3. ANTECEDENTES

En este capitulo se desarrollaran en detalle los aspectos básicos y los fundamentos teóricos/prácticos que han hecho posible el desarrollo del proyecto. Para ello se analiza detalladamente todas las tecnologías usadas en el proyecto. Dividiendo este capitulo en dos partes, una parte de estudio centrada en la parte teórica del TFG, esta parte se desarrollara explicando y justificando teóricamente el uso de los Autoencoders variacionales, comenzando por el Deep Learning, extendiéndonos a los Autoencoders y finalizandola con los Autoencoders Variacionales.
La segunda parte contendrá la parte tecnológica y el desarrollo del Sistema, profundizando en las tecnologías, englobando desde la nube hasta los lenguajes de programación usados.

3.1 ESTUDIO

En esta parte detallaremos la parte teórica del proyecto, comenzando por el Deep Learning, tras ello profundizaremos en los autoencoders y sus características y finalizaremos explicando los fundamentos del sujeto de estudio del presente TFG, los Autoencoders Variacionales (VAE).

3.1.1 Deep Learning

El Deep Learning es un área del Machine Learning que trata de optimizar las perdidas y las complejidades derivadas de añadir capas ocultas a una red neuronal.

Para aprender que es el Deep Learning primero debe verse que son las redes neuronales y sus tipos de aprendizaje.

REDES NEURONALES ARTIFICIALES (ANN)

Las redes neuronales son una representación abstracta de como funciona el cerebro humano, al igual que nuestro cerebro, están basadas en el concepto de neurona. Una neurona no es mas que un elemento que recibe una entrada o entradas y mediante una funciona matemática que aplica a la entradas produce una salida, en función de dicha salida se podría decir si esta neurona se ha activado o no.

Esta función de activación puede ser lineal (ecuación cuyo dibujo no cambia de plano) lo cual vuelve a nuestra función binaria, con dos estados, activada o desactivada dependiendo de la función, pero en la mayoria de casos si se aproxima a 0 esta desactivada y si se aproxima a 1 esta activada. Otro tipo de funciones son las no lineales (el dibujo de la ecuación pasa por distintos planos), que son funciones que permiten normalizar datos entre valores, dependiendo de la función como podemos ver en la Figura 3.1.

En la figura 3.1 vemos 3 funciones:

a) f(x) = mx, es una función lineal donde m es la pendiente, como hemos comentado antes, esta función comparte un problema grave con las funciones binaria, que no son normalizables, por lo que a la larga los valores de la entrada escalan sin limite.
b) f(x) = tanh x = $\frac{\sinh x}{\cosh x}$, es la función de la tangente hiperbólica, una función no lineal que se puede normalizar entre dos valores, normalmente (-1, 1).

c) $f(x) = \frac{1}{1+e^{x}}$, es la función sigmoidal, al igual que la anterior es no lineal lo que permite normalizar valores y por lo tanto describir una evolución en las neuronas para su activación.

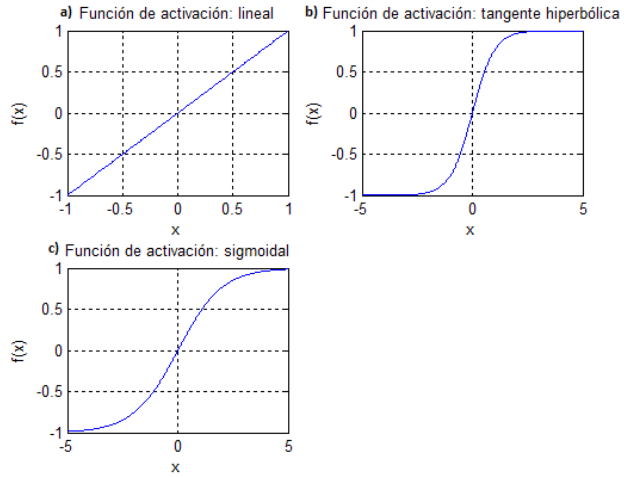

Figura 3.1: Ejemplo de funciones de activación

Por lo tanto una red neuronal artificial, es una serie de neuronas artificiales interconectadas en una red de forma que se comunican entre ellas y producen de una entrada, una salida. Estas redes neuronales artificiales organizan las neuronas por capas, de forma que puedes crear varias capas de distintos tamaños formando una red, como se muestra en la figura 3.2.

Ademas hay distintos tipos de capas dependiendo de donde se coloquen en la red, por lo que podríamos tener, capas de entrada, capas de salida, y capas ocultas. A medida que insertamos capas ocultas la red se vuelve mas potente (mas efectiva, ya que reduce el error en la generalización), pero también se vuelve mas compleja.

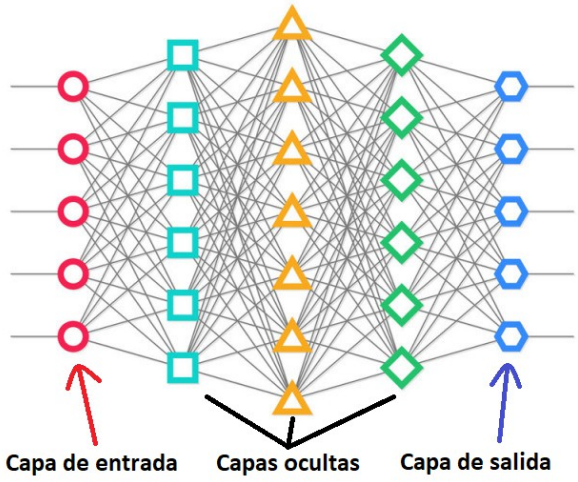

Figura 3.2: Ejemplo de red neuronal con 3 capas ocultas

Una red neuronal al final esta basada en como aprende (el Learning de Deep Learning, que se vera mas adelante) y en como se organizan y comunican sus capas, es decir, es su arquitectura o estructura, la cual es diseñada en base a la experimentación, el ensayo y error en la mayoría de casos. Esta arquitectura es la que lo hace llamar Deep al Deep Learning, ya que este se basa en modelos cuya estructura es profunda, es decir, tiene varias capas anidadas (eso se consigue cuando hay varias capas ocultas).

APRENDIZAJE EN REDES NEURONALES

Para que una red neuronal sea funcional, debe asignar pesos a las conexiones entre neuronas, de esa forma construye la comunicación entre las distintas neuronas. Esta asignación de pesos (inicializandolos a valores aleatorios) se hace de forma automática por la propia red mediante su entrenamiento, el entrenamiento es la aplicación de un algoritmo de aprendizaje a al red de forma que ella misma reajuste los pesos entre neuronas hasta que lleguemos al resultado esperado.

Una algoritmo de aprendizaje por tanto es un algoritmo que es capaz de aprender a partir de datos, pero , ¿que se quiere decir con aprender? , existe una definición Proporcionada por Tom Mitchell:

“Un programa de ordenador aprende de la experiencia E respecto a alguna clase de tarea T y un rendimiento P si el rendimiento en T,moderado por P, aumenta con experiencia E”[1]

Lo que nos da 3 claves o pilares de un algoritmo de Machine Learning:

- Una tarea T
- Un Rendimiento P
- Una experiencia E

Hay distintos tipos de tareas T [2], entre ellos destaco la síntesis de muestras. La síntesis de muestras o síntesis y muestreo es una tipo de tarea en la que al algoritmo se le pide generar nuevos ejemplos que son similares a los del conjunto de datos de entrenamiento. Esto puede ir desde sintetizar el audio de una frase o generar muestras de fuentes y tipos de letras a partir de un determinado conjunto de fuentes ya conocidas.

El rendimiento es una medida de los algoritmos que nos dice su efectividad, trata de mejorarse conforme se entrena, por lo que normalmente se traduce el rendimiento en el ratio de error, y en cuanto mas bajo, mejor rendimiento.

El factor experiencia nos divide los algoritmos dependiendo de si la experiencia es **supervisada** o **no supervisada**. Los algoritmos con experiencia supervisada o Algoritmos de Aprendizaje Supervisado son los que en el aprendizaje asocian a una entrada, una salida esperada, de forma que el rendimiento es medido en lo lejana que es nuestra salida a la esperada.
Los Algoritmos de Aprendizaje no supervisado son algoritmos cuyo rendimiento se mide en la calidad de su propia salida sin depender de nada a priori.

Para ejemplificar lo visto lanzamos dos ejemplos:

Un algoritmo supervisado seria una clasificación de los tipos de iris, de forma que sabemos a priori y se lo decimos al algoritmo, que iris pertenecen a que clase.

Un algoritmo no supervisado seria una clasificación de flores en distintos tipos de clases, sin saberlas a priori, debería analizar características y asociar flores a clases de flores.

Cuando añadimos capas ocultas a una red neuronal, este aprendizaje sea del tipo que sea, se vuelve mas complejo, ya que se debe corregir el error de una forma mas compleja, mas profunda. EL deep learning intenta técnicas con el aprendizaje no supervisado, o aprendizajes supervisados no normativos (como las redes neuronales convulacionales) para reducir la complejidad que deriva de profundizar una red neuronal.

3.1.2 Autoencoders

Un autoencoder es una red neuronal entrenada para que copie su entrada en su salida. Internamente tiene una capa oculta h que codifica la entrada, esta arquitectura se representa en la figura 3.3, por lo que un autoencoder consta de dos partes:

1. Un codificador (encoder) que toma la entrada y la codifica en **h** (h=f(x))
2. Un decodificador (decoder) que toma como entrada **h** y su salida es una copia de la entrada del encoder (r=g(h))

A priori puede ser poco útil su uso ya que si es exitoso su función será g(f(x))=x pero pueden ser restringidos para copiar ciertas entradas que sean parecidas al dataset (conjunto de datos) de entrenamiento.

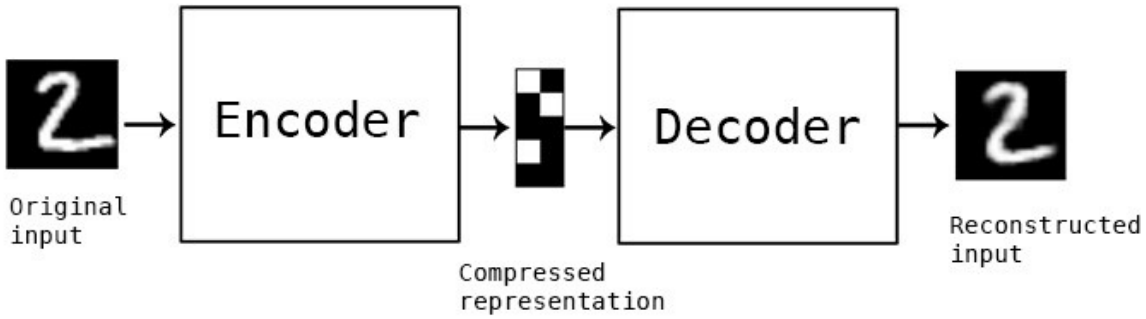

Figura 3.3: Estructura de un autoencoder

Copiar una entrada no es demasiado útil como hemos visto, lo que nos lleva a preguntarnos ¿que ventaja tienen los Autoencoders?, hay varias respuestas pero todas coinciden en que pueden comprimir datos a nivel de codificarlos por ejemplo en una variable de x dimensiones (el sujeto de estudio de este TFG) lo que dice que son capaces de asimilar características y codificarlas en su capa oculta.

Los Autoencoders tiene 3 características principales [3]:

1. Son específicos al nivel de datos, es decir, cuando un Autoencoder es entrenado con un tipo de datos, solo puede tratar ese tipo de datos, por ejemplo, un autoencoder entrenado con fotos de caras no trabajara bien con fotos de arboles al no tener las mismas características.
2. Tienen perdidas, lo que nos dice que los datos producidos por el decoder no son idénticos a los de la entrada ya que hay perdidas en la compresión de los datos diferenciandolos de la compresión aritmética sin perdidas.
3. Aprenden automáticamente de ejemplos de datos, esto nos dice que su aprendizaje es no supervisado y que son muy útiles ya que no necesitan nuevos algoritmos para cambiar de datos de entrenamiento, solo un conjunto de datos apropiado. Aunque a efectos prácticos a los autoencoders se les llama auto supervisados, ya que no tiene un algoritmo supervisado peor su algoritmo de aprendizaje no supervisado se basa en su propia entrada, por lo que se auto supervisan.

Estas características nos dicen mucho de su utilidad y su funcionamiento en general, pero no nos especifican como trabajan, que hace al autoencoder asimilar características y codificarlas para poder reproducirlas mas tarde. La respuesta a esto esta en el conjunto de entrenamiento usado (su aprendizaje), su función de perdida (como calculan la perdida en el entrenamiento), su capa intermedia (codificación) y su estructura (las capas dentro del encoder y el decoder)

Siempre buscamos evitar que aprendan la función identidad **(**f(g(x))=x**)**, ya que cuando se codifica un autoencoder de manera muy potente deriva a esa función que simplemente copia la entrada sin aprender características.

Conjunto de entrenamiento

Hemos dicho que los autoencoders son específicos a nivel de datos, es decir, que solo funcionan con los datos parecidos a los datos con los que han sido entrenados. Cuando ponemos una capa oculta, o código, h mas pequeña que la entrada, forzamos al autoencoder a priorizar en el aprendizaje que aspectos copiar de la entrada, por lo que recurre a las características de la misma.

Entonces el conjunto de entrenamiento influye, ya que podemos usar distintos conjuntos para generar determinadas características, de forma que podemos hacer que un autoencoder reconstruya datos incompletos (usando un conjunto de entrenamiento con ruido y forzando al autoencoder a eliminarlo) o simplemente que aprenda características y propiedades de caras.

Capa intermedia

Al diseñar un autoencoder debemos pensar en el tamaño de la capa de código, ya que cuanto mas pequeña mayor nivel de compresión, pero mas complejo se vuelve el autoencoder, al igual que cuanto mas grande es mas sencillo es el aprendizaje y al compresión, pero podemos acercarnos mas a la función identidad.

Cuando la capa de código es mas pequeña que la de entrada, se le llama undercomplete al autoencoder con dicha capa y es forzado a aprender las características de la entrada y comprimirlas. Esto se traduce en su función de perdida que se vuelve L(x,g(f(x))) donde L es la función de perdida que penaliza cuando x y g(f(x)) (el autoencoder copiando la entrada) no se parecen, esta función es similar al error cuadrático medio que veremos mas adelante.

Cuando la capa intermedia es de igual tamaño el autoencoder pasa a ser una “fotocopiadora” en la que la entrada se copia idéntica en al salida y no se aprende ninguna característica, por lo que su aprendizaje falla.

Cuando la capa intermedia es de mayor tamaño que la capa de entrada el autoencoder se llama overcomplete y es otro caso de fallo en el aprendizaje, se veta mas a los autoencoders de forma que ni uno con funciones lineales de activación podría aprender algo útil. Las funciones de activación y como se relacionan con las funciones de perdida se vera mas adelante.

Si definimos el tipo de dato de la capa intermedia tambien cambiamos el autoencoder, ya que la representación de las características cambia y por lo tanto cambia la manera de decodificarlas. Por ejemplo podemos definir el dato como un punto, un valor, o como un distribución y definir las funciones del encoder y el decoder como probabilidades en una distribución Pencode(h,x) y Pdecode(x,h) esto quiere decir que las funciones pasarían de el valor para esta x de h a la probabi9lidad de que existiendo esta x sea h.

Estructura

La estructura de un autoencoder a nivel general es siempre la misma, un decoder, un encoder y una capa intermedia h en la que esta el código o compresión. Pero la estructura del encoder y el decoder intervienen en la potencia del autoencoder, ya que cada uno de estos dos elementos puede formar un sistema el solo.

Los decoder y encoder pueden funcionar con solo una capa pero si les añadimos capas ocultas a cada uno de ellos, el autoencoder se vuelve mas potente, pero puede acabar aprendiendo al función identidad que tratamos de evitar.

A mas capas insertemos en el encoder, o en el decoder, los volveremos mas potentes pero hay que tener cuidado con su potencia, esta potencia también es regulable con los aspectos anteriores, ya que un decoder y un encoder potentes con una compresión muy pequeña (una capa intermedia de poco tamaño) puede ser muy útil.

Funciones de perdida

Las funciones de perdida pueden variar como aprende una red neuronal, ya que miden el error y son las encargadas de marcarle a la red neuronal la medida de su aprendizaje en función de cuanta perdida tiene la entrada conforme a la salida deseada.

Las funciones de perdida definen el tipo de autoencoder, ya que definen el como aprende y por lo tanto definen que características prioriza el algoritmo de aprendizaje. Hay varios tipos de funciones de perdida, aquí daremos varios ejemplos para ver como puede afectar estas funciones al aprendizaje del autoencoder.

La función de perdida mas básica es la función L(x,g(f(x)), que nos dice que la perdida es la medida en la que la entrada y la salida son idénticas o no. Esta función básica trata de realizar la misma función que el Análisis de Componentes Principales [4] (PCA, sus siglas en ingles). Este análisis es una técnica

de estadística en la que se trata de analizar una tabla que representa las observaciones de varias variables dependientes y correlacionadas. Su objetivo es extraer la información mas importante de la tabla y expresarla como variables ortogonales llamadas *componentes principales* al representar estas variables como puntos en uno o varios mapas, se puede ver la medida de semejanza entre ellas.

Una función que cambia el paradigma de los autoencoders es la función L(x,g(f(x))) + Ω(h), esta función añade un nuevo parámetro (Ω(h)) que mide la optimización o la realización de una tarea concreta, por ejemplo la clasificación en clases. Este parámetros es llamado penalización por dispersión.

Como estas funciones hay mas, cada uno puede mejorar el aprendizaje, transformarlo en otro tipo de aprendizaje o incluso como la ultima cambiar el paradigma del autoencoder poniendo un parámetro que obliga al autoencoder a cumplir una determinada tarea.

3.1.3 Variational Autoencoders

Estos autoencoders son el sujeto de estudio de nuestro proyecto, su mayor particularidad que los diferencia de los demás es como codifica la capa h, o capa oculta del autoencoder. Este modelo de Autoencoder usa una variable latente.

Variable latente

Una variable latente es una variable oculta, es decir, es una variable que no se conoce su valor, o no es observable dicho valor, solo puede ser inferido mediante modelos matemáticos a partir de otras variable que si son observables.

Se usa una variable latente porque el VAE antes de producir una salida, comprueba que su salida y los trazos que la forman coinciden con la entrada pero no podemos saber que valores de z (variable oculta) han generado dicho dato, o dichos trazos.

Para decir que nuestro modelo ha funcionado, debemos estar seguros de que para cada elemento X en el dataset hay uno o varios valores de la variable latente que producen algo similar a X. Inferimos z (variable latente) mediante una función de probabilidad de densidad o función de densidad P(z) definida sobre un espacio o dominio Z.

Pero debemos adaptar el modelo a esta variable, por lo que el encoder y el decoder cambian. El encoder ahora debe seguir una distribución Gaussiana o distribución normal (Distribución cuyo centro tiene

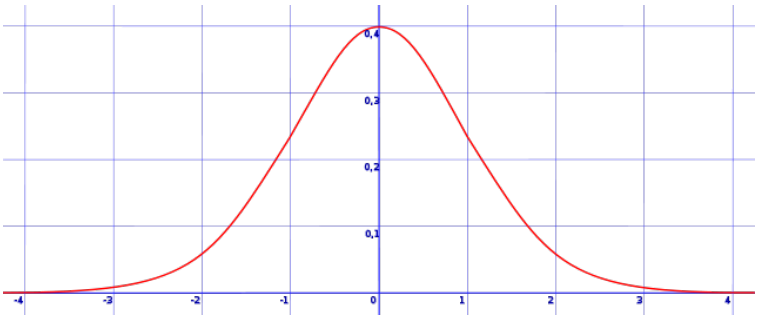

Figura 3.4: Ejemplo de una distribución normal o campana de Gauss, donde el centro es el mas probable y esta probabilidad disminuye conforme nos acercamos a los extremo. Este ejemplo ademas consta de media = 0 y varianza = 1.

mas probabilidades, las cuales disminuyen según te separas a los extremos, figura 3.4) ya que gracias a ello se pueden usar técnicas de optimización para aproximar los valores de X a z.

El decoder entonces debe aprender que distribución seguir de forma que generemos la distribución P(X|z) sin importar el tipo de distribución que sea, esta debe poder imitar una X del dataset dada una o varias z.

Variational Autoencoder como generadores de objetos

Usando este modelo de variable latente este tipo de autoencoder se usa como generador de objetos, la implementación de este modelo de variable latente se hace por medio de un principio básico de la distribuciones normales o Gaussianas, dada una distribución normal P(X) podemos aplicarle a sus valores una función g(z) de forma que convirtamos esa distribución en otra completamente distinta. Nuestro autoencoder aprende esta función de los datos de entrada.

En nuestro caso z se representa como una o varias distribuciones normales, este numero de distribuciones n sera el numero de dimensiones que tendrá la variable, es decir, el tamaño de la capa oculta del autoencoder. Para definir la distribución normal que sigue z usamos la media (μ) y la varianza (σ^2) para que se quede con forma de N(μ, σ^2). La media y la varianza son extraídas de la salida del encoder.

Nuestro objetivo es que esa distribución normal se parezca lo máximo posible a una normal con media 0 y varianza 1. Para ello necesitamos entrenar el VAE de forma que de la media y la varianza extraídas del encoder se forme dicho objetivo. Para ello usamos la divergencia KL.

La divergencia KL también llamada Divergencia de Kullback-Leibler es un método mediante el cual se puede comparar dos distribuciones cuantificar su similitud.

En el entrenamiento el autoencoder se encarga de inferir la formula g(z) que dada la distribución P(z) y dada una o varias X donde P(z|X) exista una distribución P(X|z). De esta forma se decodifica la variable latente en datos pertenecientes al dominio de X (nuestro dataset de entrenamiento).

Cuando el VAE esta entrenado podemos descartar toda la parte del encoder y quedarnos solo con la variable latente y el decoder, al que le pasamos valores pertenecientes a la distribución P(z) generando así objetos nuevos y originales a partir de los datos de entrenamiento.

Esto es posible ya que para cada X existe una distribución P(X|z) donde para cada z hay una o mas X parecida a la X original, por lo que se generan modificaciones del propio dato original.

3.2 DESARROLLO

En esta parte detallaremos la parte técnica del proyecto, dividiendo el tema en dos partes, backend y frontend las cuales se encuentran en un punto común, las tecnologías API REST. En este capitulo detallaremos las tecnologías que caracterizan al proyecto.

3.2.1 Computación en la nube

La computación en al nube es un nuevo paradigma que ha salido en los últimos años, en resumidas cuentas es que la tarea de computo, calculo, etc. la carga operativa la tenga el servidor de forma que los clientes hagan el mínimo esfuerzo/calculo/computo posible para usar un servicio en la red.

Una definición mas exacta seria, la computación en la nube en un modelo que permite proporcionar servicios de forma ubicua, segura y en tiempo real cuyos sistemas pueden ser desplegados y mantenidos con muy poco esfuerzo.

La computación en la nube se basa en una características esenciales, ademas de en modelos de despliegue y modelos de servicio [5].

Características

- Un consumidor puede usar capacidades de computación, por ejemplo la hora del servidor, automáticamente sin la necesidad que el humano interaccione con los promovedores de servicio.
- Los servicios pueden ser accedidos desde un grupo heterogéneo de mecanismos, así como móviles, tables, navegadores, etc.
- Computación ubicua, esto quiere decir que no se sabe con exactitud la localización física del servicio, los servicios en al nube se distribuyen los recursos del servicio entre distintas maquinas de forma que unidas todas en forman un servicio.
- Las capacidades de un sistema en al nube pueden escalarse rapidamente.
- El servicio esta medido y regulado de forma que se pueda optimizar y su uso pueda ser controlado y/o monitorizado por el consumidor o el proveedor.

Modelos de servicios

- *Servicio software (SaaS, Software as a Service)*, una aplicación software es subida a una infraestructura en la nube y actuá como servicio. El consumidor no puede operar ni manejar nada sobre los medios hardware, pero puede cambiar configuración y características desde la aplicación software.
- *Servicio de plataforma (PaaS, Platform as a Service)*, este tipo de servicio ofrece al usuario o consumidor la posibilidad de desplegar diversas aplicaciones en la nube, de forma que puede controlar todo lo referido a aplicaciones (lenguaje, librerias, servicios, etc) pero no tiene acceso ni control sobre la plataforma (SO, conexión de red, etc.).
- *Servicio de infraestructura (IaaS, Infrastructure as a Service)*, permite desplegar servidores en la nube, de forma que el consumidor no tiene control sobre la infraestructura (hardware) del servicio pero si sobre todo lo demas, de forma que peude montar redes, cambiar los sistemas operativos, etc.

Modelos de despliegue

- *Nube privada*, el control de la infraestructura es privado, solo lo puede gestionar la organización propietaria o un tercero que sea contratado para dicho fin.
- Nube de comunidad, el control de la infraestructura es gestionado por una comunidad de consumidores.

- Nube publica, la infraestructura es gestionada de forma publica, es decir, es gestionada por una organización publica como una universidad, un gobierno, etc.
- Nube híbrida, la infraestructura es el resultado de la mezcla de dos de los anteriores modelos como por ejemplo una infraestructura publica llevada por la comunidad de consumidores.

3.2.2 Backend

El backend de nuestro sistema esta formado por:

1. Un elemento de persistencia, nuestra BBDD no SQL en MongoDB
2. La lógica de dominio, el elemento central que provee el servicio, nuestro sistema generador de fuentes programado en Keras
3. La lógica del servidor, es decir, la comunicación programada en Flask diseñando un servidor REST con una API REST

Todos estos componentes, y por lo tanto, todo el backend del sistema esta programado usando el lenguaje de programación Python.

Python es un lenguaje de programación interpretado, lo que significa que no es compilado, si no que es ejecutado mediante scripts sin guardar el resultado de la interpretación (lectura del código y traducción a lenguaje maquina).

El lenguaje Python tiene una sintaxis sencilla de forma que es un lenguaje muy facil de aprender y muy rápido de programar.

MongoDB y las bases de datos no SQL

Una base de datos no SQL es una base de datos no relacional que, por lo tanto, no usa el lenguaje SQL en sus consultas y accesos. Esto es debido a que esa tecnología se vuelve ineficiente cuando la aplicación escala o necesita mas agilidad en sus consultas.

Las bases de datos no SQL pueden ser de varios tipos, dependiendo del paradigma que usen para suplir el SQL en sus consultas, esto nos da pie a 4 tipos principales:

1. Bases de datos documentales, empareja una clave a una estructura de datos llamada documento, estos documentos pueden contener tuplas clave-valor, tuplas clave-array, o incluso documentos anidados.
2. Almacenamiento en grafos, son usadas para almacenar información de redes de datos
3. Almacenamiento clave-valor, son las más simples ya que cada elemento en al base de datos es guardado como un atributo con clave junto a su valor.
4. Alamacenami9ento en columnas anchas, son BBDD que almacenan datos en forma de columnas en vez de en filas.

En nuestro proyecto se usa MongoDB que es una base de datos no SQL del tipo documental, sus documentos son de la forma de un JSON [6], de forma que los campos pueden cambiar entre documentos y la estructura de losa documentos puede cambiar en el tiempo.

MongoDB usa consultas con forma de JSON, donde sus documentos están distribuidos, es decir, MongoDB es una base de datos en la nube, es decir, se usa vía servidor (para usarlo debes iniciar un programa que actúa de servidor) de esta forma puedes distribuir tus bases de datos de forma geográfica.

Ademas MongoDB esta integrado con una consola propia (puedes usarlo sin necesidad de un lenguaje de programación), peor también esta diseñado para ser usado en varios lenguajes de programación, como Python, Java, C++, C# y Javascript.

Keras y el aprendizaje

En la primera mitad de este capitulo hemos explicado los fundamentos teóricos de este TFG (Autoencoders variacionales) peor su implementación en Python es una tarea técnica que precisa de tecnologías que permitan su fácil implementación en el lenguaje Python.

Keras es una herramienta que permite implementar en Python con una API sencilla redes neuronales, de forma que puedas crear programas en Python que sean capaces de implementar cualquier función referida a redes neuronales (entrenamiento, compilación, exportación, importación, etc.). Esto es gracias a su backend propio basado en Theano y TensorFlow, que es usado mediante funciones simples en Keras.

El aprendizaje por lo tanto también es implementado para que se cargado por Keras, Keras solo necesita listas, una para los datos de entrenamiento y otra para los datos de validación, dependiendo del modelo que use tu red neuronal (aprendizaje supervisado, no supervisado, etc.) necesitaras mas o menos listas.

Por lo que el problema para el aprendizaje pasa por dos puntos, crear los datasets o conjuntos de datos validos para el entrenamiento, y asegurarse que esos dataset son validos.

Un dataset es un conjunto de datos, aplicados a redes neuronales, son el conjunto de datos en el que basaremos el aprendizaje, un data set es eso, grandes cantidades de datos.

Un dataset es valido dependiendo del modelo y de lo que el propio ingeniero del conocimiento que lo crea y gestiona considere valido. Estos criterios son por ejemplo:

- Variedad de datos en el conjuntos
- Cantidad de datos de cada variedad suficiente
- Inexistencia de ruido (datos no validos o datos erróneos)

Para crear un dataset valido simplemente hay que crear un dataset y adaptarlo a nuestra necesidades o crearlo directamente ya valido. Un dataset se adapta de varias formas:

- Eliminando ruido (en nuestro caso fuentes basadas en símbolos)
- Generando los datos de la forma que precisa la red
- Buscando fuentes de datos valida para poder generar la suficiente cantidad de datos fiables
- En caso de la no existencia de fuentes para generar suficientes datos, transformar los datos para que sean datos nuevos, como por ejemplo, girar una imagen x grados de la original.

El aprendizaje también consta de parámetros que son ajustables y que en función de ellos cambiara el aprendizaje de la red, nosotros hablaremos de dos de esos parámetros:

1. Batch size, es el tamaño de los grupos por los que se entrena la red, es decir, si tenemos un dataset de 6000 elementos, y el batch size es de 100, la red se entrenara con grupos de 100 en 100 hasta completar los 6000 datos.
2. Numero de epochs o épocas, es el numero de vueltas que queremos que le de la red al dataset para entrenarse, es decir, si el numero de épocas es 100, la red neuronal se entrenara 100 veces con su dataset.

Juntos estos parámetros podemos entrenar la red neuronal tantas veces en grupos de X tamaño y de esa forma transformar el aprendizaje hasta que los resultados del aprendizaje sean validos (nos sirvan para nuestro propósito, generar fuentes legibles en nuestro caso).

Flask y el uso de API REST como servicio

Flask es un microframework de Python que nos permite construir servicios de Internet de forma simple y rápida, el micro significa que nuestro servicio puede caber entero en un solo archivo Python [7]. Esto significa que el núcleo del framework es simple pero muy escalable.

Esta librería de Python tiene una característica que nos interesa por encima de las demás, su sencillez a la hora de construir APIs .

Una API es una forma de comunicación vía programación con el servicio web, es decir, es una interfaz con el servidor, una API consta de funciones que son ejecutables o invocables por un programa que conecta con el servicio por medio de los distintos protocolos del WWW (World Wide Web). Al usar estas funciones el programa que las usa se comunica con el servidor.

La palabra REST nos indica cadencia de estado, es decir, que el estado del servicio web (entendiendo estado como estado del programa, es decir, el valor en el que se encuentran las variables del programa en ese momento) no cambia.

Al crear y/o diseñar un servidor API REST lo que hacemos es crear una API REST que funciona como servicio, es decir, un servidor que solo responde a las peticiones de sus funciones.

Las funciones de un API REST pueden ser accedidas mediante peticiones HTML y URL que contenga la dirección de dichas peticiones, de esta forma creamos un servicio web basado e una API REST con Flask.

Las APIs REST contienen las siguientes características:

- Identificación de recursos: Los recursos deben ser totalmente accesibles, es decir deben poseer un URI de identificación única
- Manipulación de recursos: Uso de recursos HTTP a partir del acceso gracias a sus métodos Get, Post, Put, Delete, Patch. Aunque en los ultimos años solo se usan GET y POST

- Metadatos para describir nuevos recursos: Usa métodos o tecnologías estándar como HTTP, URI, XML, JSON, etc.
- Comunicación sin estado: no guardas las transacciones entre el cliente y el servidor. Tiene como finalidad disminuir el tiempo de respuesta de invocación al servicio web.

3.2.3 Frontend

Nuestro frontend es una aplicación Android que sirve como ejemplo para utilizar y visualizar las funciones del servidor, es decir, las funciones de nuestro backend, que contiene el generador automático de fuentes (sujeto de estudio y trabajo del presente proyecto).

La aplicación Android, como veremos mas adelante, consiste en un modelo vista-controlador con la persistencia basada en Tareas asíncronas para conectar al servidor.

Android como lenguaje de programación

Android es un SO movil basado en un kernel linux y que incluye un middleware y aplicaciones clave. Pero Android también es un lenguaje de programación ene l cual se crean las aplicaciones de este dispositivo, este lenguaje, basado en XML y Java (también han añadido el lenguaje Kotlin como sustitutivo de Java).

Toda aplicación Java consta de una biblioteca de elemento como imágenes, valores (colores, strings, etc.) guardados en archivos XML y una estructura de programa basad en clases Java.

Android y las conexiones con un servidor API REST

Una particularidad de Android es que impide realizar conexiones a Internet desde el hilo principal de la aplicación, esto se traduce en que para realizar una peticion HTTP (como es nuestro caso) debemos crear un hilo aparte que realice la petición y reciba el resultado para posterior tratamiento del mismo.

Para este tipo de tareas Android trae una herramienta llama tareas asíncronas (AsyncTask) , esta herramienta no es mas que una clase en java que sirve de plantilla para crear tareas asíncronas propias que funcionan como hilos aparte del principal.

REFERENCIAS

[1] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

[2] Ian Goodfellow,Yoshua Bengio and Aaron Courville, *Deep Learning*, Parte 1:Machine Learning Basics, MIT Press, 2016, http://www.deeplearningbook.org/contents/ml.html

[3] Chollet F. Building Autoencoders in Keras. Sab. 14 Mayo 2016. https://blog.keras.io/building-autoencoders-in-keras.html

[4] ABDI, Hervé; WILLIAMS, Lynne J. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2010, vol. 2, no 4, p. 433-459.
https://pdfs.semanticscholar.org/53b9/966a0333c9c9198cdf03efc073e991647c12.pdf

[5] MELL, Peter, et al. The NIST definition of cloud computing. 2011
http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf

[6] w3schools, JSON – Introduction https://www.w3schools.com/js/js_json_intro.asp

[7] Flask, Foreword, 2010. http://flask.pocoo.org/docs/1.0/foreword/#what-does-micro-mean

FUENTES

DEEP_LEARNING
https://www.investopedia.com/terms/d/deep-learning.asp
https://rubenlopezg.wordpress.com/2014/05/07/que-es-y-como-funciona-deep-learning/
http://www.deeplearningbook.org/
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.4088&rep=rep1&type=pdf
http://publications.lib.chalmers.se/records/fulltext/248445/248445.pdf

AUTOENCODERS
http://www.deeplearningbook.org/contents/autoencoders.html
https://ermongroup.github.io/cs228-notes/extras/vae/
https://arxiv.org/pdf/1606.05908.pdf
https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/
http://alexadam.ca/ml/2017/05/05/keras-vae.html
http://dohmatob.github.io/research/2016/10/22/VAE.html
http://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and.html
https://blog.keras.io/building-autoencoders-in-keras.html

VAE
https://blog.keras.io/building-autoencoders-in-keras.html
https://arxiv.org/pdf/1606.05908.pdf
http://papers.nips.cc/paper/6275-ladder-variational-autoencoders.pdf
https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/
https://ermongroup.github.io/cs228-notes/extras/vae/
http://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and.html

CLOUD COMPUTING http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf

MONGODB https://www.mongodb.com/nosql-explained
https://www.mongodb.com/what-is-mongodb

KERAS https://keras.io/

FLASK http://flask.pocoo.org/