UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

Framework web de uso de sistemas de machine learning.

Presentado por Javier Martínez Riberas
en Universidad de Burgos — 3 de julio de 2017

Tutores: Dr. José Francisco Díez Pastor
Dr. César Ignacio García Osorio

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

D. José Francisco Díez Pastor y D. César Ignacio García Osorio, profesores del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Exponen:

Que el alumno D. Javier Martínez Riberas, con DNI 71299495R, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado "Framework web de uso de sistemas de machine learning.".

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección de los que suscriben, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 3 de julio de 2017

Vº. Bº. del Tutor:	Vº. Bº. del co-tutor:
D. José Francisco Díez Pastor	D. César Ignacio García Osorio

Resumen

Recientemente, la mayoría de empresas que usan productos informáticos están intentando mejorar sus servicios con un enfoque 'AI first', esto es más prevalente en los grandes del sector que tratan de aplicar el aprendizaje automático de manera más amplia.

La escalabilidad es un concepto con una definición clara pero muy abstracta [13], la mejor definición que se ha encontrado se atribuye a Bondi [3] y la define como la capacidad de un sistema para ser capaz de ampliarse para manejar mayor cantidad de trabajo.

Desde el punto de vista de mantenimiento, un sistema escalable es aquel al que podemos mantener o incrementar su funcionalidad sin incurrir en una cantidad de deuda técnica demasiado grande, permitiendo un desarrollo más rápido.

La escalabilidad es una propiedad compleja en el mundo software ya que existen varias fuentes de ampliación del sistema, la más común es la escalabilidad aumentando el número de copias del software que se ejecutan, en el mismo ordenador o en distintos ordenadores. Se puede descomponer el sistema en los subsistemas más pequeños posibles, de manera que la complejidad disminuye convirtiendo el sistema en más fácil de mantener. También existe una última, que es separar los servicios en particiones distintas, de manera que cada partición se encarga de dar servicio a parte de los resultados.

Estas tres opciones de escalado se denominan en el libro *The Art of Scaling* [1] como el '*cubo de la escalabilidad*'.

En este trabajo se busca obtener un sistema escalable en los dos primeros ejes de escalado. El primer eje o dimensión lo vamos a conseguir con contenedores, el sistema que usaremos es docker. La segunda dimensión la obtendremos con la arquitectura de microservicios.

Esto busca facilitar el acceso a la solución de un problema común en el campo del aprendizaje automático que es la clasificación de imágenes. Esto se hace mediante *Tensorflow*.

Con esto se pretende dar un ejemplo de una de las maneras de evitar problemas de escalabilidad, un problema bastante común en el mundo del desarrollo software. Esta escalabilidad es tanto de rendimiento como de mantenimiento. Se busca proporcionar una arquitectura fácilmente escalable en ambos sentidos cuyo reflejo en código no es complejo para el futuro aprendizaje. También se intenta proporcionar un esqueleto que pueda ser usado por los futuros proyectos que necesiten un control básico de usuarios, pudiendo invertir el tiempo liberado en otras partes o temas de mayor interés.

Para ilustrar cómo se resuelve el reto de añadir más servicios, se ha implementado un servicio que usa *Deep Learning* para clasificar imágenes sobre un conjunto de clases, como por ejemplo Imagenet [14].

Descriptores

Mineria de datos, redes neuronales, clasificadores, aplicación web...

Abstract

Recently, most software centered companies are trying to improve their services with an 'AI first' approach, this is more prevalent in large companies trying to apply Marchine Learning more widely.

Scalability is a concept with a clear but very abstract definition [13], the best definition that has been found is attributed to Bondi [3] who defined it as the ability of a system to be able to expand to handle an increasing work load.

From the point of view of maintenance, a scalable system is that can be maintaned or whose functionality can be increased without incurring in a technical debt too large, allowing a faster development.

Scalability is a complex property in the software world as there are several places where a system can be expanded. The most common way of providing scalability is by increasing the number of copies of running software, on the same computer or on different computers. Another approach it to divide the system into the smallest subsystems possible so that the complexity decreases making the system easier to maintain. Finally, it is possible to separate the services in different partitions so that each partition is in charge of providing part of the results.

These three scaling options are called the '*cube of scalability*' in the book *The Art of Scaling* [1].

In this work we seek to scale a system using the first two scaling axes. We will achieve the scaling in the first axis or dimension by using containers, the system we will use is docker. The scaling in the second dimension will be obtained with the microservice architecture.

This seeks to facilitate access to the solution of a common problem in the field of Machine Learning that is the image classification. This is done by using Tensorflow. This way of proceed gives an example of one of the ways to avoid scalability problems, a quite common problem in the world of software development. This scalability is both in performance and maintenance. It seeks to provide an architecture easily scalable in both directions that avoid increasing the complexity of the code to facilitate its future reuse and learning. It also attempts to provide a skeleton so that it can be used by future projects that need to use a basic user control, investing the time released in other parts or topics of greater interest.

To illustrate how the challenge of adding more services is solved, a service that uses Deep Learning has been implemented to classify images on a set of classes, such as Imagenet [14].

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Introducción

El uso mayoritario del aprendizaje automático o *Machine Learning* (ML) se puede ver en empresas como Google que ha pasado de buscar el término exacto pedido por el usuario a intentar, con la información de que disponen, de averiguar en que contexto se está buscando, por ejemplo, si predice que eres programador y buscas R, probablemente los primeros resultados sean del lenguaje de programación.

El desarrollo de este proyecto consiste en dar un servicio de *Machine Learning* por la actualidad y relevancia de la tecnología. El proyecto consistirá en un clasificador que agrupará las imágenes en los conjuntos de Imagenet [14].

Actualmente, la escalabilidad es muy importante en el desarrollo software debido a la necesidad de hacer sistemas cada vez más grandes y complejos que a la vez no consuman recursos de manera excesiva. Esto se puede conseguir con ciertas arquitecturas como los microservicios. Otras arquitecturas que proporcionan este tipo de ventajas son los *Serverless* [12, 15].

Este trabajo se orienta a conseguir un sistema escalable. Se intentará que los métodos para conseguir este sistema sean lo más actuales, prestigiosos y usados, ya que probablemente esos métodos serán fundamentales el día de mañana. También se persigue adquirir los conocimientos necesarios para poder replicar este sistema sobre proyectos ya creados.

El sistema en cuestión será una página web, ya que es un sistema altamente accesible (desde casi cualquier plataforma) con mayor facilidad de mantenimiento que la mayor parte de aplicaciones específicas a un dispositivo concreto. Otra ventaja que proporcionan las páginas web es que tendremos acceso a todos los errores y fallos que surjan en ejecución.

Se usarán tecnologías puntas para conseguir estos objetivos. Como docker, para la escalabilidad y reducción de deuda técnica, gracias a la arquitectura de microservicios que facilita.

La metodología que se usará será la integración continua ya que permite reducir la deuda técnica derivada de la integración de distintos servicios entre sí. Esta consiste básicamente en intentar integrar los servicios que se disponen a cada paso que se da en la creación de software. En proyectos grandes, esto podría ser cada día, y en proyectos pequeños, cada *commit*.

Objetivos del proyecto

A continuación se indican los objetivos, tanto teóricos, marcados por los requisitos, como los objetivos que se persiguen con el proyecto. También se incluyen ciertos requisitos que no tienen por que ser obvios, pero que en la actualidad se esperan de cualquier aplicación web.

2.1. Objetivos principales

- Conseguir un sistema actual y escalable.
- Seguir principios de desarrollo actualizados.
- Proporcionar control de usuarios a aquella persona que lo necesite y su proyecto encaje con el que aquí se muestra.

2.2. Servidor web y página web

- Arquitectura MVC (Model View Controller).
- Facilidad de uso: Que el diseño sea intuitivo y fácil de aprender a usar.
- Internacionalización: Preparar la aplicación para que esté disponible en varios idiomas.
- Sistema responsivo: Que se adecue al dispositivo desde el que se visita.

2.3. Servicio de Minería

- Proporcionar una manera de clasificar imágenes con *Deep Learning*.
- Proporcionar una manera de cambiar las clases en las que se clasifican las imágenes.

2.4. Proceso de desarrollo

Los puntos siguientes se ven actualmente como necesidades fundamentales en cuanto al desarrollo de proyectos a los que quizá no se de suficiente importancia.

- Sistema para el control de dependencias
- Despliegue del proyecto
- Sistema de control de versiones
- Tener un proceso de CI (*Continuous integration*)
- Programar con agilidad

2.5. Objetivos personales

- Aprender arquitecturas actuales, las cuales se pueden usar tanto en industria como en academia.
- Avanzar mis conocimientos a partir de los obtenidos en la carrera, sobre todo aquellos que tienen importancia real y quizá no se hayan estudiado suficientemente en la carrera
- Profundizar en el entorno de Python, ya que en los últimos años se ha incrementado su importancia tanto para *data scientists* como para desarrolladores web, que son las dos profesiones que encuentro muy interesantes.

Conceptos teóricos

Algunos conceptos teóricos tanto de la parte técnica como de la parte de procesos de software

3.1. Deuda técnica

La metáfora de deuda nació como una forma de expresar la diferencia entre el entendimiento del programa (la abstracción) y su implementación [4].

Con el tiempo surgió un término similar la deuda técnica, que es la metáfora que expresa el coste que conlleva el dejar código que no tiene un buen diseño en el proyecto. Si seguimos manteniendo el proyecto, esta deuda cada vez costará más el arreglarla y tendremos que pagar los intereses [7].

Las razones de incremento de la deuda con el paso del tiempo pueden ser el simple hecho de olvidar cómo funciona cada línea de código, o la complejidad emergente del acoplamiento de código antiguo con otro nuevo sin cambiar las abstracciones ni desacoplar el código.

Este tipo de deuda se ve como necesaria, según algunos autores [17], se debe incurrir en deuda técnica, ya que es beneficioso para el desarrollo. Esto se debe a que, a corto plazo, es imposible prever cómo se va a desarrollar el proyecto, y unos cambios demasiado tempranos, pueden ser un mal diseño a medio o largo plazo. La deuda 'buena' o beneficiosa se suele considerar como aquella que como mucho dura una semana, una vez existe por más tiempo pasa a ser cada vez más costosa de solucionar o 'pagar'.

Una característica importante de esta deuda es que una vez dejas de soportar el mantenimiento de un proyecto la deuda técnica desaparece, al contrario que la deuda financiera.

Se puede idealizar la deuda técnica y considerar que todas las semanas en las que se dedica una cantidad de tiempo a solucionarla esta deuda técnica

Figura 3.1: Deuda técnica: Beneficiosa y perjudicial [17].

desaparece, pero esto no corresponde con la realidad. Normalmente, la deuda, aunque se solucione, suele ir incrementando poco a poco, debido a la complejidad de un sistema mantenido a lo largo de una gran cantidad de tiempo.

La manera de tener esto en cuenta es tener un 'techo de deuda'. Este techo debería ser lo suficientemente alto como para que no se alcance todos los meses, pero no demasiado alto como para que cuando se llegue al techo fijado, el proyecto sea directamente un fracaso o demasiado caro de pagar.

La opinión de algunas personas es que se debe llegar al techo cada 6 meses, desde mi propia inexperiencia, creo que deberíamos evaluar la deuda a los 4 meses, para ver si la re-estructuración del proyecto es viable e intentar buscar un punto de tiempo en el que el pagar la deuda técnica sea algo más simple, ya que si esperamos hasta los 6 meses, probablemente no podamos elegir un momento óptimo.

Es importante destacar que existen otros argumentos para mantener la deuda técnica baja: al que menos importancia parece darse es que la deuda técnica es algo difícilmente cuantificable hasta que se refactoriza, si no la reducimos, no sabemos cuanta tenemos.

3.2. Integración continua (*Continuous Integration*)

Normalmente acortado con las siglas CI, es un método habitual para reducir la deuda técnica. Es la práctica de ejecutar y testear conjuntamente todos los servicios que deban ir relacionados una vez se inicie el despliegue en producción. Esto nos asegura que van a funcionar en producción la mayoría de las veces,

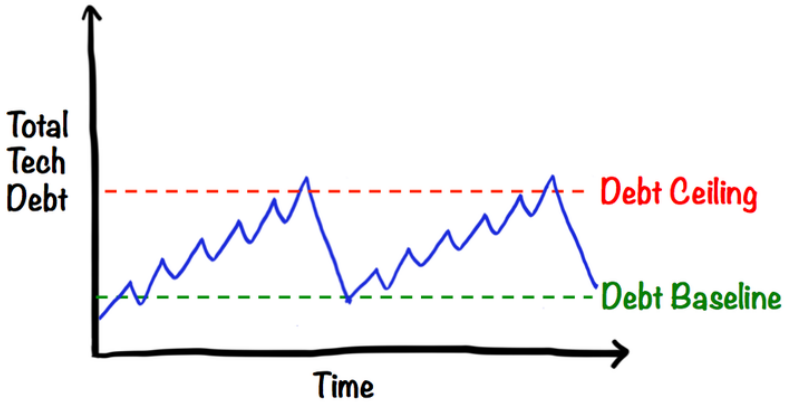

Figura 3.2: Deuda técnica: Techo y base [17]

unas pocas veces habrá problemas por la escalabilidad, errores dado el cambio del hardware, problemas con el rendimiento... Un ejemplo es si ejecutamos Java y los test no fallan, pero al ponerlo en producción le damos a la JVM más de 200 GB de memoria RAM, esto causa comportamientos inesperados.

Entrega continua (*Continuous Delivery*)

Es un término asociado a CI, consiste en un proyecto que dispone de CI y buena cantidad de test de calidad, una vez ya tengamos un sistema a punto podemos empezar a modificar o mejorar funcionalidades añadiéndolas directamente a producción si pasa los test, requiere que hagamos los test casi a la vez que el código. Esto fomenta y premia técnicas como *extreme programming* (XP) que se basan en TDD (*Test Driven Development*), una forma de programar que pide que hagamos los test antes que el resto del código.

3.3. DevOps

DevOps es un acrónimo inglés de: '*software* ***Dev****elopment and information technology* ***Op****erations****s***' es un término que engloba un conjunto de prácticas de colaboración y comunicación entre desarrolladores software y técnicos informáticos. Los objetivos de esta comunicación y colaboración son una construcción de software más consistente y confiable. Este proceso heredero de las técnicas ágiles se basa en una *cadena de herramientas*. Esta cadena de herramientas es algo que no está completamente definido, pero más o menos la podemos concretar. Cabe tener en cuenta que esta cadena cambia según a quien le preguntes.

La '*cadena de herramientas*' de DevOps

Esta cadena de herramientas se basa en siete procesos con sus correspondientes herramientas:

1. **Plan**: Consistente en determinación de métricas, requerimientos... y una vez pasemos de la primera iteración ha de tener en cuenta el feedback del cliente.
2. **Creación**: Es el proceso de programar y crear el software, la herramienta en este proceso es el software de control de versiones que vayamos a usar.
3. **Verificación**: Proceso de comprobación de la calidad del software. Normalmente consiste en hacer test de diversos tipos (aceptación, seguridad...)
4. **Preproducción** o **empaquetación**: En esta fase se piden aprobaciones de los distintos equipos y se configura el paquete.
5. **Lanzamiento**: En este punto se prepara el horario de lanzamiento y se orquesta el software para poder ponerlo en el entorno de producción objetivo.
6. **Configuración**: Una vez el software está desplegado toda la parte de la infraestructura y configuración de la misma se incluye en esta categoría, como por ejemplo las bases de datos, configuración de las mismas...
7. **Monitorización**: Tras entregar el software, se mide su rendimiento en la infraestructura objetivo y se mide la satisfacción del usuario final. Se recogen métricas y estadísticas

3.4. Microservicios

Los microservicios son una arquitectura y un patrón de diseño, que no se ha inventado en un momento concreto, si no que ha surgido como una tendencia o patrón del diseño de sistemas en el mundo real.

Un microservicio es un servicio pequeño y autónomo que puede trabajar junto a otros, es importante que tenga un objetivo claro y que lo haga bien.

Esto está reforzado por el concepto del «*Principio de responsabilidad única*» de C. Martin [20] «*Juntar aquellas cosas que cambian por la misma razón y separar aquellas que cambian por razones diferentes*».

El tamaño de un microservicio es algo muy discutido, pero generalmente se requiere que sea mantenible por un equipo 'pequeño' (6-10 personas), y se pueda reescribir en 2 semanas.

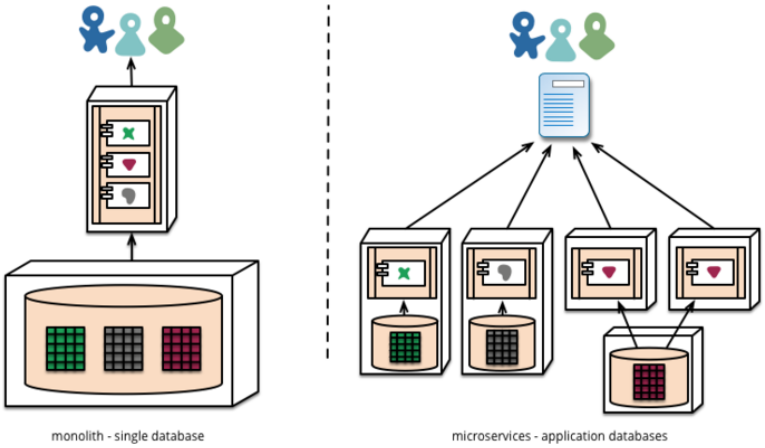

Figura 3.3: Arquitecturas monolítica y microservicios. [9].

Esto se debe conseguir mediante una interfaz de programación de aplicación (API: *Application Programming Interface*), que permita a los clientes acceder al servicio sin causar acoplamiento, algo más difícil de hacer que de decir.

Los principales beneficios son:

- **Sistemas heterogéneos**: Facilita usar distintas tecnologías en distintos sistemas, para usar la mejor herramienta en cada ocasión.
- **Resiliencia**: Si falla un microservicio, el resto del sistema se puede mantener levantado, aislando los problemas y facilitando la alta disponibilidad.
- **Escalabilidad**: En caso de que una parte del sistema necesite mayor número de recursos, podemos crear una nueva instancia sin cambiar el funcionamiento del resto del sistema.
- Facilidad de despliegue: Los cambios se mantienen más contenidos.
- **Replazabilidad**: Un microservicio que pasa a ser obsoleto puede ser reemplazado o eliminado, en vez de tener que mantenerlo porque al eliminarlo se rompe todo el sistema, algo que no debería pasar, pero que es común en los sistemas legados.
- **Test a nivel de servicio**: Cada microservicio se puede testear por su cuenta propia, de manera que aunque el sistema aumenten su complejidad, podamos controlar cada microservicio por separado.

Estos beneficios por supuesto vienen con desventajas:

- **Distribución**: Raramente se van a ver microservicios en una arquitectura que no esté distribuida, esto hace que sea más difícil de programar y tengamos que tener en cuenta otros errores.
- **Seguridad**: Es más difícil asegurar la seguridad de un microservicio, ya que debemos protegernos contra más puntos de vulnerabilidad, a no ser que estemos ejecutando en un clúster seguro.
- **Complejidad de operación**: Hace falta un equipo de operaciones con madurez, de manera que se puedan redistribuir los microsistemas regularmente.
- **Test a nivel de sistema**: Los test a nivel de sistema incrementan algo su complejidad, ya que vamos a tener que usar microservicios falsos, como mocks o similares casi siempre, a no ser que estemos en un sistema pequeño.

Por último, vale la pena hablar del coste en productividad, esto no es una ventaja ni un perjuicio, es un *trade-off*, un intercambio.

En un sistema monolítico, tenemos menos coste en productividad, hasta que llegamos a un punto crítico en el que la complejidad aumenta demasiado o necesitamos escalar.

En los microservicios, tenemos menos coste a la hora de seguir iterando aumentando la complejidad de un sistema, pero a cambio tenemos mayor barrera de entrada, hasta que se tiene el primer prototipo funcional y los desarrolladores tienen que aprender cómo programar bajo esta arquitectura

3.5. Deep Learning

Definido estrictamente el *deep learning* consiste en el uso de redes neuronales artificiales con más de una capa oculta.

Redes neuronales artificiales

Las redes neuronales son sistemas de computación inspirados por las redes neuronales biológicas, que podemos encontrar en los cerebros de los animales. Estos sistemas requieren un tiempo de 'aprendizaje' (entrenamiento o mejora progresiva del rendimiento) para, a partir de unos ejemplos, resolver una tarea.

Hoy en día, las redes neuronales artificiales se forman a partir de un grafo acíclico dirigido, y se organizan típicamente en capas. Las capas más comunes son:

- La capa de entrada: donde la red recibe los datos.

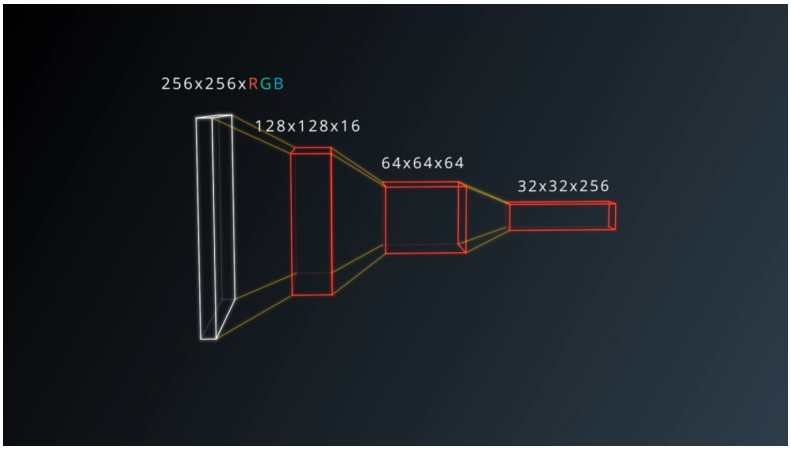

Figura 3.4: Red neuronal convolucional: imagen de Udacity: https://www.udacity.com/course/deep-learning–ud730

- Las capas ocultas: que son las que 'aprenden' mediante ajustes a sus propiedades.
- La capa de salida: que devuelve la predicción o resultado deseado.

Las propiedades que tienen las capas intermedias son:

- La función de activación: esta es la función que determina si la neurona se activa (pasa información a otras neuronas) o no. En algunos modelos siempre pasan información y esta función simplemente la modifica. Las más usadas son:
- Los pesos: son la parte que cambia al entrenar el modelo, de manera que afectará al resultado.

La forma de cambiar los pesos y de que la red aprenda se llama propagación hacia atrás (*backprop* o *backpropagation*), que calcula el gradiente de la función de perdida (diferencia entre el valor y el objetivo).

Redes neuronales convolucionales

Las redes neuronales convolucionales se distinguen de las redes neuronales tradicionales en las capas que las componen, capas convolucionales.

La convolución es una operación matemática diseñada para imitar el estímulo producido en una neurona de la corteza cerebral en respuesta a una imagen.

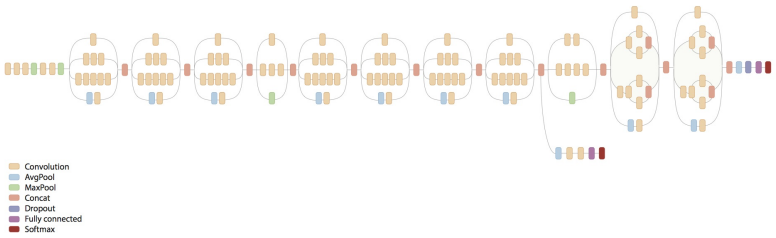

Figura 3.5: Arquitectura inception v3: imagen de Tensorflow: https://github.com/tensorflow/models/tree/master/inception

La potencia de la convolución se presenta a la hora de pasar una ventana por distintas partes de la entrada con los mismos pesos, aprendiendo así patrones más complejos.

La primera red neuronal convolucional ampliamente reconocida es LeNet-5 por Yann LeCunn [19].

A partir de esta se desarrollan otras, como por ejemplo, AlexNet [18] e *inception* [24].

En este trabajo se presenta el uso y re-entrenamiento de *inception* v3.

Cabe destacar que *inception* presenta una diferencia importante respecto a las arquitecturas anteriores, en vez de elegir entre una convolución 1x1, 3x3, 5x5 o un *pooling*[1], los módulos *inception* hacen todas las operaciones y las combinan en el resultado.

[1] *Pooling*: operación que consiste en devolver el valor más alto/medio/más bajo de la ventana a la que se aplica.

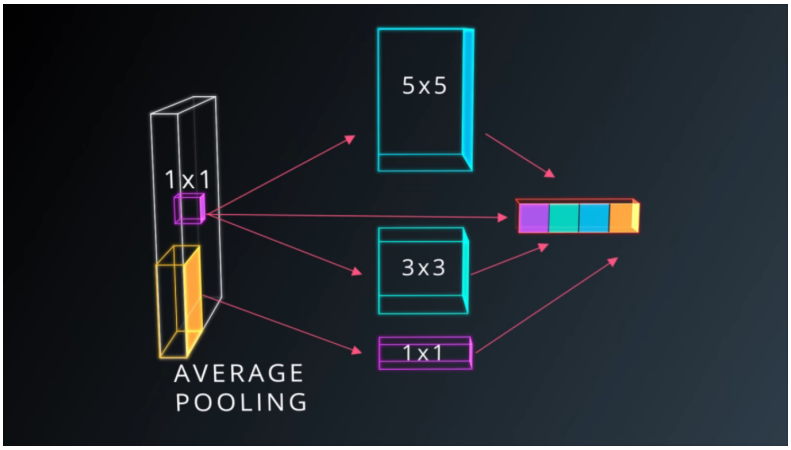

Figura 3.6: Modulo inception: imagen de Udacity: https://www.udacity.com/course/deep-learning–ud730

Técnicas y herramientas

En esta parte de la memoria se detallan tanto técnicas y herramientas, como metodologías y bibliotecas usadas.

4.1. Metodología

Al no haber una metodología concreta que se adapte al trabajo de una sola persona, se ha buscado un conjunto de metodologías que se pudiesen combinar para dar una metodología conjunta más aplicable a la situación particular que se tiene.

De **DevOps** se ha extraído la *toolchain* (cadena de herramientas) para minimizar riesgos totales del proyecto y se ha buscado reducir la deuda técnica nacida de la necesidad de desplegar el producto y no estar preparado para ello, algo que resulta ser más habitual de lo que debería ser.

De **Scrum** recogemos las herramientas como los *sprint* y las reuniones tras cada *sprint*, tanto para concretar el siguiente *sprint*, como para hacer una retrospectiva del anterior.

Se ha estudiado también **extreme programming** (XP) y se ha visto que no es viable para el proyecto por la cantidad de esfuerzo que requiere para una sola persona hacer el trabajo de dos como es el **pair programming**.

De todos los métodos ágiles se ha adquirido la mentalidad de no intentar planificar todo desde el principio e intentar planear las cosas con el conocimiento adquirido en cada iteración. Se intenta decidir en todo lo más tarde posible, para intentar conseguir el máximo conocimiento posible, para mejorar las decisiones.

Se ha decidido que la recopilación de información de los *sprints* y este tipo de documentación se va a hacer en un tablero **kanban**, la herramienta que nos

proporciona este sistema va a ser **ZenHub**[2], se usa por la facilidad y el nivel de integración que nos da con otros sistemas también elegidos.

4.2. Patrones de diseño

Se ha intentado usar los patrones de diseño para conseguir una arquitectura mejor y más simple para nuevos desarrolladores. También hay que tener en cuenta que no todos los patrones tienen sentido en un lenguaje de programación como Python, por su naturaleza dinámica.

Model-View-Controller

El patrón modelo vista controlador nos ayuda a simplificar mucho la estructura de los archivos de una página web o aplicación que dependa de vistas. Esto es especialmente beneficioso cuando se introduzca a un miembro nuevo al proyecto, ya que permite conocer rápidamente como funciona mucha cantidad de código, de manera que si necesitase realizar cambios, lo tuviese fácil.

4.3. Control de versiones

El control de versiones es una necesidad hoy en día tanto en sistemas compuestos de un desarrollador, como los compuestos de múltiples desarrolladores, esto nos permite poder dejar sin terminar ciertas funcionalidades que empecemos a implementar y se identifiquen como muy complejas, para seguir con otras que den más valor al cliente y luego continuar con las complejas o dejarlas de lado según beneficie o no al cliente.

Este solo es un ejemplo de las ventajas, entre las que se cuentan también otras como: recuperación de versiones estables, visualización de los cambios que han dado resultado a un defecto de programación...

El sistema de *hosting* lo usaremos para facilitar el acceso a la información contenida en el sistema, también, al ser externo (no estar en el mismo ordenador que se use) nos servirá de sistema de copias de seguridad.

Usaremos **Git**[3] por razones de documentación y conocimiento ya adquirido, aunque existen otros sistemas como Subversion[4].

Las alternativas principales de sistema de hospedaje de Git son: Bitbucket[5], GitHub[6] y Gitlab[7].

[2] ZenHub: https://www.zenhub.com/.
[3] Git: https://git-scm.com/.
[4] Subversion: https://subversion.apache.org/.
[5] Bitbucket: https://bitbucket.org.
[6] GitHub: https://github.com.
[7] Gitlab: https://gitlab.com.

Se ha elegido GitHub por una mezcla de razones históricas con razones de integración con otros sistemas como el sistema kanban (ZenHub).

Previamente se conocía GitHub y se sabe que los proyectos *open source* no tienen ninguna limitación, además GitHub ofrece fácil integración con otros sistemas como **Travis**[8] y **Slack**[9], si hiciese falta.

4.4. Integración continua

La integración continua es el método por el cuál intentamos integrar todos nuestros productos continuamente para ver si funcionan correctamente en conjunto. Nos va a ayudar a minimizar el riesgo de fallo del proyecto, sobre todo si se mantiene en desarrollo un largo periodo de tiempo.

La herramienta que se va a usar es **Travis**, esto se debe a la gran documentación, facilidad de integración con Git y GitHub, y otras integraciones que nos ayudaran con otras secciones. Otra ventaja es que, al ejecutar tu *build* y test en sus servidores, no te tienes que preocupar de casi nada.

Otras opciones que se podrían usar son Jenkins[10] que como ventaja es *open source* y tiene gran cantidad de *plugins*. La desventaja principal es que requiere de más trabajo por nuestra parte.

4.5. Quality Assurance

Dado que no tenemos departamento de QA, como deberíamos, para llevar una *toolchain* como la especificada en DevOps, usaremos herramientas automatizadas, que aunque no tengan la calidad de una revisión humana, es lo mejor de que disponemos.

La herramienta que se ha elegido es **CodeClimate**[11], se ha usado por que no ha sido demasiado complicado el incluir esta herramienta en la línea de producción que ya teníamos (Github+Travis). Se ha intentado usar SonarQube[12], pero no ha sido tan sencillo (se intento introducir dentro de la *toolchain* automática, no manualmente).

[8] Travis: https://travis-ci.org/.
[9] Slack: https://slack.com/.
[10] Jenkins: https://jenkins.io/.
[11] CodeClimate: https://codeclimate.com/.
[12] SonarQube: https://sonarqube.com/.

4.6. Test

Los test se han realizado con **pytest**[13]. Esta herramienta es muy parecida a unittest[14] (la herramienta de test en la biblioteca standard de Python). Tiene ventajas, facilita el depurado al decirte exactamente qué ha fallado y con qué valores, y los métodos como *setup* y *teardown*. Se integró esta herramienta con Travis.

Otras bibliotecas pueden ser unittest o nose2[15].

Recubrimiento

La forma más sencilla para ver el recubrimiento de un test en Python es Coverage.py[16], tiene varias opciones como *report* para salida en consola o html para un html con el cubrimiento bien señalado. Se consiguió integrar esta herramienta con Travis, pytest y CodeClimate de manera que se ejecuta con la integración continua.

4.7. Dependencias

Para el control de dependencias (seguridad, últimas versiones y licencias) se ha usado VersionEye[17], integrado con GitHub, esta herramienta se adhiere mediante un *webhook* a GitHub y nos dice si se ha descubierto alguna brecha de seguridad en nuestras herramientas, cuáles son sus últimas versiones, si las estamos usando, o si las licencias del proyecto son compatibles con la que tenemos.

4.8. Comunicación

La comunicación se ha hecho de diferentes maneras: correo electrónico, Slack, *issues* de GitHub y de manera física. El correo electrónico se ha usado para casi toda la comunicación a distancia. Slack se ha usado para integrar herramientas y notificar sobre estas. La comunicación física es el medio que más se ha usado, esto no se debe a ningún motivo en particular, simplemente ha sido el medio más natural para todos.

[13] pytest: https://docs.pytest.org/en/latest/.
[14] unittest: https://docs.python.org/3/library/unittest.html.
[15] nose2: http://nose2.readthedocs.io/en/latest/index.html.
[16] Coverage.py: https://coverage.readthedocs.io/en/coverage-4.4.1/.
[17] VersionEye: https://www.versioneye.com/.

4.9. Documentación

La documentación se ha basado en dos sistemas: en código y fuera de código.

En código

En el código se ha usado la documentación recomendada por la comunidad de python en el PEP 287 [11], este recomienda usar **reStructuredText**[18] (rst).

Se han seguido las *guidelines* (guías) de un equipo de programadores de la comunidad de Python: pocoo [16], esto es para poder usar Sphinx[19] (generador de documentación) para transformar esas cadenas de documentación en una documentación tanto en html o en LaTeX si hiciese falta.

Fuera de código

Para la documentación que se está leyendo se ha usado la plantilla propuesta por el tribunal de evaluación de TFG para crear documentos LaTeX aunque se ha considerado usar rst al igual que en el código, ya que si hiciese falta, se podría transformar a LaTeX o a html, siendo mucho más flexible. No se ha hecho por que no existe una plantilla y la barrera de entrada es algo alta.

4.10. Editor de texto

Se ha mirado tanto en editores de texto más simples como Vim[20], Atom[21], Sublime Text[22]... Estos cuentan con suficientes *plugins* que acaban siendo una IDE.

Las IDEs también se han mirado y se han considerado tanto Spyder[23], LiClipse[24] como PyCharm[25].

De todos estos no hay ventajas en usar unos u otros, ya que todos acaban teniendo la misma capacidad gracias a *plugins*, se ha decidido usar PyCharm por que era la única IDE que todavía no se había probado.

[18] Es un lenguaje de marcado con indicaciones intuitivas para marcar la estructura de un documento ver [10]
[19] Sphinx: http://www.sphinx-doc.org/en/stable/index.html.
[20] Vim: http://www.vim.org/.
[21] Atom: https://atom.io/.
[22] Sublime Text: https://www.sublimetext.com/.
[23] Spyder: https://github.com/spyder-ide/spyder.
[24] LiClipse: https://www.liclipse.com/.
[25] PyCharm: https://www.jetbrains.com/pycharm/.

4.11. Bibliotecas

Se ha usado Flask[26] como *microframework*, ya que sirve tanto para implementar el patrón MVC, como para crear un sistema altamente escalable.

Para facilitar ciertas funcionalidades como control de usuarios, seguridad... se han usado las extensiones: flask-babel[27] y babel[28] (internacionalización), flask-login[29] (control de usuarios), flask-oauthlib[30] y oauthlib[31] (uso de oauth simplificado), flask-sqlalchemy[32], sqlalchemy[33] y psycopg2[34] (acceso a la base de datos), flask-wtf[35] y WTForms[36] (Formularios html) y flask-bcrypt[37] y Bcrypt[38] para encriptación.

Como podemos ver, muchas de estas extensiones de flask tienen el mismo nombre pero con flask añadido, esto se debe a que son envoltorios (también llamados *wrappers*) de la biblioteca en cuestión, pero adaptados para inicializarse con Flask (generalmente añaden un constructor y un constructor *lazy*).

Se ha usado el módulo Paramiko[39] para simplificar el control de errores de SSH.

Tensorflow

Tensorflow™[40] es la biblioteca de *machine learning* más popular[41] en Python. Es una biblioteca *Open Source* que permite computación numérica a partir de grafos.

El nombre de *tensor* se le da ya que los vectores de datos que se pasan entre operaciones tienen este nombre. Las operaciones se representan como nodos en el grafo. Tiene compatibilidad con GPUs y esta preparado para poder ejecutarse en sistemas distribuidos.

Originalmente la biblioteca fue desarrollada por Google. Es importante destacar que aunque el modelado y entrenamiento esté pensado para hacerse

[26] Flask: http://flask.pocoo.org/.
[27] flask-babel: https://github.com/python-babel/flask-babel.
[28] babel: http://babel.pocoo.org/en/latest/.
[29] flask-login: https://flask-login.readthedocs.io/en/latest/.
[30] flask-oauthlib: https://flask-oauthlib.readthedocs.io/en/latest/.
[31] oauthlib: https://oauthlib.readthedocs.io/en/latest/index.html.
[32] flask-sqlalchemy: http://flask-sqlalchemy.pocoo.org/2.1/.
[33] sqlalchemy: https://www.sqlalchemy.org/.
[34] psycopg2: http://initd.org/psycopg/.
[35] flask-wtf: https://flask-wtf.readthedocs.io/en/stable/.
[36] WTForms: https://wtforms.readthedocs.io/en/latest/.
[37] flask-bcrypt: https://flask-bcrypt.readthedocs.io/en/latest/.
[38] Bcrypt: https://pypi.python.org/pypi/bcrypt/3.1.0.
[39] Paramiko: http://www.paramiko.org/.
[40] Tensorflow™: https://www.tensorflow.org/.
[41] La metrica de popularidad fue el número de '*stars*' en GitHub.

en Python, se pueden usar los modelos una vez entrenados en gran variedad de lenguajes de programación, entre los que se incluyen C++, Go, Java...

Como curiosidad, la forma de programar los modelos es parecida a la funcional. Se declara el orden de las capas u operaciones y se ejecutan pasando una entrada y pidiendo el resultado de una o varias capas, lo cual devolverá una serie de tensores.

Aspectos relevantes del desarrollo del proyecto

5.1. Idea e inicio

Se eligió un proyecto con relación con el *deep learning*, ya que existía un interés previo, tanto a nivel personal, como a nivel profesional. La gran versatilidad de este tipo de redes, ya sea en clasificación, con los modelos profundos, compresión, con los *autoencoders*, o incluso generación de imágenes, con las revolucionarias GAN, está claramente demostrada, ahora queda buscarles nuevos usos que supongan una aplicación práctica más directa y útil.

A nivel personal, la importancia que tienen las redes neuronales en el mundo actual, hizo que surgiese una gran curiosidad. Un ejemplo de la aplicación de este tipo de tecnología podrían ser los coches automáticos o aplicaciones surgidas últimamente, como la viral durante un par de días FaceApp.

A nivel profesional, el *deep learning* se ha posicionado como uno de los algoritmos que mejores resultados ha dado en muchos campos, esto abre tanto puertas profesionales y posibilidades de trabajo en industria, como posibilidades de investigación y estudio en academia.

5.2. Formación

El proyecto requería una serie de conocimientos de los que no se disponía en un principio, como es normal en cualquier proyecto no trivial. Algunos de los conocimientos necesarios se aprendieron durante la carrera, pero por supuesto en un campo tan grande y cambiante como es la informática, la gran mayoría de tecnologías se tuvieron que investigar y aprender por cuenta propia.

Lo necesario para llevar a cabo el proyecto se expondrá en las siguientes secciones. Se intentaran omitir detalles que parezcan triviales o de muy poco

interés para el lector, de manera que la lectura no sea demasiado pesada.

Antes de hacer nada más, lo principalmente necesario fue el conseguir un entorno con integración continua.

Esto se presenta como una dificultad bastante grande ya que el único libro conocido sobre el tema [6] aunque parece fiable, según algunas personas es mucho más largo de lo que debería, y teniendo en cuenta que está centrado para Java se ha decidido intentar evitarlo.

Los materiales alternativos a este libro han sido generalmente entradas de blogs y hablando con gente que lo ha usado [8, 21] (aunque quizá no suficientemente experimentados).

Otras cosas que se han tenido que aprender han sido Tensorflow[42] y metodologías de uso de Git [5].

Tensorflow se estudió con los tutoriales oficiales relevantes para las partes a los que se quería dar uso. Aunque quizá no fuese necesario también se revisó la Tensorflow Dev Summit [2] para comprobar tanto la escalabilidad como las posibilidades más punteras en servidor y de distribución de los modelos en otros dispositivos.

Gitflow se siguió conforme a la explicación más antigua que se conoce, presentada en 2010, por Vincent Driessen [5].

5.3. Página web y versionado de bases de datos

No se conocía el desarrollo web con Python ni con Flask. Para adquirir el conocimiento sobre esto se usaron los tutoriales explore Flask[43] y megatutorial de Flask[44].

Hay que tener cuidado porque el primer tutorial está en Python2. Esto generalmente no da casi ningún problema,ya que Flask se ocupa de muchos de los problemas de versión, pero a la hora de usar SQLAlchemy los hash de las contraseñas en Python2 se pueden guardar como string, pero en Python3 se deben guardar como objetos de bytes.

Una de las lecciones más importantes aprendidas de estos tutoriales es la importancia del versionado del modelo de la base de datos. Esto se hace para poder migrar de una versión a otra de forma automática.

Es una manera bastante fácil de mantener la base de datos en buenas condiciones. Esto se debe a que si al hacer alguna migración falla, se cancela y

[42] Tensorflow: https://www.tensorflow.org/.
[43] explore Flask: https://exploreflask.com/en/latest/.
[44] megatutorial de Flask: https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world/page/0.

no perdemos datos, de manera que si alguna migración falla, podemos añadir lógica al como migrar para facilitar el proceso.

A parte de poder hacer versiones o 'commits' a mano, podemos hacer versionado automático de la base de datos, lo que infiere a partir de los modelos el cambio que ha ocurrido en la base de datos. Esta opción no se ha trabajado, ya que no se ha llegado a necesitar.

Alembic: primeros pasos

Alembic[45] nos permite de forma efectiva versionar las bases de datos de la misma manera que versionamos el código. Para poder empezar a usar Alembic en un proyecto, debemos ir a la carpeta del proyecto y ejecutar el comando:

```
$ alembic init alembic
```

Este comando creará tanto la carpeta '*alembic*', como el archivo '*alembic.ini*', ambos colgando de la ruta desde donde hayamos ejecutado el comando.

En la carpeta se almacenan las versiones y *scripts* de migración entre ellas. El archivo ini tiene la configuración de Alembic y para que sepa donde tenemos la base de datos tendremos que cambiar la ruta de la base de datos a donde tengamos la nuestra. Esta ruta se guarda en el parámetro *sqlalchemy.url*.

```
>>> sqlalchemy.url = driver://user:pass@localhost/dbname
```

Para crear una revisión (*commit* inicial en este caso) debemos usar el comando:

```
$ alembic revision -m "message"
```

Cabe destacar que esto influye sobre todo en el mantenimiento de sistemas en producción, ya que si no está en producción, los datos son más fáciles de recuperar y se pueden recurrir a alternativas menos ortodoxas (hacerlo a mano, usar *scripts* hechos a mano...).

5.4. Heroku y transición a contenedores

Tras tener la versión básica de la página web, se decidió hacer un despliegue en Heroku[46], una de las páginas que nos permiten publicar nuestra página web dinámica.

[45] Alembic: .
[46] Heroku: https://www.heroku.com/.

Heroku en concreto funciona con contenedores, pero nos reduce gran parte del peso del aprendizaje. La versión gratuita nos limita a un contenedor (donde dejamos nuestro proyecto como si fuera un ordenador normal) y nos deja un segundo contenedor limitado a una base de datos PostgreSQL[47] con unos límites de tamaño.

Tras conseguir completar el despliegue en Heroku y tener que integrar Tensorflow en el proyecto, para facilitar la integración, el despliegue en otros sistema y el mantenimiento del software, se decidió hacer los contenedores de forma explicita. La alternativa elegida fue **Docker**[48], por ser la opción más madura.

Para aprender a usar *Docker*, se usó la documentación oficial[49] y el libro *Docker orchestration* [23]. [hacer una explicación básica de Docker al estilo de la de alembic?]

5.5. Docker Compose y microservicios

Docker Compose[50] sirve para organizar nuestros contenedores en despliegues concretos, de manera que se puedan usar en cualquier momento, por ejemplo, podríamos tener varios *Docker Compose* con distintas configuraciones según nuestras necesidades, tanto para distintos servicios de alojamiento, como para distintas configuraciones de despliegue. Esto último es algo bastante complicado y que si nos propusiésemos hacerlo, tendría que ser con antelación o refactorizaciones decentemente grandes.

El aprendizaje de *Docker Compose* fue sobre todo mediante la documentación oficial, aunque el libro [23] también se usó. [explicar docker compose al estilo de alembic si/no]

Se puede desplegar con *Docker Compose*, pero con una gran limitación y es que no nos permite usar varios ordenadores de la manera que los servicios de orquestración nos lo permiten.

Los microservicios como arquitectura se conocieron debido a su popularidad actual, muchos blogs han hecho artículos sobre ellos y algunos *newsletter* como el de O'Reilly o el de **Nginx**[51] les han dado mucha importancia últimamente, adjuntando tutoriales y conferencias con las ultimas noticias.

[47] PostgreSQL: https://www.postgresql.org/.
[48] Docker: https://www.docker.com/.
[49] documentación oficial: https://docs.docker.com/.
[50] Docker Compose: https://docs.docker.com/compose/.
[51] Nginx: https://nginx.org/en/.

5.6. Orquestación

Los conocidos como '*orquestadores* nos facilitan, como su nombre indica, orquestar servicios entre varios ordenadores, lo cual nos permite elegir configuraciones más versátiles y acordes a nuestras necesidades.

Los orquestadores son *Docker Swarm*, *Kubernetes* y *OpenShift*. [explicar kubernetes... si/no?]

Docker swarm[52] fue el servicio que se intentó usar, debido a la existencia de secretos, que son ficheros con contraseñas o archivos que no deberían ser públicos. Estos secretos no están en *Docker Compose*, pero sí en *Swarm*, aunque el tamaño del enjambre sea uno, que es equivalente a *Compose*, lo que parece una estrategia comercial más que un diseño bien planteado.

Usar *Docker Swarm* dio gran cantidad de problemas y, tras una investigación online, la opinión general es que ya está listo para entornos de producción, la experiencia que podemos relatar es que *Swarm* da problemas, que son difíciles de diagnosticar. Quizá sea un caso concreto o que se hizo algo de manera incorrecta, pero tras bastante tiempo perdido sin ser capaces de avanzar, se decidió continuar con otras partes del proyecto.

5.7. Conclusiones

Gitflow, scrum y metodologías pensadas para equipos

Gitflow es un sistema complejo y que requiere conocimiento medio de *Git* para poder llevarse a cabo. Realmente no se necesita para proyectos tan pequeños como los de una sola persona y el trabajo que requiere realmente se vio que no merecía la pena. Aunque parece beneficiosa para equipos, no se ha podido comprobar y cabe considerar que existen críticas de la metodología en cuestión [22].

Scrum también es parecido, se ha visto como funciona en otros entornos y es sin duda beneficioso sobre todo para equipos de tamaño mediano. Una cosa que sí que merece la pena de *Scrum* es el tablero *kanban*, aunque solo seas una persona, merece la pena como forma de tener claras que partes del proyecto están hechas hasta que punto.

Tensorflow, versiones provisionales y tecnología punta

Tras aprender *Tensorflow* el verano pasado y no seguir usándolo tan activamente durante el primer cuatrimestre, se presento un problema, *Tensorflow* era una biblioteca en desarrollo inestable, al sacar la versión 1.0 se perdió parte del conocimiento adquirido, la *API* cambió, algunos de los *scripts* dejaron

[52] Docker swarm: https://docs.docker.com/engine/swarm/.

de funcionar y se desarrollaron bibliotecas más maduras y de más alto nivel, dejando otra parte del conocimiento adquirido parcialmente obsoleto.

Como ya se ha comentado, *Docker Swarm* también dio problemas, la suposición es que se hizo algo incorrecto en *Docker* o *Docker Compose* que hizo que *Swarm* no funcionase correctamente. Al ser tecnología punta no hay tanta información disponible en Stack Overflow[53] u otros foros similares sobre el tipo de problemas que nos encontramos.

Docker Compose también dio problemas con las configuraciones de red, aunque menos importantes que *Docker Swarm*. Suponemos lo mismo, pero estos sí que se pudieron arreglar.

Microservicios

Los microservicios, como hemos hablado, tienen una cantidad de aplicaciones muy diversa, y son una arquitectura muy capaz, sobre todo cuando estamos proporcionando SaaS, o un sistema que necesita de muchas partes semi independientes.

En el ámbito universitario, parece que la aplicación más clara que tienen, es la de que varios alumnos sean capaces de colaborar en un proyecto común. Esto se introduce ya que parece común el hecho de que varios alumnos quieran trabajar en un proyecto común y no puedan hacerlo por limitaciones en el sistema universitario. Con una arquitectura de microservicios podemos facilitar que se vea la distinción entre el trabajo de cada uno de manera que superamos esa limitación. También al ser más probable que un proyecto común acabe creciendo más allá de uno individual, esta arquitectura nos ayuda a tener una transición suave, sin excesivo trabajo, ni demasiadas complicaciones, al desplegar en producción.

Si no se piensa llevar el proyecto hasta unos niveles de complejidad elevados, o mantener una cantidad no trivial de servicios durante una cantidad de tiempo media-larga, no merece la pena, ya que fuerza más trabajo al equipo sin conseguir la mayoría de beneficios.

[53] Stack Overflow: https://stackoverflow.com/.

Conclusiones y líneas de trabajo futuras

En esta sección se exponen los resultados de la experiencia con el trabajo y lineas futuras con las que mejorar y dar continuidad al proyecto.

6.1. Conclusiones

Tras trabajar en un proyecto de tanta envergadura puedo decir sin temor a equivocarme que ha sido demasiado amplio, por culpa de mi propia ambición, tocando muchos temas de los cuales no salimos suficientemente preparados de la carrera:

- **Desarrollo web:** Es una de las partes de las que más trabajo tienen ahora mismo, debido a la cantidad de empresas que quieren tener una página web. Acabamos con una mínima preparación en C#, y sin suficiente conocimiento de la infraestructura física (servidores, *switches*, *routers*...), ni lógica (balanceadores de carga, *reverse proxies*, servidores web...) que se necesitan para poner una página web en funcionamiento.
- **Redes neuronales convolucionales profundas:** Tras cursar la asignatura de Computación neuronal y evolutiva destacó el desconocimiento de como modelar redes neuronales y que las convolucionales, que ahora mismo parecen el futuro de campos como la conducción automática, reconocimiento y generación de imágenes..., casi ni se mencionasen.

Otros temas como la virtualización de los entornos de despliegue no parece tan importante, ya que la mayoría de las aplicaciones pueden desarrollarse y desplegarse cómodamente sin este tipo de servicios. Quizá valga la pena

cierta mención de este tipo de soluciones con las ventajas e inconvenientes de contenedores y maquinas virtuales.

Personalmente pienso que la orquestración de servicios sin tener el hardware necesario es demasiado difícil. A pesar de su dificultad probablemente acabe siendo necesario el conocimiento de este tipo de sistemas para el desarrollo de software del tipo *Software as a Service* (SaaS), otra alternativa a este tipo de servicios es usar la 'nube', de empresas como heroku, amazon (aws)...

El proyecto ha llegado hasta un punto en el que puedo decir, que estoy satisfecho con lo conseguido, y a pesar de tener que recortar en ciertos aspectos como la investigación relacionada con el *deep learning*, el conocimiento adquirido durante este periodo parece invaluable. Por último querría recomendar a cualquiera que lea este trabajo que tenga cuidado con uno de los problemas más grandes con el desarrollo ágil, el *scope creep*, esta es la idea de que, según trabajamos en un proyecto, surjan nuevas e interesantes lineas de desarrollo, investigación o oportunidades y al perseguirlas acabemos ampliando el enfoque del proyecto a cubrir más temas, sin tener en cuenta si esto es suficientemente beneficioso o implica el recortar de otras partes del proyecto.

6.2. Líneas de trabajo futuras

Las lineas de trabajo futuras son más o menos claras:

- Nginx: Estudiar Nginx y su configuración como balanceador de carga, *reverse proxy* y servidor de http y https.
- Redis: Investigar sobre servicios similares a Redis y memcache que, básicamente, mejoran la eficiencia de uso de una base de datos mediante el cache en memoria principal de las consultas realizadas.
- NoSQL: No se recomienda usar NoSQL como fuente de escalabilidad a no ser que sea el único lugar del cuál podemos aumentar el rendimiento, esto se debe a que páginas como Facebook usan MySQL para los usuarios, las bases de datos NoSQL son para cargas mucho más pesadas, como estadísticas captadas en tiempo real. Algunas posibilidades son Apache Cassandra (Pycassa[54]) y MongoDB (Pymongo[55]).
- DNS: Se cree necesario investigar cómo hacer balance de carga a nivel de DNS, porque es una forma aparentemente sencilla de escalar una aplicación geográficamente, es decir permitir balance de carga según el tiempo de respuesta de distintos servidores que tengamos a nuestra disposición.

[54] Pycassa: https://pycassa.github.io/pycassa/.
[55] Pymongo: https://api.mongodb.com/python/current/.

Bibliografía

[1] Martin L Abbott and Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[2] Multiple authors. TensorFlow Dev Summit 2017, 2017. https://goo.gl/0sySqI.

[3] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[4] Ward Cunningham. Debt Metaphor, 2009. https://www.youtube.com/watch?v=pqeJFYwnkjE.

[5] Vincent Driessen. A successful Git branching model , 2010. http://nvie.com/posts/a-successful-git-branching-model/.

[6] Paul M Duvall. *Continuous integration*. Pearson Education India, 2007.

[7] Martin Fowler. TechnicalDebt, 2003. https://martinfowler.com/bliki/TechnicalDebt.html.

[8] Martin Fowler. Continuous Integration, 2006. https://martinfowler.com/articles/continuousIntegration.html.

[9] Martin Fowler. Microservices, 2014. https://martinfowler.com/articles/microservices.html.

[10] David Goodger. reStructuredText, 2002. http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html.

[11] David Goodger. PEP 287, 2002. https://www.python.org/dev/peps/pep-0287/.

[12] Michael Hausenblas. *Serverless Ops*. O'Reilly Media, 2016.

[13] Mark D Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, 1990.

[14] Image-net Team. Image-net. `http://image-net.org/`.

[15] Serverless Team. Serverless. `https://serverless.com/`.

[16] Pocoo. Pocoo Styleguide. `http://www.pocoo.org/internal/styleguide/`.

[17] Henrik Kniberg. Good and Bad Technical Debt, 2013. `http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt`.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003.

[21] Dan Radigan. Reaching true agility with continuous integration. `https://www.atlassian.com/agile/continuous-integration`.

[22] Adam Ruka. GitFlow considered harmful, 2015. `http://endoflineblog.com/gitflow-considered-harmful`.

[23] Randall Smith. *Docker Orchestration*. Packt Publishing, 2017.

[24] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbig-niew Wojna. Rethinking the inception architecture for computer vision. corr abs/1512.00567 (2015), 2015.