = -ELBO$$

La razón por la que se define como el negativo del Lower Bound, es porque a diferencia de otros muchos casos en los que el objetivo del aprendizaje de la red reside en minimizar la función de pérdida, en nuestro caso, como se ha explicado detalladamente en la sección 3.3.1 nuestro objetivo es maximizar el ELBO.

Siendo el ELBO:

$$ELBO(\phi, \theta) = -D_{KL} + \log(P(x))$$

El primer termino de esta expresión es, la divergencia de Kulback-Leibler que para los casos en los que el prior de z , $p_\theta(z)$ y la aproximación posterior variacional $q_\phi(z|x)$, son gaussianas se calcula como:

$$-D_{KL}(q_\phi(z|x) || p_\theta(z|x)) = \frac{1}{2} \sum_{j=1}^J (1 + \log(\sigma_j^2) - (\mu_j^2) - (\sigma_j^2))$$

Siendo J la dimensión del espacio latente.

Puesto que yo he utilizado una distribución gaussiana con estructura de covarianza diagonal para aproximar el posterior variacional $\log q_{\phi}(z|x) = \log N(\mu, \sigma^2)$, la divergencia de Kullback-Leibler viene calculado como:

$$-D_{KL}(q_{\phi}(z|x) || p_{\theta}(z|x)) = \frac{1}{2} \sum_{j=1}^J (1 + (\sigma_j^2) - (\mu_j^2) - (e^{\sigma_j^2}))$$

El segundo término de la expresión del ELBO es, el marginal likelihood, $p_{\theta}(x|z)$. Como se ha utilizado una distribución de Bernoulli multivariante, este término es calculado como:

$$\log p_{\theta}(x|z) = \sum_{i=1}^D x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i)$$

Si juntamos estos dos términos y reescribimos el ELBO quedaría como resultado la expresión que he utilizado para calcular el Lower Bound:

$$ELBO = \frac{1}{2} \sum_{j=1}^J (1 + (\sigma_j^2) - (\mu_j^2) - (e^{\sigma_j^2})) + \sum_{i=1}^D x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i)$$

4.5 Optimizador

El objetivo de una red neuronal es minimizar la función de pérdida, en nuestro caso como ya sabemos es diferente. Para conseguir minimizar la función de pérdida se utilizan algoritmo de optimización que van cambiando poco a poco los parámetros de la red (Actualización de pesos y bias) en dirección donde se minimice la pérdida o error.

Uno de los algoritmos de optimización más utilizados son el clásico descenso de gradiente estocástico. Sin embargo, para mi implementación he utilizado un algoritmo relativamente moderno (2015), llamado Adam [41].

El algoritmo de optimización Adam es una variante del clásico descenso de gradiente estocástico que actualmente está teniendo una adopción muy elevada en problemas de lenguaje artificial y aprendizaje profundo en visión artificial. Este algoritmo se explica con más detalle en el Anexo A.

El optimizador Adam se basa en técnicas de descenso de gradiente, cuando nosotros en realidad estamos interesados en técnicas de ascenso de gradiente, es decir, en técnicas que nos permitan ir en dirección positiva al gradiente para maximizar el ELBO. Para solucionar este inconveniente se usa el negativo del ELBO como función de pérdida para ir en dirección contraria al descenso de gradiente, es decir, consiguiendo una técnica de ascenso de gradiente.

Este optimizador tiene una serie de parámetros:

- Beta 1: Tasa de reducción exponencial para los estimadores de primer orden
- Beta 2: Tasa de reducción exponencial para los estimadores de segundo orden
- Alfa: Velocidad de Aprendizaje
- ϵ : Es un número muy pequeño que evita la división por cero

Unos buenos valores para estos parámetros son según este paper [41]:

$\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ y $\epsilon = 10^{-8}$

5 Implementación

La implementación del Autoencoder Variacional se ha llevado a cabo con la librería de código abierto Tensorflow[44]. Tensorflow es una librería de aprendizaje automático desarrollada por Google. Proporciona una API muy completa para implementar redes neuronales.

La implementación que aquí se expone es utilizando el Dataset MNIST¹² pero cabe destacar que también se ha realizado otra implementación para el Dataset Frey Face¹³. Todas las implementaciones junto con las pruebas realizadas están disponibles en mi repositorio personal¹⁴.

Además, en el Anexo E se explica el dataset MNIST para dar una pequeña introducción a la representación de los datos y su manipulación.

5.1 Código

El código que aquí se muestra se puede ver en este notebook ipython¹⁵, donde además se puede ejecutar. Siempre y cuando se tengan las herramientas necesarias instaladas. En el Anexo B hay un manual de instalación para poder ejecutar los notebooks de todas las implementaciones.

5.1.1 Importación de módulos y dataset

Tensorflow nos proporciona una serie de datasets de ejemplo para utilizarlo con su API, sin necesidad de un tratamiento previo, lo cual ahorra mucho tiempo. Entre estos dataset de ejemplos de esta MNIST. Importar los datos es tan sencillo como se observa en este fragmento de código. Además, en este fragmento se incluyen las librerías necesarias para la implementación y una sentencias para evitar warnings molestos.

```
import os os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import time
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
```

¹² MNIST: es un dataset de imágenes de dimensión 28x28 (784 Píxeles), de dígitos escritos a mano comprendidos entre el 0 y el 9.

¹³ Frey Face: es un dataset de imágenes de caras de dimensiones 28x20 (560 Píxeles), de rostros humanos con diferentes expresiones y posiciones. Triste, Sonriendo, de lado...

¹⁴ Repositorio personal:
<https://github.com/FernandoArribas/MachineLearning/tree/master/Modelos%20Generativos%20Profundos>

¹⁵ Notebook iPython VAE MNIST:
<https://github.com/FernandoArribas/MachineLearning/blob/master/Modelos%20Generativos%20Profundos/TFG-Visualizacion-MNIST.ipynb>

5.2 Parámetros de configuración del modelo

En este fragmento de código se crean variables para configurar los parámetros del modelo. Entre ellas se crean variables como la tasa de aprendizaje, la cual es muy importante en el proceso de optimización. Así como variables que definen el número de neuronas de las capas ocultas y la dimensión del espacio latente.

```
#Variables de Train
num_epocas = 50
batch_size = 100
Learning_rate = 1e-3
n_datos = mnist.train.num_examples
num_batch = int(n_datos / batch_size)
#Variables del dataset
tam_imagen = 784 # Imagenes de 28x28 pixels
tam_latente = 2
tam_hidden_layer = 500
```

5.3 Promesas del modelo: Placeholder

Se define el Placeholder¹⁶ que se utilizara como entrada de la red. Por este motivo se crea con shape¹⁷ de dimensiones [None, tam_imagen], donde la primera dimensión hace referencia al índice de las imágenes de entrada que puede ser cualquier número y la segunda dimensión es del tamaño de la imagen, es decir, en el caso del MNIST 784 Pixeles (28x28).

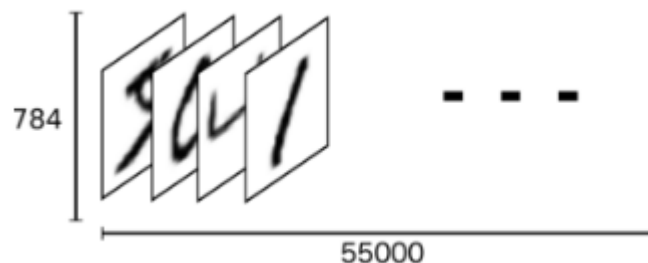


Figura 22 Shape placeholder de entrada de datos

```
x = tf.placeholder(tf.float32, shape=[None, tam_imagen]) #Entrada de
datos, imagenes
```

¹⁶ Placeholder: es una promesa de dar un valor más adelante.

¹⁷ Shape: es la forma que tiene el tensor, extrapolándolo a las matemáticas equivale a las dimensiones de una matriz.

5.4 Encoder

En este fragmento de código se implementa el codificador del Autoencoder Variacional. En primer lugar, se definen e inicializan los pesos y las bias de la red acorde a la técnica elegida y explicada en la sección 4.2 utilizando variables de Tensorflow.

Una vez definidos los pesos y las bias de la red se definen nuestros modelos de regresión para cada neurona de ambas capas del Encoder.

$$modelo = tf.matmul(x, peso) + bias$$

Además, de la función de activación para cada capa.

```
#Semilla aleatoria
tf.set_random_seed(0)
#Pesos y biases
#Primera capa oculta
W_encoder1 = tf.Variable(tf.random_normal([tam_imagen, tam_hidden_layer],
stddev= tf.pow(float(tam_imagen), -0.5)))
b_encoder1 = tf.Variable(tf.random_normal([tam_hidden_layer], stddev=
tf.pow(float(tam_hidden_layer), -0.5)))

#Segunda capa: salida encoder
W_encoder_out = tf.Variable(tf.random_normal([tam_hidden_layer, tam_hidden_layer],
stddev= tf.pow(float(tam_hidden_layer), -0.5)))
b_encoder_out = tf.Variable(tf.random_normal([tam_hidden_layer], stddev=
tf.pow(float(tam_hidden_layer), -0.5)))

W_z_var = tf.Variable(tf.random_normal([tam_hidden_layer, tam_latente],
stddev=tf.pow(float(tam_hidden_layer), -0.5)))
b_z_var = tf.Variable(tf.random_normal([tam_latente], stddev=tf.pow(float(tam_latente), -0.5)))

W_z_mean = tf.Variable(tf.random_normal([tam_hidden_layer, tam_latente],
stddev=tf.pow(float(tam_hidden_layer), -0.5)))
b_z_mean = tf.Variable(tf.random_normal([tam_latente], stddev=tf.pow(float(tam_latente), -0.5)))

#Model del Encoder
encoder_capa1 = tf.matmul(x, W_encoder1) + b_encoder1
encoder_capa1 = tf.nn.relu(encoder_capa1)
encoder_out = tf.matmul(encoder_capa1, W_encoder_out) + b_encoder_out
encoder_out = tf.nn.relu(encoder_out)
```

5.5 Espacio Latente

Aquí se computan las medias y las varianzas que son la salida del codificador. Y se define la función de muestreo del espacio latente, como z . Esta función ha sido explicada en la sección que habla del truco de la Re-parametrización 3.3.3 .

```
#Mean
z_mean = tf.matmul(encoder_out, W_z_mean) + b_z_mean
#Vaarianza
z_var = tf.matmul(encoder_out, W_z_var) + b_z_var

epsilon = tf.random_normal(tf.shape(z_mean), mean=0.0, stddev=1.0, dtype=
=tf.float32)
z = z_mean + (tf.multiply(tf.sqrt(tf.exp(z_var)), epsilon))
```

5.6 Decoder

En este fragmento de código, se define la red neuronal del decodificador. Es muy similar a la definición del Encoder, salvo que en este caso se dispone de una última capa de salida en el que se aplica la función sigmoide, `tf.nn.sigmoid(decoder_out)`, ya que como estamos trabajando con pixeles de una imagen estos pueden ser o bien 0 o bien 1.

```
#Pesos y biases
#Primera capa oculta
W_decoder1 = tf.Variable(tf.random_normal([tam_latente, tam_hidden_la
yer], stddev=tf.pow(float(tam_latente), -0.5)))
b_decoder1 = tf.Variable(tf.random_normal([tam_hidden_layer], stddev=
tf.pow(float(tam_hidden_layer), -0.5)))

#Segunda capa oculta
W_decoder2 = tf.Variable(tf.random_normal([tam_hidden_layer, tam_hidd
en_layer], stddev= tf.pow(float(tam_hidden_layer), -0.5)))
b_decoder2 = tf.Variable(tf.random_normal([tam_hidden_layer], stddev=
tf.pow(float(tam_hidden_layer), -0.5)))

W_decoder_out = tf.Variable(tf.random_normal([tam_hidden_layer, tam_i
magen], stddev=tf.pow(float(tam_hidden_layer), -0.5)))
b_decoder_out = tf.Variable(tf.random_normal([tam_imagen], stddev= tf
.pow(float(tam_imagen), -0.5)))

#Model del Decoder
decoder_capa1 = tf.matmul(z, W_decoder1) + b_decoder1
decoder_capa1 = tf.nn.relu(decoder_capa1)
decoder_capa2 = tf.matmul(decoder_capa1, W_decoder2) + b_decoder2
decoder_capa2 = tf.nn.relu(decoder_capa2)
decoder_out = tf.matmul(decoder_capa2, W_decoder_out) + b_decoder_out
decoder_out = tf.nn.sigmoid(decoder_out)
```

5.7 Función de pérdida

La función de pérdida es definida tal y como se ha mostrado en la sección 4.4 y su implementación no tiene mayor complejidad.

```
#Para evitar que el cálculo de gradientes de error cuando los logaritmos son log(x) con x~0 se suma un sesgo 1e-9
decoder_out = tf.clip_by_value(decoder_out, 1e-9, 1 - 1e-9)
likelihood = tf.reduce_sum(x * tf.log(decoder_out) + (1 - x) * tf.log(1 - decoder_out), 1)
#Divergencia KL: -D_KL(q(z)||p(z))
KL = (1/2) * tf.reduce_sum(1 + z_var - tf.square(z_mean) - tf.exp(z_var), 1)

#El likelihood se escala al tam de los batches
ELBO = tf.reduce_mean(KL + likelihood)

#FUNCIÓN DE PÉRDIDA: Donde maximizamos el ELBO
#Como nuestro objetivo es maximizar el ELBO vamos en sentido positivo al gradiente
function_pérdida = -ELBO
```

5.8 Optimizador

Gracias a la API que nos proporciona la librería de Tensorflow implementar el optimizador es trivial y tan solo hay que decidir que parámetros del optimizador son los más idóneos para nuestro modelo. Los parámetros utilizados son los definidos en la sección 4.5.

```
optimizer = tf.train.AdamOptimizer(learning_rate = Learning_rate,
beta1 = 0.9, beta2 = 0.999, epsilon = 1e-08)
train_step = optimizer.minimize(function_pérdida)
```

5.9 Entrenamiento

El entrenamiento con Tensorflow consiste en llamar al optimizador usando una sesión de Tensorflow¹⁸ que ejecuta todas las sentencias que se han añadido al grafo computacional, es decir, todas las sentencias que se han definido en los fragmentos de códigos anterior.

```
print('Entrenamiento')
session = tf.InteractiveSession()
tf.global_variables_initializer().run()
valores_coste = []
for epoca in range(1, num_epocas+1):
    average_coste = 0
    inicio = time.time()
    for i in range(num_batch):
        batch, _ = mnist.train.next_batch(batch_size)
```

¹⁸ Session Tensorflow: Es la forma que tiene Tensorflow de ejecutar el código que se va añadiendo al grafo computacional.

```
_, coste = session.run([train_step, function_coste], feed_dict = {x: batch} )
    average_coste = (average_coste + coste)
    valores_coste.append(coste)
    fin = time.time()
    tiempo = fin - inicio
    print('Epoca: %d Tiempo: %f Loss= %f' % (epoca, tiempo, average_coste/num_batch))
```

6 Experimentos y resultados

Una vez implementado el Autoencoders Variacional según el diseño explicado en la sección 4, se han realizado numerosos experimentos obteniendo resultados muy satisfactorios desde diferentes perspectivas. Todos estos experimentos han sido aplicados sobre la implementación dada en el apartado anterior, que ha sido elegida después de realizar varios diseños probando con diferentes estructuras y diseño de la red distintos.

Los experimentos realizados han sido orientados principalmente a la modificación de los parámetros del modelo, donde se han ido probando con diferentes valores y comparando los resultados. Todos estos experimentos van acompañados de visualizaciones muy representativas que ayudan mucho para entender y analizar los resultados obtenidos.

Un último experimento muy significativo que se ha realizado ha sido utilizar el codificador del Autoencoder variacional para extraer características de alto nivel de las imágenes y usar estas características para tareas de clasificación.

Concretamente se ha utilizado el codificador para aprender una representación de alto nivel de las imágenes de entrada (Variables latentes), estas variables latentes son utilizadas como entrada en un clasificador lineal con el objetivo de comparar los resultados de este clasificador cuando lo entrenamos con estas variables latentes y cuando lo entrenamos con las imágenes originales.

Cabe destacar que los experimentos se han realizado con mi más que humilde ordenador personal que el siguiente Hardware:

- Intel i7-2630QM CPU 2.00 GHz
- 4 Gb de memoria RAM
- Sistema operativo con 64 Bits

6.1 Entrenamiento del modelo

Se ha entrenado la red con diferentes dimensiones del espacio latente, tanto para el dataset MNIST como para el Frey Face.

6.1.1 Experimentos con MNIST

Estos son los resultados de la función de pérdida para los diferentes experimentos.

Experimento 1

- Numero de épocas: 50
- Batch Size: 100
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 2
- Neuronas de las capas ocultas: 500
- Dataset: MNIST

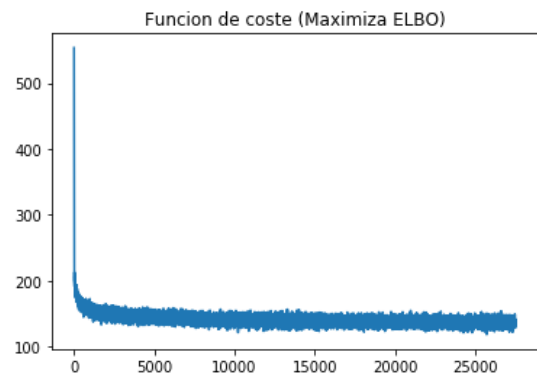


Figura 23 Función de perdida Experimento 1

Entrenamiento

Época: 0 Tiempo: 20.329743 Pérdida= 580.56123
Época: 1 Tiempo: 21.549743 Pérdida= 177.767873
Época: 10 Tiempo: 22.116709 Pérdida= 143.696704
Época: 20 Tiempo: 20.147452 Pérdida= 139.802531
Época: 40 Tiempo: 18.044507 Pérdida= 136.943999
Época: 50 Tiempo: 16.660365 Pérdida= 135.980511

Se puede observar como en la primera época la función de pérdida es muy alta y rápidamente se va minimizando en cada interacción, hasta que se va estabilizando donde ya prácticamente no mejora a partir de **135.98**.

Otra forma de ver el proceso de aprendizaje es verlo de forma visual, podemos ir mostrando como la red va reconstruyendo una imagen aleatoria de entrada a lo largo del aprendizaje. Podemos observar como al principio existe algo de ruido en la imagen reconstruida pero rápidamente se consigue reconstruir el dígito de entrada con éxito.



Figura 24 Proceso de aprendizaje: Reconstrucción de imagen aleatoria (Experimento 1)

Experimento 2

- Numero de épocas: 50
- Batch Size: 100
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 10
- Neuronas de las capas ocultas: 500
- Dataset: MNIST

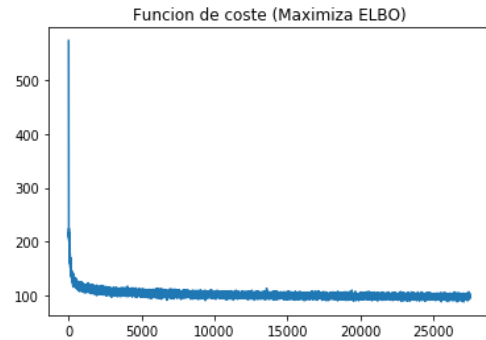


Figura 25 Función de pérdida experimento 2

Entrenamiento

Época: 0 Tiempo: 14.439644 Pérdida= 570.4643

Época: 1 Tiempo: 14.806759 Pérdida = 147.226382

Época: 10 Tiempo: 15.259352 Pérdida = 104.356407

Época: 20 Tiempo: 16.421852 Pérdida = 100.918969

Época: 40 Tiempo: 17.473157 Pérdida = 98.265996

Época: 50 Tiempo: 17.868237 Pérdida = 97.528469

Con estos valores de los parámetros podemos observar que no solo se consigue disminuir la función de pérdida bastante respecto al experimento 1 sino que también se logra conseguirlo más rápidamente.

Además, con estos valores de los parámetros del modelo podemos ver que los resultados también son mejores en tareas de reconstrucción, donde apenas se puede ver ruido a partir de las primeras épocas.



Figura 26 Proceso de aprendizaje: Reconstrucción de imagen aleatoria (Experimento 2)

Experimento 3

- Numero de épocas: 50
- Batch Size: 100
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 50
- Neuronas de las capas ocultas: 500
- Dataset: MNIST

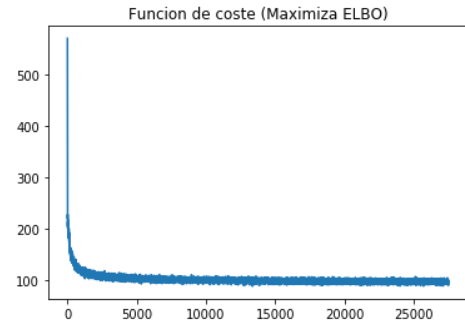


Figura 27 Función de pérdida experimento 3

Entrenamiento

Época: 0 Tiempo: 15.169185 Pérdida = 540.842798

Época: 1 Tiempo: 15.308746 Pérdida = 161.469195

Época: 10 Tiempo: 18.058111 Pérdida = 103.025157

Época: 20 Tiempo: 16.702869 Pérdida = 99.912300

Época: 40 Tiempo: 16.864400 Pérdida = 97.802639

Época: 50 Tiempo: 16.530946 Pérdida = 97.214724

Con estos valores de los parámetros podemos observar que se obtienen resultados muy similares a los obtenidos por el experimento 2, por lo que podemos decir que el valor de la función de pérdida igual 97.214 debe estar cerca del mínimo al que podemos llegar.

Siempre debemos tener en cuenta que si aumentáramos el número de épocas podríamos disminuir un poco más este valor, pero ya no sería de una manera relevante.

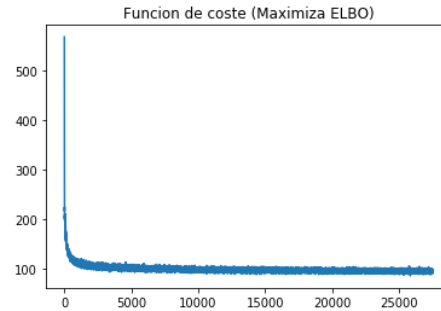
Del mismo modo que con el experimento 2, en tareas de reconstrucción se comporta de forma muy similar.



Figura 28 Proceso de aprendizaje: Reconstrucción de imagen aleatoria (Experimento 3)

Experimento 4

- Numero de épocas: 50
- Batch Size: 100
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 20
- Neuronas de las capas ocultas: 500
- Dataset: MNIST



Entrenamiento

Época: 1 Tiempo: 24.638738 Pérdida = 148.805110

Época: 10 Tiempo: 26.233851 Pérdida = 101.593139

Época: 20 Tiempo: 27.217925 Pérdida = 98.604531

Época: 40 Tiempo: 26.405099 Pérdida = 96.426431

Época: 50 Tiempo: 26.767085 Pérdida = 95.786946

Podemos observar que utilizando 20 dimensiones en el espacio latente obtenemos el valor de la función de pérdida más bajo de todos los experimentos realizados. Dado que el valor utilizado como dimensión del espacio latente en este experimento está comprendido entre los valores de los experimentos 2 (dimensión latente = 10) y 3 (dimensión latente = 50), podemos afirmar que usando 20 dimensiones latentes es como mejor se comporta nuestro modelo.

6.1.2 Experimentos con Frey Face

Experimento 1

- Numero de épocas: 50
- Batch Size: 1000 (10 Épocas del MNIST)
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 2
- Neuronas de las capas ocultas: 500
- Dataset: Frey Face

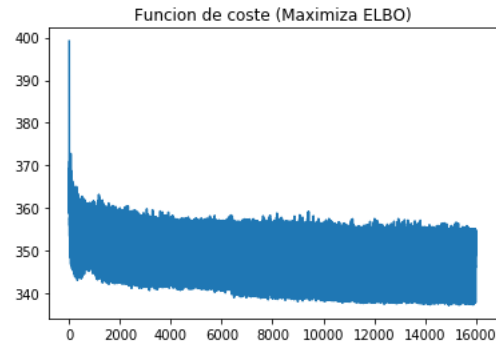


Figura 29 Función de pérdida experimento 1 (Frey Face)

Entrenamiento

Época: 100 Tiempo: 0.769022 Pérdida = 350.949242

Época: 200 Tiempo: 0.812540 Pérdida = 350.168257

Época: 300 Tiempo: 0.820902 Pérdida = 350.033113

Época: 400 Tiempo: 0.814549 Pérdida = 350.139278

Época: 500 Tiempo: 0.725487 Pérdida = 349.059999

Época: 600 Tiempo: 0.739344 Pérdida = 348.961367

Época: 700 Tiempo: 0.796922 Pérdida = 348.312843

Época: 800 Tiempo: 0.718788 Pérdida = 348.081236

Época: 900 Tiempo: 0.735490 Pérdida = 347.903923

Época: 1000 Tiempo: 0.750055 Pérdida = 347.798731

Del mismo modo que con el dataset MNIST se puede observar como la red va aprendiendo y se comportan cada vez mejor en tareas de reconstrucción con el paso de las épocas.



Figura 30 Proceso de aprendizaje: Reconstrucción de imagen aleatoria (Expe 1 Frey Face)

Experimento 2

- Numero de épocas: 50
- Batch Size: 5000 (50 Épocas del MNIST)
- Tasa de aprendizaje: $1e^{-3}$
- Dimensión latente: 2
- Neuronas de las capas ocultas: 500
- Dataset: Frey Face

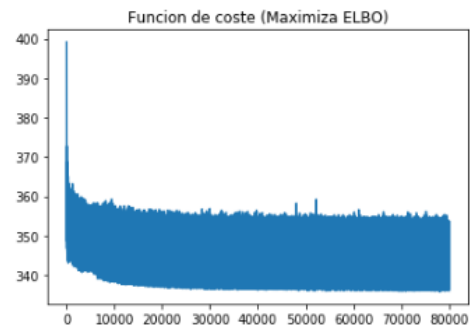


Figura 31 Función de pérdida experimento 2 (Frey Face)

Entrenamiento

Época: 100 Tiempo: 0.377165 Pérdida = 350.949242

Época: 1000 Tiempo: 0.379253 Pérdida = 347.798731

Época: 2000 Tiempo: 0.397266 Pérdida = 347.669384

Época: 3000 Tiempo: 0.473322 Pérdida = 347.504955

Época: 4000 Tiempo: 0.504336 Pérdida = 347.014309

Época: 5000 Tiempo: 0.531353 Pérdida = 346.750311

Vemos que a pesar de aumentar considerablemente el número de épocas no mejoramos prácticamente nada respecto a la función de pérdida.



Figura 32 Proceso de aprendizaje: Reconstrucción de imagen aleatoria (Expe 1 Frey Face)

6.2 Clasificador Lineal: Regresión logística multinomial

La idea es utilizar el Autoencoder Variacional para obtener características de alto nivel y ocultas de las imágenes (Variables latentes) y utilizar estas variables como entrada en un clasificador lineal en este caso Softmax (regresión logística multinomial). Es decir, se trataría de un aprendizaje semi-supervisado para clasificar. El objetivo de este experimento es comparar los resultados de este clasificador cuando lo entrenamos variables latentes y cuando lo entrenamos con las imágenes originales.

Si las variables latentes aprendidas por el codificador son ricas en información deberíamos obtener mejores resultados en la clasificación con ellas que con las imágenes originales. En otras palabras, si nuestro codificador extrae características de alto nivel de las imágenes, que representan y explican mejor los datos de entrada que las imágenes originales, en tareas de clasificación deberían comportarse de mejor manera.

Para realizar este experimento se ha implementado un clasificador lineal con Tensorflow, concretamente se ha usado Softmax, que se trata de un clasificador de regresión logística multinomial. Se han implementado dos variantes, en la primera de ellas se ha implementado este clasificador acoplándole a la entrada de este, la salida de las medias de un Autoencoder Variacional. Por otro lado, se ha implementado la misma versión del clasificador anterior, pero entrenándolo con las imágenes originales.

6.2.1 Clasificador Semi-supervisado con Autoencoder Variacional

Esta red está formada por un Autoencoder Variacional, concretamente la parte del Encoder, donde su salida (Las medias de las variables latentes) sirve como entrada del clasificador de regresión logística.

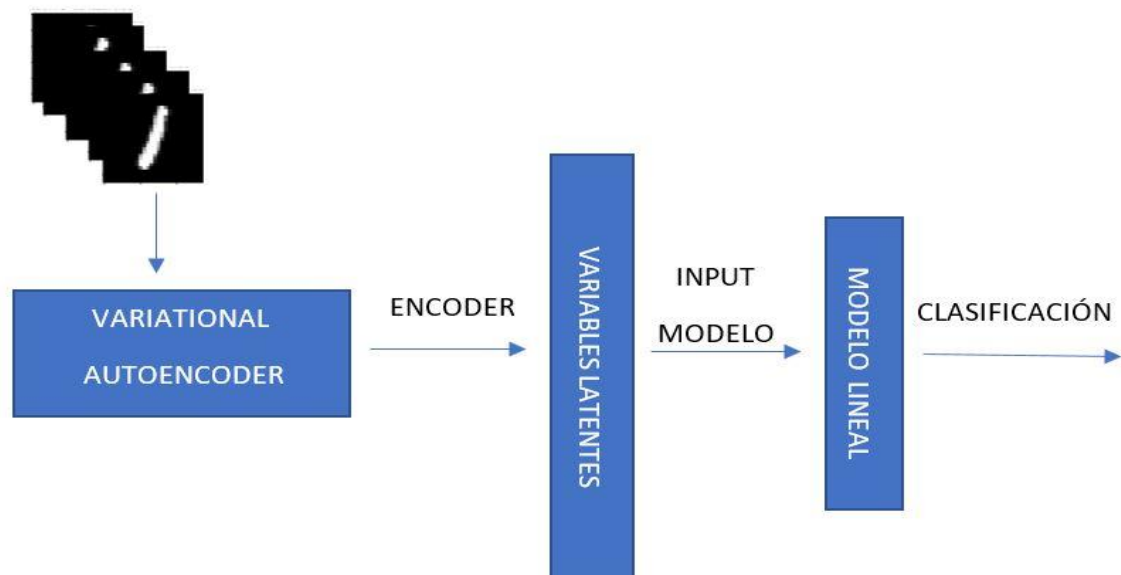


Figura 33 Arquitectura de la red del clasificador lineal semi-supervisado con VAE

El Código de este clasificador se puede ver en el Anexo 0.

6.2.2 Clasificador supervisado con imágenes originales

La arquitectura de la red en este caso es un modelo muy sencillo y su código se puede ver en el Anexo D.

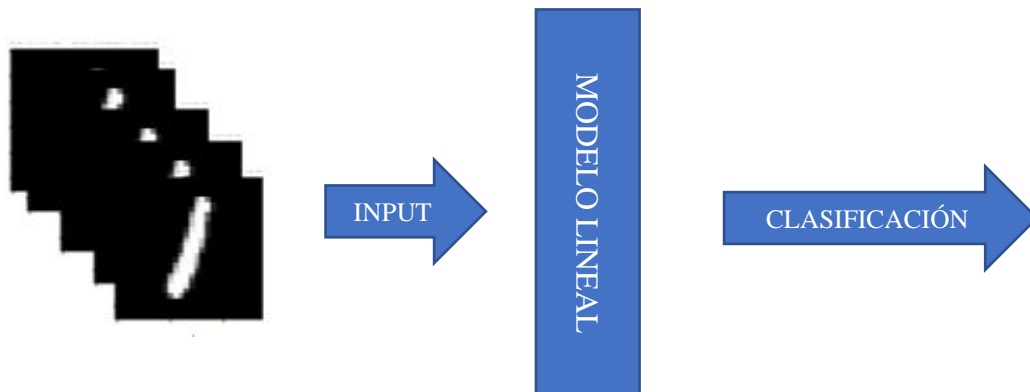


Figura 34 Arquitectura de la red clasificador lineal supervisado

6.2.3 Resultados

Una vez entrenados ambos clasificadores, manifiestan los resultados esperados. El clasificador entrenado con las medias correspondientes a la salida del Encoder de un Autoencoder Variacional, revelan mejores resultados.

Los resultados en la clasificación del dataset MNIST no arrojan lugar a dudas, aquí se muestra el porcentaje de acierto de ambos clasificadores:

- **Clasificador lineal Semi-Supervisado con Autoencoder 98%**
- **Clasificador lineal supervisado con imágenes originales 92%**

Para ver los resultados de ambos clasificadores de una forma más visual se ha desarrollado un widget con matplotlib¹⁹. El widget nos permite comparar la certeza de las clasificaciones realizadas por ambos clasificadores, dado un numero de entrada elegido mediante un slider interactiva.

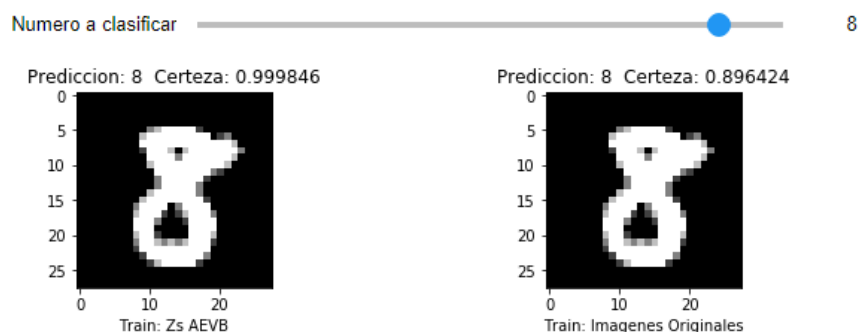


Figura 35 Widget interactivo para comparar clasificador lineal normal y clasificador con VAE

¹⁹ Matplotlib: es una librería para la generación de gráficos para el lenguaje de programación Python.

7 Visualizaciones

Una forma de entender mejor los datos y los resultados obtenidos con el Autoencoder Variacional, es mostrar los datos de forma visual.

La representación de los datos de forma visual nos aportará una visión desde otra perspectiva de como son los datos y como se distribuyen e incluso nos permitirá interactuar con ellos.

7.1 Jugando con la dimensionalidad

Una de las visualizaciones que se han realizado ha sido, mostrar una serie de muestras aleatorias de la red generativa modificando la dimensionalidad del espacio latente.

Se puede ver como dependiendo de la dimensión del espacio latente se tienen mejores resultados en tareas generativas.

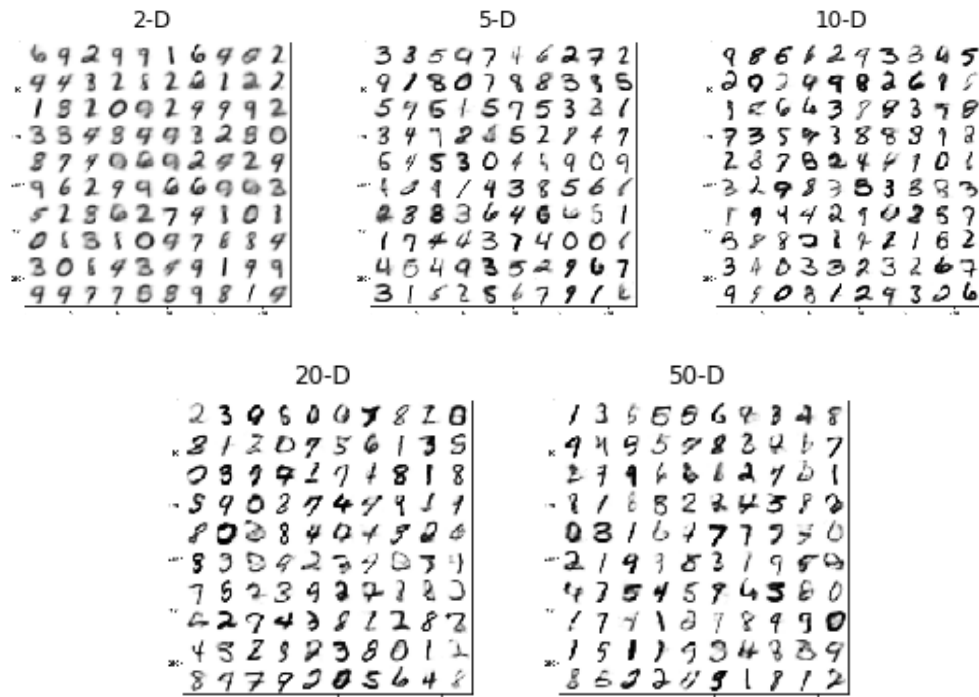


Figura 36 Muestras aleatorias del modelo generativo de MNIST con diferentes dimensiones del espacio Latente

7.2 Dispersión del espacio latente

Una de las visualizaciones más representativas que podemos obtener una vez entrenada la red, es como se distribuyen las variables latentes en el espacio. Esta visualización nos permite ver la estructura aprendida del espacio latente y las diferencias estructurales entre los datos.

7.2.1 Espacio Latente 2-D mostrado en dos dimensiones MNIST

Una vez se ha entrenado la red con un espacio latente de dos dimensiones. Se puede pintar este espacio, pintando la distribución de las medias que se obtienen como salida del codificador. Es fácil observar como la red neuronal ha aprendido una representación de alto nivel de los datos, suficientemente buena como para poder distribuir los diferentes dígitos del dataset MNIST, de manera que los números que son iguales permanecen cercanos en el espacio. Cada clúster de colores representa un dígito.

Los clústeres que están cercanos en el espacio significan que son dígitos que a pesar de que su valor sea distinto, estructuralmente son similares.

Esto que implica que números distintos aparezcan cerca y en ocasiones superpuestos en el espacio. Para aclarar esta idea, imaginemos como se escribe a mano el número 8 y el número 3, aunque sean números diferentes, en cuanto a su forma de trazado son bastantes similares de ahí que aparezcan cercanos en el espacio. Lo mismo ocurre con el 4 y el nueve.

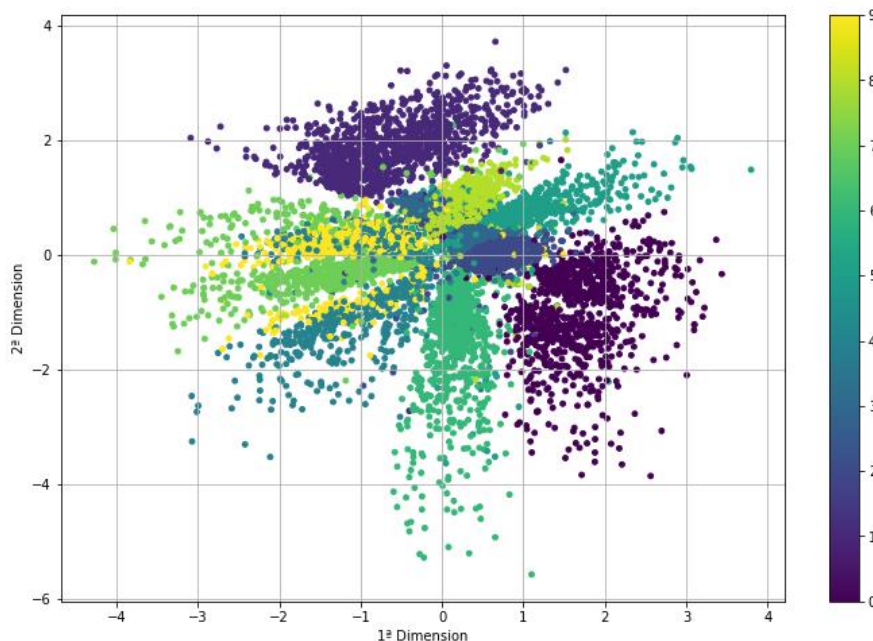


Figura 37 Espacio latente 2-D mostrado en dos dimensiones

7.2.2 Espacio Latente 2-D mostrado en tres dimensiones MNIST

Al igual que la visualización anterior también podemos ver la distribución del espacio latente con dos dimensiones, pero esta vez mostrado en tres dimensiones. Desde esta perspectiva se puede ver muy bien como existen clúster de números que se superponen unos con otros debido a sus similitudes estructurales.

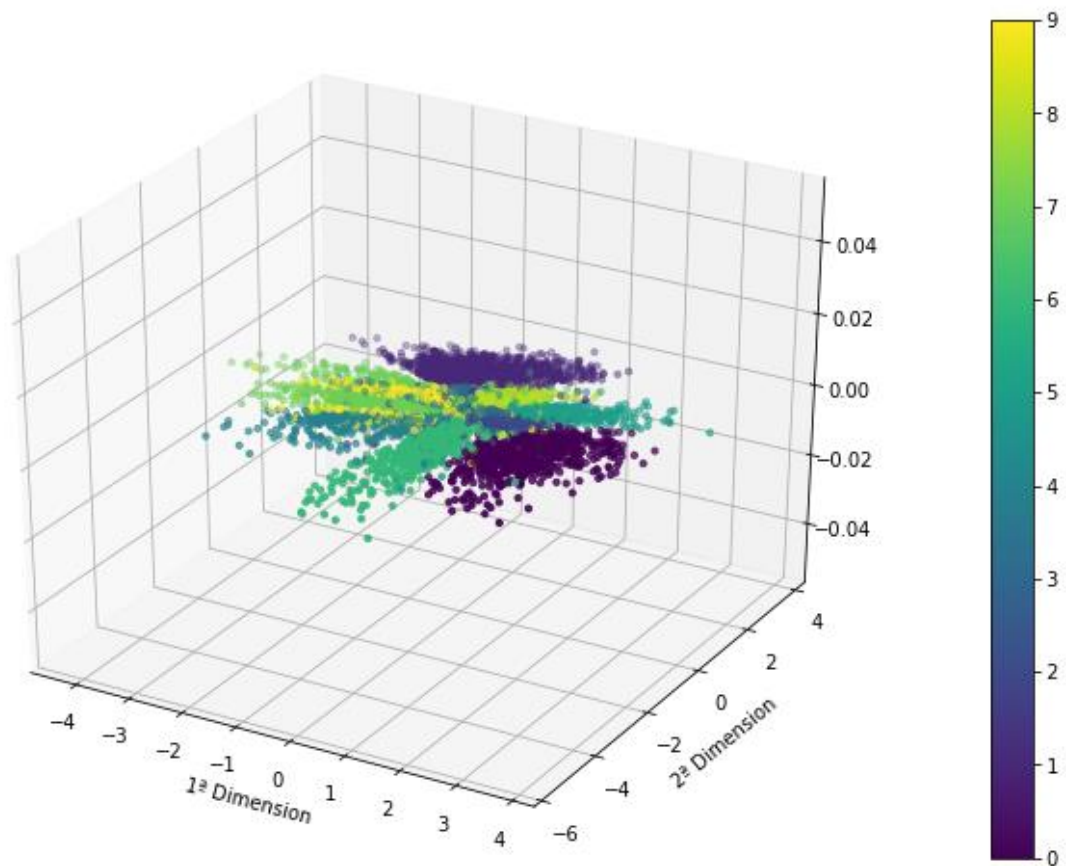


Figura 38 Espacio latente 2-D mostrado en tres dimensiones

7.2.3 Dispersión del espacio latente Frey Face

En el caso del Dataset Frey Face la distribución del espacio latente es menos relevante debido a que se disponen de menos cantidad de datos. Por lo que el espacio latente se muestra algo más irregular que en el caso del dataset MNIST.

En el Frey Face los clústeres vienen definidos por la posición y expresión de los rostros de las imágenes. Es decir, rostros tristes estarán cercanos entre sí, mientras que rostros alegres estarán cercanos entre sí, pero alejados de estos primeros.

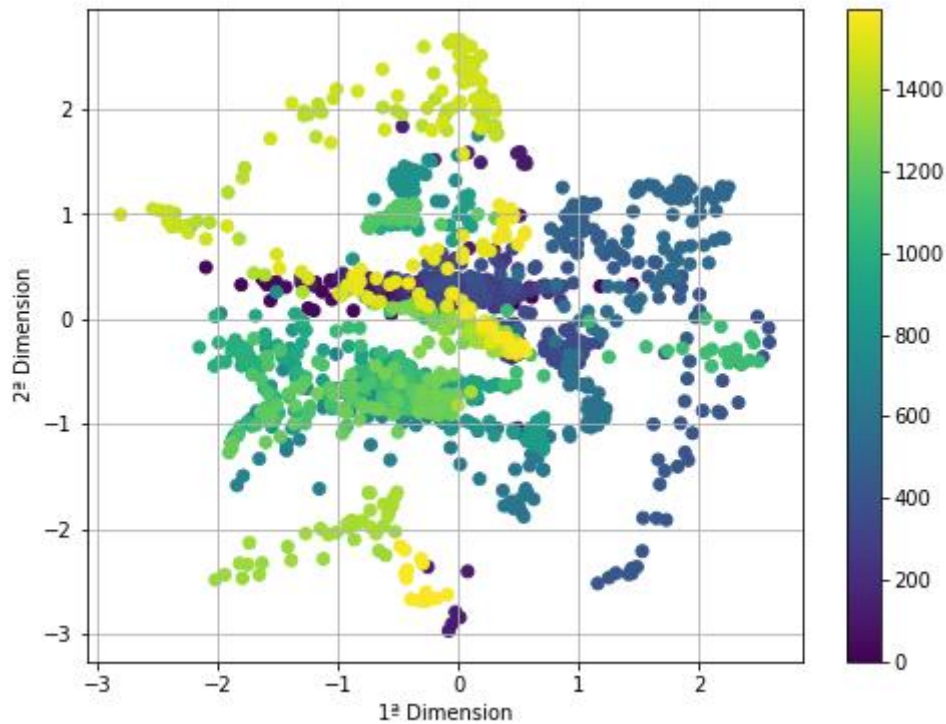


Figura 39 Espacio latente de 2-D del dataset Frey Face mostrado en dos dimensiones

7.3 Manifold

Esta representación de los datos consiste en la visualización de múltiples datos aprendidos por modelos generativos con dos dimensiones del espacio latente, en este caso con el Autoencoder Variacional.

7.3.1 Manifold MNIST

Se puede observar que los dígitos con similitudes estructurales similares a la hora de trazar su contorno están cerca en la imagen.

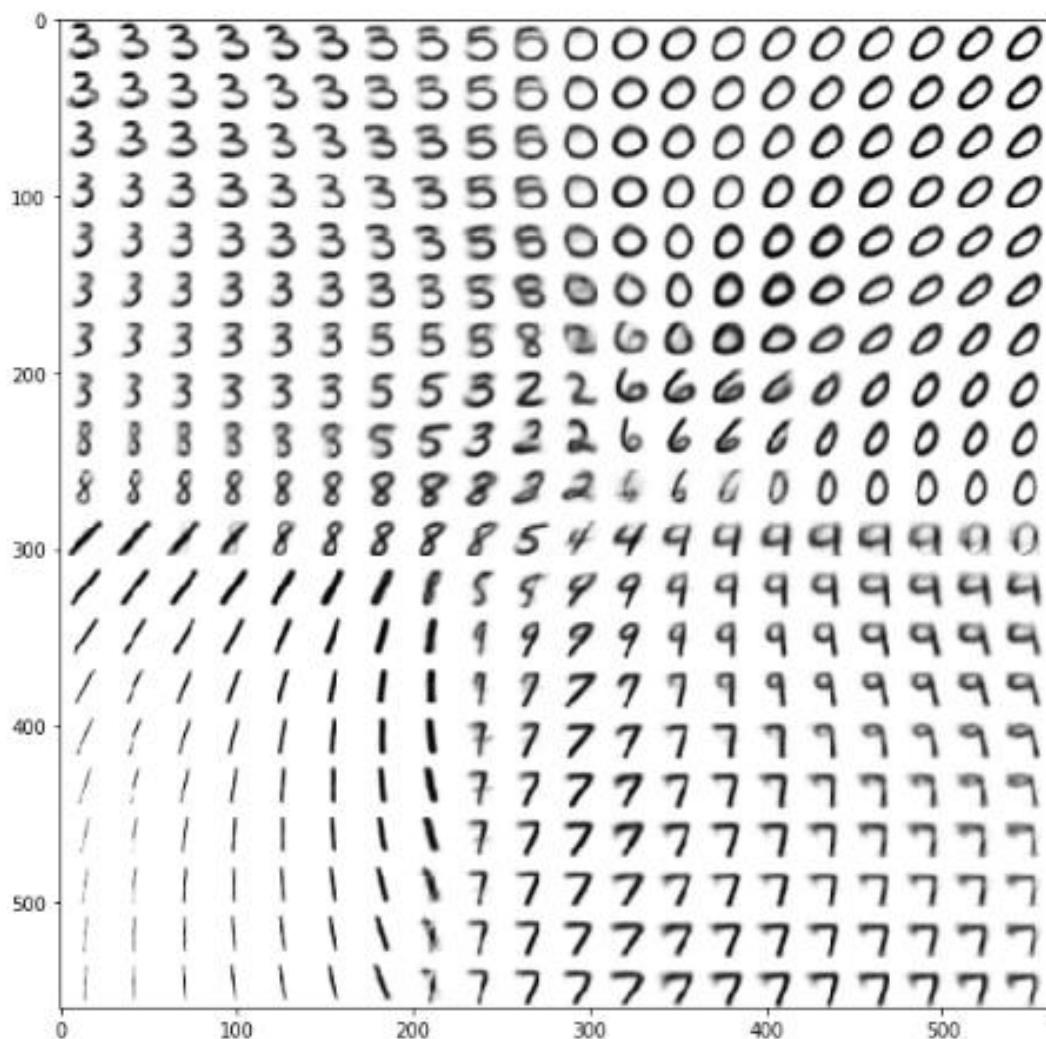


Figura 40 Manifold MNIST

7.3.2 Manifold Frey Face

En el caso del dataset Frey Face se puede observar de forma evidente como los rostros con expresiones similares aparecen juntos en el espacio latente.

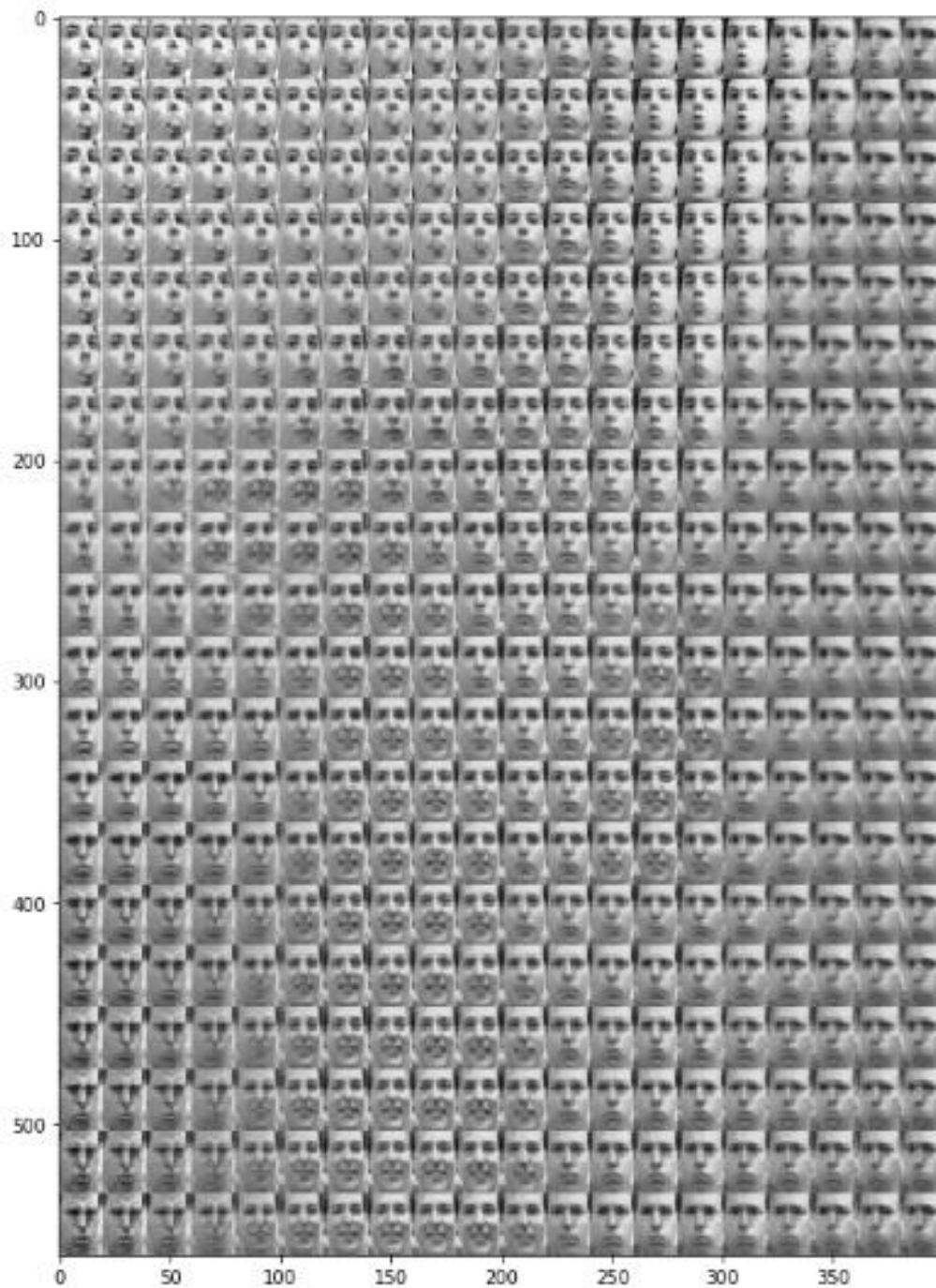


Figura 41 Manifold Frey Face

7.4 Espacio latente en 2D interactivo

Se ha desarrollado un widget interactivo con la librería matplotlib de Python que nos permite movernos por el espacio latente de dos dimensiones gracias a dos sliders que corresponde a la primera y segunda dimensión del espacio latente respectivamente, es decir representan las medias y las varianzas que se obtienen como salida del codificador.

Es un widget muy útil ya que nos permite desplazarnos por el espacio latente en sus dos dimensiones de forma gradual y ver como dígitos con similitudes estructurales van superponiéndose unos con otros, apreciándose la transición entre ellos de forma suave y progresiva.

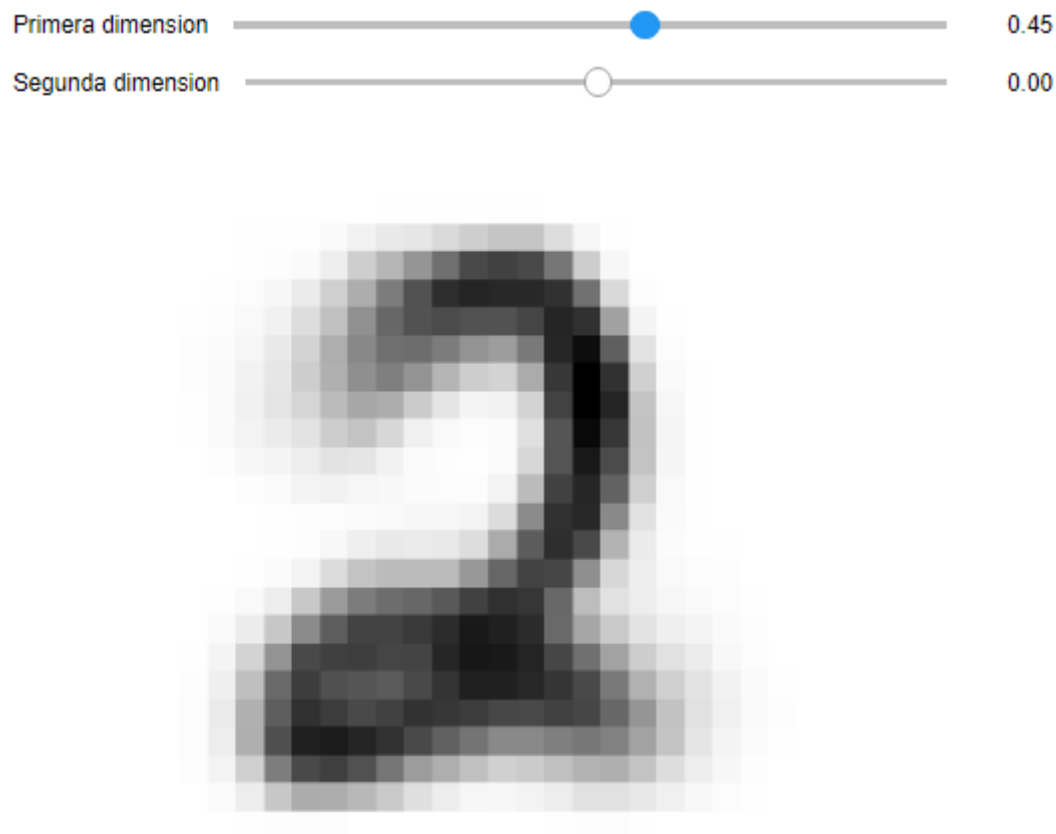


Figura 42 Widget interactivo para moverse por el espacio latente de dos dimensiones

8 Conclusiones y trabajo futuro

8.1 Conclusiones

Con este trabajo de fin de grado se pretende evidenciar que las redes neuronales no son una película futurista si no que representan el presente más inmediato. Se han repasado algunas de sus aplicaciones y se ha podido ver su amplio campo de acción, mejorando, por ejemplo, un sector tan importante como el de la salud.

El impacto de las redes neuronales en la sociedad esta tan presente entre nosotros que ya es casi imposible imaginarse la vida diaria sin utilizar de forma explícita o implícita alguna tecnología que esté usando redes neuronales (sistemas de recomendación, traductores, redes sociales...)

Actualmente, la IA y en particular las redes neuronales profundas son una de las líneas de investigación con más auge y repercusión dentro de la informática a nivel mundial. Conocer su funcionamiento es algo fundamental en la actualidad.

Antes de iniciar este trabajo de fin de grado se tenían una serie de objetivos que se han conseguido. Estos objetivos pretendían dar una visión general del mundo de las redes neuronales, profundizar en un modelo generativo profundo concreto: los Autoencoders Variacionales y realizar una implementación de este modelo con el fin de ser estudiado.

Se han realizado experimentos sobre el comportamiento que tiene el Autoencoder Variacional cuando modificamos la dimensión del espacio latente. Los resultados reflejan que el peor valor para la dimensión del espacio latente es cuando elegimos dos dimensiones, obteniendo un valor de la función de pérdida de 135.96. En contra posición conforme aumentamos la dimensionalidad del espacio latente obtenemos mejores resultados, que comienzan a estancarse cuando la dimensión es mayor que 20, llegando a un valor de la función de pérdida de 95,786.

Para finalizar se ha realizado un último experimento para comprobar como de buenas son las características de alto nivel (variables latentes) que extrae un VAE de las imágenes. Para ello se ha utilizado un clasificador lineal (regresión logística multinomial) con el objetivo de verificar cómo se comportan en tareas de clasificación estas variables latentes frente a las imágenes originales sin extracción de características de alto nivel previas. Para analizar los resultados de este experimento se ha implementado una herramienta gráfica interactiva que permite comparar la certeza en la clasificación con ambas formas de entrenamiento.

En los resultados obtenidos queda reflejado que usando las variables latentes como entrenamiento de un clasificador lineal se consiguen porcentajes de acierto superiores al 98%, dato que contrasta con el 92% de acierto máximo que se obtiene entrenado el mismo clasificador con las imágenes originales.

De este modo se concluye que las variables latentes aprendidas por un Autoencoder Variacional son más ricas en información que las imágenes originales, luego nos permiten entender mejor los datos.

Además, se han utilizado herramientas gráficas con la finalidad de comprender mejor los datos de una forma más visual. Para asimilar mejor la distribución de las variables latentes en el espacio se han pintado gráficos de dispersión, que nos han permitido ver como la red neuronal ha sido capaz de agrupar los diferentes dígitos del dataset MNIST correctamente, de manera que los números que son iguales permanecen cercanos en el espacio, y los dígitos distintos permanecen alejados. El concepto de igualdad usado aquí no se refiere tanto a su valor numérico sino más bien a similitudes estructurales a la hora de trazar un dígito a mano. En los resultados obtenidos se ha podido observar como los dígitos 8 y 3 estaban cercanos en el espacio latente, esto se explica porque a pesar de que el 8 y 3 sean números totalmente diferentes estructuralmente comparten similitudes cuando los pintamos a mano.

Para detectar que números están cercanos entre sí en el espacio latente además de los gráficos de dispersión ya mencionados, se ha desarrollado una herramienta interactiva que permite movernos con los valores de las medias y las varianzas sobre el espacio latente, generando transiciones progresivas de números, apreciando así los números que permanecen cercanos y comparten ciertas características dentro del espacio latente.

En todos los resultados obtenidos queda reflejada la potencia que tienen los Autoencoders Variacionales, tanto en tareas generativas como en tareas de clasificación.

8.2 Trabajo futuro

Las líneas de investigación podrían ir orientadas en aplicar el Autoencoder Variacional detallado en este trabajo de fin de grado, para resolver temas más complejos como puede ser usar este modelo generativo para generar otro tipo de datos como pueden ser video y música. Esto sería muy interesante porque se podrían usar estas técnicas para reconstruir partes de videos o música con ruido, mejoran las actuales técnicas que ya se usan en este sentido. Se podría usar un Autoencoder Variacional para generar contenido multimedia útil.

Otra dirección que se podría tomar de cara a investigaciones futuras es utilizar los VAE para diseñar nuevos prototipos de coches dentro del sector automovilístico.

También se podría aplicar el Autoencoder variacional junto con otras técnicas novedosas como usar Autoencoders e inferencia probabilística con datos perdidos [42]. Esta técnica ayuda a manejar los datos perdidos en la red, hecho que debería ayudar a mejorar los resultados. Otras de las técnicas que podrían suponer una línea de investigación a futuro es la subóptima inferencia que se produce en los VAE [43].

Por último, un amplio abanico de posibilidades de investigación queda abierto, para el uso de los Autoencoders Variacionales en aprendizaje semi-supervisado como el utilizado en esta sección 6.2.1. Pudiendo mejorar así posibles modelos de clasificación actuales que no usan variables latentes para su entrenamiento.

[42]

Referencias

- [1] Terminator3: <https://www.filmaffinity.com/es/film477986.html>
- [2] IDC (International Data Corporation) Predicciones para 2018 <http://www.blog-idcspain.com/predicciones-idc/>
- [3] ConversionUplift: <https://www.conversion-uplift.co.uk/glossary-of-conversion-marketing/artificial-intelligence/>
- [4] Deep Learning for healthcare: <https://www.nvidia.com/en-us/deep-Learning-ai/industries/healthcare/>
- [5] KDD papers: <http://www.kdd.org/kdd2017/papers/view/collaborative-variational-autoencoder-for-recommender-systems>
- [6] Disney research: <https://www.disneyresearch.com/publication/factorized-variational-autoencoder/>
- [7] Cornell University Library: <https://arxiv.org/pdf/1603.02514.pdf>
- [8] American Association of geographers: <https://aag.secure-abstracts.com/AAG%20Annual%20Meeting%202018/abstracts-gallery/702>
- [9] Qure.ai Blog: <http://blog.quire.ai/notes/using-variational-autoencoders>
- [10] Forbes: <https://www.forbes.com/sites/tomdavenport/2017/11/05/revolutionizing-radiology-with-deep-Learning-at-partners-healthcare-and-many-others/#63e01ad85e13>
- [11] NetWorldWorld: <https://www.networkworld.com/article/3183745/health/how-deep-Learning-is-transforming-healthcare.html>
- [12] Redes Neuronales: http://www.itnuevolaredo.edu.mx/takeyas/apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/RNA/Redes%20Neuronales2.pdf
- [13] Regla de aprendizaje de Hebb: https://es.wikipedia.org/wiki/Teor%C3%ADa_hebbiana
- [14] Figura de una Neurona: <http://www.educarchile.cl/ech/pro/app/detalle?ID=137486>
- [15] Analogía entre neurona biología y neurona artificial: <http://www.um.es/LEQ/Atmosferas/Ch-VI-3/F63s4p3.htm>
- [16] Descenso de gradiente: https://en.wikipedia.org/wiki/Gradient_descent
- [17] Variational Autoencoders: <https://arxiv.org/pdf/1606.05908.pdf>
- [18] EMC: https://es.wikipedia.org/wiki/Error_cuadr%C3%A1tico_medio
- [19] Cross-Entropy: https://en.wikipedia.org/wiki/Cross_entropy
- [20] Variational Autoencoders: <https://www.doc.ic.ac.uk/~js4416/163/website/autoencoders/variational.html>
- [21] Divergencia KL: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- [22] Christopher M. Bishop: Pattern Recognition and Machine Learning
- [23] P. Kingma y Max Welling. Auto-encoding Variational Bayes: <https://arxiv.org/pdf/1312.6114.pdf>
- [24] Rajesh Ranganath, Sean gerrish y David M. Blei. Black Box Variational Inference: <https://arxiv.org/pdf/1401.0118.pdf>
- [25] Music generation: <https://arxiv.org/pdf/1705.05458.pdf>
- [26] Extracting a Biologically Relevant Latente Space from cancer Transcriptomes <https://www.biorxiv.org/content/early/2017/08/11/174474>
- [27] Juego de suma cero: https://en.wikipedia.org/wiki/Zero-sum_game
- [28] Generative Adversial Networks: <https://arxiv.org/pdf/1406.2661.pdf> y <http://shashwatverma.com/generative-adversarial-networks.html>

- [29] Blog de Agustinus Kristiadi: <https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/>
- [30] Jaan Altosaar-Encoder-decoder: <https://jaan.io/what-is-variational-autoencoder-vae-tutorial/>
- [31] Distribución Gaussiana: https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal
- [32] Teorema de bayes: https://es.wikipedia.org/wiki/Teorema_de_Bayes
- [33] Metropolis-hastings: https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm
- [34] Metodo de Montecarlo: https://es.wikipedia.org/wiki/M%C3%A9todo_de_Montecarlo
- [35] Divergencia Kullback-Leibler: https://es.wikipedia.org/wiki/Divergencia_de_Kullback-Leibler
- [36] Variational inference: <https://arxiv.org/pdf/1601.00670.pdf>
- [37] Distribución de Bernoulli: https://es.wikipedia.org/wiki/Distribuci%C3%B3n_de_Bernoulli
- [38] MultiLayer Perceptron: https://en.wikipedia.org/wiki/Multilayer_perceptron
- [39] Efficient BackProp: <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>
- [40] Adam Optimizer: <https://MachineLearningmastery.com/adam-optimization-algorithm-for-deep-Learning/>
- [41] Adam: A Method for Stochastic Optimization: <https://arxiv.org/pdf/1412.6980.pdf>
- [42] Autoencoders and Probabilistic Inference with Missing Data: <https://arxiv.org/abs/1801.03851>
- [43] Inference Suboptimality in Variational Autoencoders: <https://arxiv.org/abs/1801.03558>
- [44] Tensorflow: <https://www.tensorflow.org/>
- [45] Jordi Torres “Hello World”
- [46] MNIST Ejemplo: https://www.tensorflow.org/get_started/mnist/beginners
- [47] Trick Reparameterization: <https://stats.stackexchange.com/questions/199605/how-does-the-reparameterization-trick-for-vaes-work-and-why-is-it-important>
- [48] Variational Autoencoders Explained: <http://kvfrans.com/variational-autoencoders-explained/>
- [49] The Variational auto-encoder: <https://ermongroup.github.io/cs228-notes/extras/vae/>
- [50] Hamidreza Saghir-An intuitive understanding of Variational autoencoders without any formula: https://hsaghir.github.io/data_science/denoising-vs-variational-autoencoder/
- [51] Vindula Jayawardana- “Autoencoders- Bits and bytes of deep Learning”: <https://towardsdatascience.com/autoencoders-bits-and-bytes-of-deep-Learning-eaba376f23ad>
- [52] Daniel Hernández Lobato: https://dhnzl.files.wordpress.com/2016/12/iwae_uncertain.pdf
- [53] Colah’s blog- Visualization MNIST: <https://colah.github.io/posts/2014-10-Visualizing-MNIST/>
- [54] Autoencoders for Image Classification: <https://es.mathworks.com/help/nnet/examples/training-a-deep-neural-network-for-digit-classification.html?requestedDomain=true>
- [55]

Glosario

VAE	Variational Autoencoders (En Castellano: Autoencoders Variacional)
GAN	Generative Adversial Networks
VI	Variational Inference
IA	Inteligencia Artificial
MLP	MultiLayer Perceptrón

Anexos

A Optimizador Adam

El algoritmo de optimización Adam[40] es una variante del clásico descenso de gradiente estocástico que actualmente está teniendo una adopción muy elevada en problemas de lenguaje artificial y aprendizaje profundo en visión artificial.

Adam se puede usar de manera análoga al tan conocido método “descenso de gradiente estocástico”. A diferencia del clásico algoritmo de descenso de gradiente estocástico que mantiene en todo su entrenamiento la misma tasa de aprendizaje para todas sus actualizaciones de pesos. El algoritmo Adam tiene una tasa de aprendizaje para cada peso de la red y se adapta individualmente a medida que se desarrolla el entrenamiento.

Para calcular las tasas de aprendizaje adaptativas individuales utiliza las estimaciones de los momentos primeros y segundo del gradiente.

Este algoritmo presenta las siguientes ventajas respecto a técnicas de descenso de gradientes típicas:

- Es computacionalmente menos costoso y eficiente.
- Buenos resultados para problemas con gradientes ruidosos
- Implementación sencilla

B Manual de instalación

Mi repositorio personal²⁰ contiene una serie de notebooks ipython con el código del Autoencoder Variacional, los clasificadores, ejemplos visuales de los resultados y ejemplos interactivos. Estos ejemplos interactivos permiten al usuario interactuar con los datos del espacio latente.

Para poder ejecutar estos códigos y herramientas de visualización de datos se necesita activar una serie de opciones y tener instalado una serie de librerías. Aquí dejo un pequeño manual para la instalación de todas ellas.

Requisitos previos

- Tener instalada una versión de Python, recomendada Python 2.7+, usada en el manual versión 3.5

Manual paso a paso

1. Descargar e instalar Anaconda

<https://www.anaconda.com/download/>

2. Preparar entorno Conda

```
~$ conda create -n nombreDelEntorno Python=3.5
```

3. Activar entorno conda

```
~$ activate nombreDelEntorno
```

Nota: El prompt debe cambia y poner el nombre del entorno.

4. Instalar Jupyter Notebook Python.

```
~$ pip install jupyter
```

5. Instalar y activar widgets interactivos en Jupyter Notebook

```
~$ pip install ipywidgets
~$ jupyter nbextension enable -py widgetsnbextension
```


6. Instalar librería matplotlib

```
~$ pip install matplotlib
```

7. Instalar librería Tensorflow

```
~$ pip install --ignore-installed --upgrade tensorflow
```

8. Instalar librería numpy

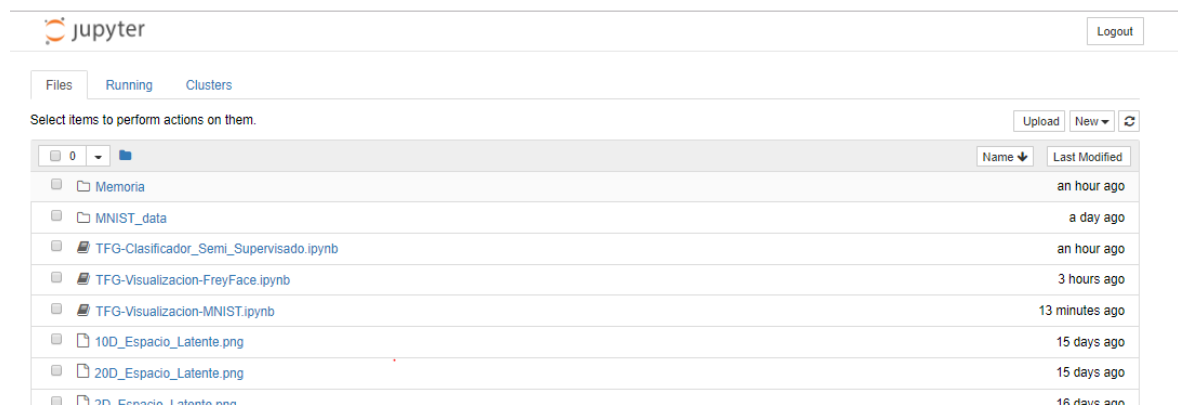
```
~$ pip install numpy
```

9. Abrir Jupyter Notebook Python

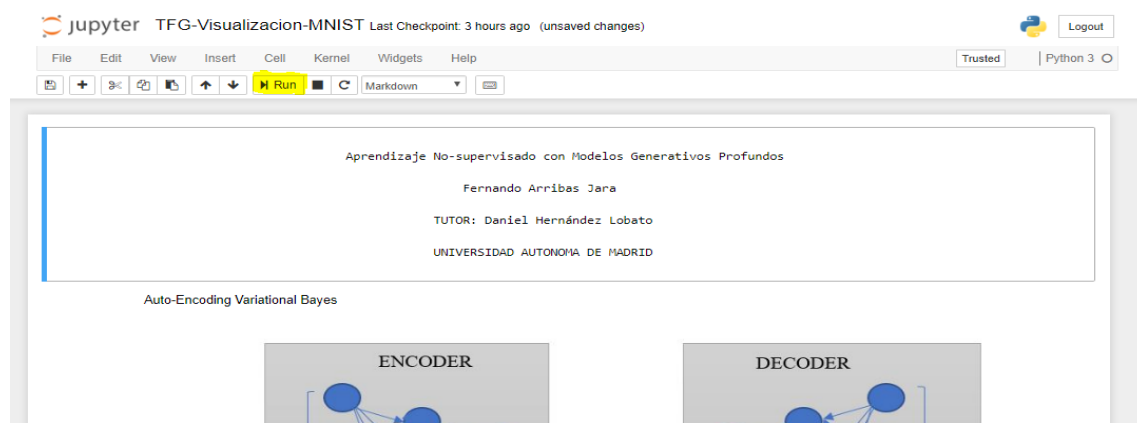
```
~$ jupyter notebook
```

10. Ejecutar Notebook

a. Primero seleccionas el Notebook a ejecutar



b. En segundo lugar, ejecutamos bloque por bloque de código con este botón.



C Clasificador Semi-Supervisado con VAE

```
#Variables de Train
num_epocas =5
batch_size = 100
Learning_rate = 1e-1
n_datos = mnist.train.num_examples
num_batch = int(n_datos / batch_size)

#Variables del dataset
tam_imagen = 784 # Imagenes de 28x28 pixels
tam_latente = 50
tam_hidden_layer = 500

input_encoder = tf.placeholder(tf.float32, shape=[None, tam_imagen]) #Entrada de
datos, imagenes

#Semilla aleatoria
tf.set_random_seed(0)

#ENCODER

#Pesos y biases
#Primera capa oculta
W_encoder1 = tf.Variable(tf.random_normal([tam_imagen, tam_hidden_layer], stddev=
tf.pow(float(tam_imagen), -0.5)))
b_encoder1 = tf.Variable(tf.random_normal([tam_hidden_layer], stddev= tf.pow(floa
t(tam_hidden_layer), -0.5)))

#Segunda capa oculta
W_encoder2 = tf.Variable(tf.random_normal([tam_hidden_layer, tam_hidden_layer], s
tddev= tf.pow(float(tam_hidden_layer), -0.5)))
b_encoder2 = tf.Variable(tf.random_normal([tam_hidden_layer], stddev= tf.pow(floa
t(tam_hidden_layer), -0.5)))

W_z_var = tf.Variable(tf.random_normal([tam_hidden_layer,tam_latente], stddev=tf.
pow(float(tam_hidden_layer), -0.5)))
b_z_var = tf.Variable(tf.random_normal([tam_latente], stddev=tf.pow(float(tam_lat
ente), -0.5)))

W_z_mean = tf.Variable(tf.random_normal([tam_hidden_layer,tam_latente], stddev=tf.
pow(float(tam_hidden_layer), -0.5)))
b_z_mean = tf.Variable(tf.random_normal([tam_latente], stddev=tf.pow(float(tam_la
tente), -0.5)))

#Model del Encoder
encoder_capa1 = tf.matmul(input_encoder, W_encoder1) + b_encoder1
encoder_capa1 = tf.nn.relu(encoder_capa1)
encoder_capa2 = tf.matmul(encoder_capa1, W_encoder2) + b_encoder2
encoder_capa2 = tf.nn.relu(encoder_capa2)
#Mean
z_mean = tf.matmul(encoder_capa2,W_z_mean)+b_z_mean

W_clasificador1 = tf.Variable(tf.random_normal([tam_latente, 10], stddev=tf.pow(f
loat(tam_latente), -0.5)))
b_clasificador1= tf.Variable(tf.random_normal([10], stddev=tf.pow(float(tam_laten
te), -0.5)))
```

```

y_clasificador1 = tf.nn.softmax(tf.matmul(z_mean, W_clasificador1) + b_clasificad
or1)

y_clasificador1_labels = tf.placeholder(tf.float32, [None, 10])

cross_entropy1 = tf.reduce_mean(-tf.reduce_sum(y_clasificador1_labels * tf.log(y_
clasificador1), reduction_indices=[1]))
train_step1 = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_ent
ropy1)

sess1 = tf.InteractiveSession()
tf.global_variables_initializer().run()

print ("Entrenando modelo...")
for epoca in range(1, num_epocas+1):
    average_coste = 0
    for _ in range(num_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        _, coste = sess1.run([train_step1, cross_entropy1], feed_dict={input_encod
der: batch_xs, y_clasificador1_labels: batch_ys})
        average_coste = (average_coste + coste)
    print('Epoca: %d Cross Entropy= %f' % (epoca, average_coste/num_batch))

#Test
prediccion1 = tf.equal(tf.argmax(y_clasificador1, 1), tf.argmax(y_clasificador1_l
abels, 1))
score1 = tf.reduce_mean(tf.cast(prediccion1, tf.float32))
print ("Score")
print(sess1.run(score1, feed_dict={input_encoder: mnist.test.images, y_clasificad
or1_labels: mnist.test.labels}))

```

D Clasificador supervisado con imágenes originales

```
input_clasificador2 = tf.placeholder(tf.float32, shape=[None, tam_imagen]) #Entrada de datos, imagenes
W_clasificador2 = tf.Variable(tf.random_normal([tam_imagen, 10], stddev=tf.pow(float(tam_latente), -0.5)))
b_clasificador2 = tf.Variable(tf.random_normal([10], stddev=tf.pow(float(tam_latente), -0.5)))
y_clasificador2 = tf.nn.softmax(tf.matmul(input_clasificador2, W_clasificador2) + b_clasificador2)

y_clasificador2_labels = tf.placeholder(tf.float32, [None, 10])

cross_entropy2 = tf.reduce_mean(-tf.reduce_sum(y_clasificador2_labels * tf.log(y_clasificador2), reduction_indices=[1]))
train_step2 = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy2)

sess2 = tf.InteractiveSession()
tf.global_variables_initializer().run()

print ("Entrenando modelo...")
for epoca in range(1, num_epocas+1):
    average_coste = 0
    for _ in range(num_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        _, coste = sess2.run([train_step2, cross_entropy2], feed_dict={input_clasificador2: batch_xs, y_clasificador2_labels: batch_ys})
        average_coste = (average_coste + coste)
    print ('Epoca: %d Cross Entropy= %f' % (epoca, average_coste/num_batch))

#Test
prediccion2 = tf.equal(tf.argmax(y_clasificador2, 1), tf.argmax(y_clasificador2_labels, 1))
score2 = tf.reduce_mean(tf.cast(prediccion2, tf.float32))
print ("Score")
print(sess2.run(score2, feed_dict={input_clasificador2: mnist.test.images, y_clasificador2_labels: mnist.test.labels}))
```

E Dataset MNIST [45]

MNIST es un conjunto de datos compuesto por imágenes de 28x28 píxeles, de dígitos hechos a mano todas ellas en blanco y negro. Se trata del conjunto de datos más utilizado para la iniciación en técnicas y Machine Learning de reconocimiento de patrones.

Es un dataset sencillo y que casi todas las librerías disponibles para la implementación de modelos de aprendizaje automático disponen de él, ya tratado y listo para usarse. Como es el caso de Tensorflow que con tan solo una línea de código podemos hacer uso de él.

Está formado por 60000 imágenes para entrenamiento y 10000 imágenes para test, estas imágenes tienen esta forma:



Figura 43 Imágenes MNIST [46]

Además, todas las imágenes disponen de una etiqueta que indica a que dígito corresponden.

Para utilizar las imágenes de MNIST en los modelos de Machine Learning se tratan como si fueran una matriz de números de $28 \times 28 = 784$. Donde cada posición de la matriz indica la intensidad del píxel entre un intervalo de 0 y 1.

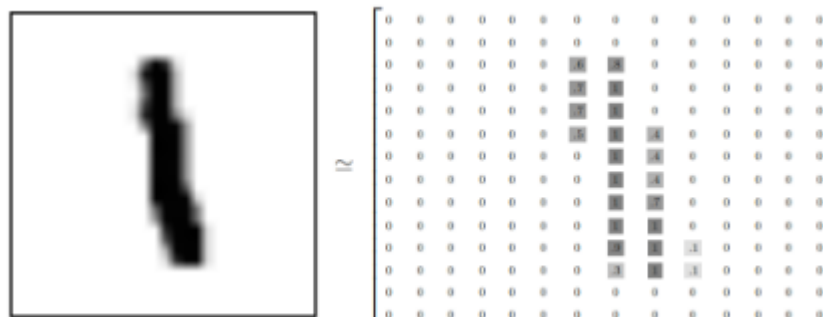


Figura 44 Representación de un dígito del dataset MNIST en un array de números