



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**Sign2Text - Transcripción de
lenguaje de signo mediante
deep learning -
Documentación técnica**



Presentado por Iván Ruiz Gázquez
en Universidad de Burgos — 6 de julio de 2022
Tutores: Dr. Daniel Urda Muñoz y Dr. Bruno
Baruque Zanon

Índice general

Índice general	i
Índice de figuras	iii
Índice de tablas	iv
Apéndice A Plan de Proyecto Software	1
A.1. Introducción	1
A.2. Planificación temporal	2
A.3. Estudio de viabilidad	9
Apéndice B Especificación de Requisitos	17
B.1. Introducción	17
B.2. Objetivos generales	17
B.3. Catalogo de requisitos	17
B.4. Especificación de requisitos	17
Apéndice C Especificación de diseño	19
C.1. Introducción	19
C.2. Diseño de datos	19
C.3. Diseño procedimental	19
C.4. Diseño arquitectónico	19
Apéndice D Documentación técnica de programación	21
D.1. Introducción	21
D.2. Sign2Text	21
D.3. Sign2Text - API	31

D.4. Sign2Text - Aplicación de cliente	37
Apéndice E Documentación de usuario	41
E.1. Introducción	41
E.2. Requisitos de usuarios	41
E.3. Instalación	41
E.4. Manual del usuario	41
Bibliografía	43

Índice de figuras

A.1. Características de la licencia MIT. Extraido de [3]	16
D.1. Superposición de información de pose sobre <i>frame</i> estático de una instancia de entrada en formato video. Imagen del <i>dataset</i> WLASL [9]	30
D.2. Utilización de la librería Tensorboard para visualizar <i>logs</i> y estructuras de redes neuronales a tiempo real	30
D.3. Esquema del cuerpo de la petición de entrada, con formato <i>multipart</i> [4]	35
D.4. Esquema de salida devuelto por el servidor. En el <i>target</i> se devolverá la etiqueta clasificada	36
D.5. Ejemplo de uso de Insomnia [5] para probar si el <i>endpoint</i> comproba el tipo de archivo recibido. Podemos ver señalado de rojo que la extensión del archivo es «.pdf» y no «.mp4»	36

Índice de tablas

A.1. Costes de <i>hardware</i>	10
A.2. Costes de <i>software</i>	11
A.3. Costes de herramientas o servicios que funcionan por uso	11
A.4. Costes de personal	12
A.5. Costes adicionales	13
A.6. Costes totales del proyecto	13
A.7. Precios de una teórica suscripción sobre la web de Sign2Text .	14
A.8. Licencias de las librerías usadas en el desarrollo de Sign2Text, la API y la demo web	15
A.9. Licencias de las herramientas	15
D.1. Lista de librerías y versiones usadas en Sign2Text	25
D.2. Lista de librerías y versiones usadas en Sign2Text-API	33
D.3. Lista de librerías y versiones usadas en Sign2Text-Cliente	39

Apéndice A

Plan de Proyecto Software

A.1. Introducción

En este proyecto se ha aplicado la metodología SCRUM de desarrollo ágil. Se han dividido todas las tareas del proyecto en *sprints* o *milestones* de 15 días cada uno. Cada sprint se compone de un *backlog* de tareas que se deben intentar cumplir en ese periodo.

Al comienzo de cada periodo, se ha realizado una reunión o *sprint meeting* en la cual se establecían las tareas que iban a realizar a continuación. Por otro lado, al finalizar cada *sprint*, se realizaba un *sprint review* del mismo para comprobar que tareas se habían completado y cuales no. Estas dos reuniones se realizaban en una sola en la que primero repasábamos los cambios y luego estudiábamos que hacer a continuación.

En un proyecto de software normal, se busca cumplir con todas las tareas establecidas en el *sprint*, y avanzar una iteración o *release* sobre el producto. En un proyecto de investigación como ha sido este, esta tarea se complica; ya que algunas de las *issues* son más bien estimaciones sobre las posibles tareas que se puede realizar más que una garantía de que se cumplan. Veremos varios ejemplos de estos a lo largo de los *sprints*.

A.2. Planificación temporal

Para la planificación de tareas, *milestones* y tiempos de entrega se han usado las *issues* internas de Gitlab con la idea de mantener el proyecto lo más unificado posible.

Un problema que tiene esto es que Gitlab tiene muchas características bajo un muro de pago. Por ejemplo, no es posible asignar puntos de historia a las *issues* y tampoco podemos exportar gráficos de *burn down*¹.

Por otro lado, al ser un proyecto con repositorio cerrado y con administrador externo al autor, no se ha podido aumentar el plan a un plan *premium* para usar dichas *features*. Por esto vamos a remitirnos a mostrar el objetivo principal en cada *sprint* así como las tareas que lo componen. Hablaremos del porcentaje de tareas completas y faremos un *review* de aquellas no completas y que deban ser movidas a siguientes *sprints*.

En este proyecto se han usado etiquetas para catagorizar las *issues* de forma ordenada según tipo, estado, prioridad y otros. Estas etiquetas están extraídas de [8], un repositorio con etiquetas para reutilizar entre proyectos. La lista completa es:

- **Tipo:**

- **Bug:** Simboliza un error en el código que no permite que continúe la ejecución
- **Discussion:** Esta etiqueta simboliza una *issue* que debe ser debatida en el *sprint review* o discutida en el propio foro de la tarea.
- **Documentation:** Una tarea marcada como *documentation* simboliza una tarea que trata sobre el estudio de literatura necesaria para la comprensión de algunos contenidos, la realización de comentarios en el código o la creación de archivos «Readme». Las tareas relacionadas con la memoria y los anexos que se están leyendo también entran en esta categoría.
- **Enhancement:** Esta etiqueta simboliza una tarea que se trata de una mejora en el código o de una mejora en la documentación.

¹Se ha pensado el uso de herramientas externas, como Zenhub; pero estas herramientas solo tiene conexión de terceros con Github

- **Epic:** Esta etiqueta simboliza una *issue* que se trata de una historia completa que se puede dividir en más *issues* y que se puede tratar de una *release* en el *sprint*.
- **Feature Request:** Esta etiqueta simboliza una *issue* que se trata de una petición de una nueva característica.
- **Question:** Esta etiqueta simboliza una *issue* que se trata de una pregunta que se puede responder en el *sprint review* o en el propio foro de la tarea. Se diferencia de *discussion* en que *question* solo se inicializa para ser preguntada en el *sprint review*, mientras que *discussion* es una tarea ya iniciada que no puede terminarse, o que, en su defecto, se ha vuelto a abrir tras cerrada.

- **Estado:**

- **Can't Reproduce:** Esta etiqueta simboliza una *issue* que no se puede reproducir debido a que el estado del proyecto no ha llegado al punto esperado todavía.
- **Confirmed:** Simboliza una tarea que va a ser realizada de forma segura.
- **Duplicate:** Esta etiqueta simboliza una *issue* que se trata de una duplicación de otra *issue*.
- **Needs Information:** Cuando una *issue* está marcada con esta etiqueta necesita información adicional para ser completada. Normalmente marcamos una tarea así cuando necesitamos leer información de artículos para continuar.
- **Won't do/fix:** Simboliza una tarea que no va a ser realizada en el *sprint*. Se usa para mantener la *issue* pero para indicar que el proyecto se ha orientado hacia otro lado.

- **Prioridad:**

- **Critical**
- **High**
- **Medium**
- **Low**

- **Otras:**

- **Help wanted:** Esta etiqueta simboliza una *issue* con la que se necesita ayuda.

- **Good first issue:** Identifica *issues* al comienzo del proyecto, que marcan un buen paso en los objetivos generales.
- **«Gran issue de los tutores!!»:** Esta *label* especial es para indicar que una *issue* ha sido propuesta por uno de los tutores en un *sprint meeting*.

Sprint 1: Toma de contacto

- **Objetivo general:** Se busca tomar contacto con el *framework* PyTorch, *datasets* públicos de lenguaje de signos, así como el comienzo de estudio de literatura sobre *machine learning*.
- **Intervalo:** 01/02/2022 - 15/02/2022
- **Tareas:**
 1. Movimiento de código de *notebooks* de **Jupyter** a módulos de **Python**.
 2. Estudio de la literatura sobre *machine learning*, más concretamente *supervised learning*, y dentro de este, las redes CNN y sus estructuras, como *ResNet*.
 3. Estudio de balance de *dataset* usado. Comprobamos si las etiquetas tienen suficientes datos para que el *framework* pueda ser entrenado y validado correctamente.
 4. Aprendemos a usar el *loader* interno de PyTorch para cargar y transformar datos.
 5. Creación de un monorepo para el repositorio del modelo, la demo web y la API
 6. Creación de *scripts* que nos permitan transformar distintos *datasets* de video a *frames*.
 7. Elección de la plataforma de entrenamiento.
 8. Descarga del dataset del sitio oficial.
 9. Preparación del *continuous integration* en Gitlab.
- **Sprint Review:** El primer *sprint* comenzó con muchas tareas, la mayoría de ellas con dificultad mínima, por lo que se cumplieron todas. Elegimos desarrollar el modelo en local, por rapidez y porque el equipo en el que ha sido entrenado es suficientemente potente.

La única *issue* que se transformó fue la «creación de un monorepo», ya que se optó por crear un repositorio separado por proyecto (modelo/*front/back*).

Cabe destacar la utilidad de implementar el *continuous integration*, así como los *scripts* de transformación de datos, que ayudaron muchísimo en la agilidad del proyecto en los siguientes *sprints*.

Sprint 2: Usando *frames*

- **Objetivo general:** En este sprint buscamos usar una red preentrenada para reconocer *frames* de un vídeo. Debemos crear la forma de importar los *frames* para el entrenamiento y estudiar como cambiar los parámetros de nuestra red para que se adapte a la entrada.
- **Intervalo:** 16/02/2022 - 03/03/2022
- **Tareas:**
 1. Uso de imágenes concatenadas: debemos modificar los *scripts* para poder exportar un conjunto de *frames* de un vídeo como una única imagen de *frames* concatenados horizontalmente.
 2. comprobación de la red usando *datasets* variados.
 3. Se deben añadir más opciones a los *scripts* para cuando usemos una entrada de la red en formato vídeo.
 4. Estudio del uso de poses e I3D para la clasificación.
 5. Creación de dataset de vídeo.
- **Sprint Review:** En este segundo sprint avanzamos mucho. Alteramos una red *ResNet* preentrenada para que sea capaz de reconocer *frames* de un video. En el caso del *dataset* original de signos, no funciona muy bien. Por esta razón probamos con un *dataset* de imágenes estáticas de animales, así como uno de signos que identifican una sola letra del abecedario. Ambos funcionan de forma excepcional. En la reunión del *sprint review* se sugiere entonces comenzar a crear una red basada en CNNs desde cero, también de clasificación de *frames*.

Sprint 3: ¿Transformers?

- **Objetivo general:** Inicialmente, estaba previsto tener un clasificador de vídeo funcional antes de esta etapa. Por esto, en este *sprint*, la idea

era implementar una red *transformers* para poder adaptar las salidas del clasificador a un formato más legible. Debido al retraso en tareas, se decidió no implementar la red *transformer* y pasar directamente a la red convolucional.

- **Intervalo:** 04/03/2022 - 19/03/2022
- **Tareas:**
 1. Uso de concatenación de *frames* como entrada de la red.
 2. Compresión de la red para la fase de inferencia en *real time*.
 3. (Cancelada) - Creación de una red *transformers* para adaptar la salida del clasificador.
 4. (Cancelada) - *Tests* para la capacidad de generar frases y corregirlas.
 5. Implementación de una red convolucional desde cero.
- **Sprint Review:** Recordemos que en el anterior *sprint*, el estado final fue la creación de una red de clasificación de *frames* con una red preentrenada. El siguiente objetivo marcado fue la instanciación de un clasificador desde cero. En esta reunión estudiamos la red creada. Con ayuda de los tutores, comenzamos a crear las distintas capas que componen la red convolucional.

Sprint 4: Inferencia ONNX

- **Objetivo general:** El objetivo de este sprint es por fin implementar una red convolucional de clasificación de vídeo. Tras esto se pretende poder inferir las entradas a tiempo real con una *webcam* mediante el uso de ONNX.
- **Intervalo:** 20/03/2022 - 04/04/2022
- **Tareas:**
 1. Añadir inferencia con ONNX a tiempo real.
 2. Añadir inferencia por *webcam*.
 3. (Aplazada) - Preparación de la red para consumo con API.

- **Sprint Review:** Tras la finalización de este *sprint*, tenemos una red de clasificación de vídeo que funciona medianamente bien en clasificación de acciones, pero que funciona mucho peor con el *dataset* de signos. En este punto, y tras ver que la red convolucional funciona y entrena, se comienza a estudiar la forma de mejorar el *accuracy* de la red. Con la inferencia por webcam ya disponible, pensamos en añadir información de poses a la red, y así comprobar si mejora su rendimiento.

Sprint 5: API Rest

- **Objetivo general:** El objetivo original para estas fechas era la creación de una API *rest* con la cual se pudiera consumir el modelo creado. En cambio, en este *sprint*, se modifica la red convolucional creada para no solo clasificar los signos a texto, sino para detectar poses también.
- **Intervalo:** 05/04/2022 - 19/04/2022
- **Tareas:**
 1. (Aplazada) - Crear plataforma de API rest.
 2. Aplicación de una doble salida de la red (Pose + Texto)
- **Sprint Review:** Repasamos que en este *sprint* se ha aplicado una doble salida de la red neuronal (Pose + Texto) con la idea de que el cálculo de las poses ayude a mejorar el *accuracy* del modelo. En la reunión vemos como funciona esto. Observamos un aumento del *accuracy* sustancial. En este punto se decide implementar alguna forma con la que los usuarios puedan probarlo.

Sprint 6: Una demo Web

- **Objetivo general:** El objetivo de este *sprint* es la creación de una plataforma de cliente con la que se pueda usar el modelo entrenado en el anterior *sprint*. Por otro lado se debe crear una API desde la que se consuma dicho modelo.
- **Intervalo:** 20/04/2022 - 04/05/2022
- **Tareas:**
 1. (Aplazado) - Creación de un pequeño *front-end* para que los usuarios puedan probar el modelo.

2. (Aplazado) - Compra del dominio del proyecto
 3. Creación de servidor con API *rest*
- **Sprint Review:** En este *sprint* se ha comenzado a crear un *endpoint* con un servidor usando **FastAPI**. Esta reunión de *sprint* se anula por que los avances no son muy extensos y no se iban a añadir nuevas tareas al próximo *sprint*.

Sprint 7: *Deploy*

- **Objetivo general:** En este sprint se busca terminar la implementación de la API, así como la creación de la demo web. Por otro lado se pretende poder hacer *deploy* de la API en la nube, así como encapsular el modelo en un contenedor docker.
 - **Intervalo:** 05/05/2022 - 15/06/2022
 - **Tareas:**
 1. Terminar la API *rest* y abrirla para su uso.
 2. Estudio de los *IaaS* para ver dónde se va a tener el *host* del modelo.
 3. Creación de la *landing page* con info de los repositorios
 4. Arreglar *pylint* para la versión final
 5. Creación de contenedor con Docker
 6. Subida del contenedor a Dockerhub
- **Sprint Review:** En este último *sprint*, de doble duración, desarrollamos finalmente la API *rest* y la demo web.

En la segunda mitad del sprint, con la *API* creada, buscamos la forma de hacer *deploy*. Primero estudiamos el uso de AWS y Azure para la API, usando finalmente Google Cloud. Contenedorizamos el modelo y el servidor con Docker para que sea de dominio público y lo subimos a Dockerhub. Asignamos un subdominio a la *API*.

Por otro lado, la demo se sube a **Vercel**, y se le asigna el dominio comprado.

Con esto hecho, nos ponemos a terminar la documentación.

Conclusión

El uso de la metodología SCRUM en proyectos de *software* es muy útil. Ayuda a mantener una buena organización de los proyectos y a tener una buena comunicación entre los equipos. Te permite mantener un *record* de todas las tareas que se cumplen, y da más o menos importancias a unas u otras.

De echo, en la parte más relacionada con *software*, como es la demo web o la creación de la API, ha sido muy útil. Donde no lo ha sido tanto es en la parte de creación del modelo.

Debido a que es un proyecto con mucha base en la investigación, el establecimiento de tareas con límites de tiempo no tiene mucho sentido. Esto es porque no sabes si una mejora que estás aplicando en el modelo va a derivar en lo que estás buscando, o por el contrario va a ser prácticamente «inútil».

En este proyecto, hemos experimentado e iterado sobre el modelo con muchísimos pequeños cambios en parámetros para encontrar la forma correcta de entrenar a la red. Estos cambios nos han llevado a veces semanas, cosa que no podíamos saber al principio. El tener tareas que pongan «presión» sobre el proyecto es al final contraproducente y produce un estrés sobre el investigador, que no sabe si puede o no cumplir las tareas porque no sabe si los métodos investigados funcionarán o no.

Por esto creo que la aplicación del método SCRUM en el apartado de investigación del proyecto, no ha sido la mejor opción, pudiendo sustituirlo en esta parte por un «método ágil intuitivo», con el cual se trabaja de forma ágil, pero adaptando intervalos y tareas en cada momento según el estado del proyecto.

A.3. Estudio de viabilidad

En esta sección vamos a estudiar la viabilidad económica y legal del proyecto. Al ser mayoritariamente de investigación, es normal, como veremos, que los costes superen en creces a los posibles beneficios que se puedan obtener. Vamos a estudiar los costes en el *hardware* empleado, tiempo de desarrollo y gasto en herramientas y *software* de terceros. Estudiaremos entonces formas de conseguir transformar el modelo en un producto viable.

Por otro lado, veremos en el apartado de la viabilidad legal, como existen algunos factores limitantes que reducen la capacidad de salida a producción, como las licencias impuestas por las librerías o por los creadores de *datasets*.

Viabilidad económica

Para el estudio de la viabilidad económica del proyecto, vamos a estudiar los costos en distintos apartados. El primero de ellos, el costo de equipamiento y *hardware*.

Costes

La realización de este proyecto conlleva una serie de costes. Estos costos se van a dividir en *hardware*, *software*, de «personal» y otros.

Costes *hardware*

Podemos observar en la tabla A.1 un desglose de gastos de *hardware*. Entre los costes se encuentra el dispositivo principal de sobremesa en el que se ha desarrollado el proyecto. Este equipo tiene un procesador AMD Ryzen 3900X de 12 núcleos a 4.6 GHz, con 32GB de memoria RAM y GPU Nvidia RTX 3060ti, con un precio de mercado de aproximadamente 3100€. El dispositivo tiene una antigüedad de 2 años.

Por otro lado tenemos un portátil ASUS ROG con un procesador Ryzen 5900HX, 16GB de RAM y GPU RTX 3070 con un precio aproximado de 1700€. Este dispositivo tiene una antigüedad de 8 meses.

Se considera que la amortización de ambos equipos ronda los 6 años, por lo que al haberlos usado 6 meses, se ha amortizado el 10 % de la inversión.

Concepto	Coste (€)	Amortizado (€)
Ordenador sobremesa	3100	310
Ordenador portátil	1700	170
Total	4800	480

Tabla A.1: Costes de *hardware*

Costes *software*

En este apartado vamos a revisar los costes en *software* y programas de terceros usados en el proyecto. Vamos a considerar la amortización a 1 año (la mayoría de licencias listadas ofrecen una versión de 1 año). Como *software* incluiremos también el sistema operativo. Al haber usado estos programas 6 meses, consideramos un 50 % de amortización. Podemos ver el desglose en la tabla A.2

Concepto	Coste (€)	Amortizado (€)
IDE VSCode	0	0
x2 Windows 10 PRO	399,98	199,99
TeXMaker	0	0
Gitlab DevOps - CI	1188	594
Dropbox	120	60
Vercel Hosting	240	120
Docker	252	126
Total	2199,98	1099,99

Tabla A.2: Costes de *software*

Costes según uso

En la tabla A.3 analizamos costes según uso. Estos incluyen los gastos del uso de servicios *cloud*.

Concepto	Tiempo (h)	Coste/hora (€/h)	Coste (€)
Google Cloud Build - CI	10	0,18	1,8
Google Cloud Run	300	0,03	9
AWS Elastic Registry	100	0,12	12
AWS Elastic Compute	100	0,10	10
AWS Elastic Balancing	100	0,10	10
AWS Route 53	2	0,25	0,50
Total	612	-	34,3

Tabla A.3: Costes de herramientas o servicios que funcionan por uso

Costes de personal

Analizamos ahora el coste de personal. El proyecto ha sido llevado a cabo por un desarrollador empleando aproximadamente 3 horas diarias durante 6 meses. A una media de 25 días por mes, obtenemos 450 horas totales de trabajo. Si las horas trabajadas semanales se reducen a 40 horas; entonces, se han trabajado $450/40 = 11,25$ semanas. Por facilidad vamos a redondear hacia arriba a 12 semanas (o 3 meses).

Podemos observar en [6] que el sueldo base medio neto en España para un desarrollador Junior es de 25698€. Si a esto le deducimos las cuotas de seguridad social e IRPF, calculadas con [10], obtenemos los datos de la tabla A.4

Concepto	Coste (€)
Salario neto anual	20354
Cuota IRPF (2Cuota Seguridad Social)	1631,8
Salario bruto anual	25698
Salario bruto 3 meses	6424,5

Tabla A.4: Costes de personal

Otros costes

Estos costes identificas costes adicionales que no pertenecen a ningún grupo anterior. Estos identifican compras únicas, gastos de luz, agua, entre otros. Puedes ver la lista completa en la tabla A.5

Concepto	Coste (€)
Dominio web	12
Impresión de documentación	45
Conexión a internet	180
Agua	80
Calefacción	259
Electricidad	205
Diseño de logo	130
Total	911

Tabla A.5: Costes adicionales

Costes totales

Podemos ver en la tabla A.6 la suma de costes del proyecto.

Concepto	Coste (€)
<i>Hardware</i>	4800
<i>Software</i>	1099,99
<i>Según uso</i>	34,3
<i>Personal</i>	6424,5
<i>Otros</i>	911
Total	13269,89

Tabla A.6: Costes totales del proyecto

Beneficios

El presente proyecto puede obtener ingreso del alquiler del uso del modelo neuronal. El acceso al modelo se puede obtener de tres formas:

- Descarga directa del contenedor docker que lo alberga: El modelo se encuentra alojado de forma pública en un contenedor docker en Dockerhub. Si queremos monetizar este aspecto, debemos asignar un precio al contenedor y cobrar por la descarga. Se plantea por ejemplo un precio único de descarga de 999€.

- Acceso al modelo por API *rest*: En la actualidad, se puede acceder a una API *rest* también de acceso público. Contiene un *endpoint* con el cual se puede consumir el modelo. Aquí se plantea una forma de cobro por uso o petición: i.e. 1€ cada 1000 peticiones.
- El otro modo de acceso está más orientado a usuarios no-expertos. Esto es una web en la cual se puede usar el modelo arrastrando un vídeo y recibiendo el resultado. Esta web se podría comerciar mediante la creación de una cuenta con una suscripción mensual. Podemos ver en la tabla A.7 un planteamiento sobre dicha suscripción. Una persona beneficiada es una persona que usa lenguaje de signos como su principal medio de comunicación.

Tipo de suscripción	Intervalo	Precio/mes (€)
Beneficiado	Cualquiera	Gratis
Estudiante	Mensual	4,99
Estudiante	Anual	2,99
Normal	Mensual	6,99
Normal	Anual	4,99
<i>Enterprise</i>	Mensual	10,99
<i>Enterprise</i>	Anual	8,99

Tabla A.7: Precios de una teórica suscripción sobre la web de Sign2Text

Viabilidad legal

En esta sección vamos a hablar sobre las licencias asociadas al desarrollo de este proyecto. Comenzaremos primero con las licencias establecidas singularmente por cada librería o *framework* usados. Veremos cuáles de las licencias usadas tienen mayor prioridad y como nos obligan a aplicar una u otra licencia en nuestro propio proyecto.

En la tabla A.9 vemos una lista de las librerías y *frameworks* usados asociados a su licencia. En la tabla ?? podemos ver lo mismo pero para las herramientas usadas en el desarrollo. En las tablas se incluyen todas las librerías usadas a lo largo de todo el desarrollo, incluida la API y la demo web.

Libreria	Licencia	Administrador de paquetes
OpenCV Python	MIT	pip
PyLint	GPLv2	pip
ONNX	Apache v2.0	pip
Black	MIT	pip
Numpy	BSD	pip
Matplotlib	Python Software Foundation	pip
Matplotlib Inline	BSD 3-Clause	pip
MyPy	MIT	pip
Pillow	BSD	pip
FastAPI	MIT	pip
onnxruntime-gpu	MIT	pip
tensorboard	Apache	pip
torch (PyTorch)	BSD	pip
torchinfo	MIT	pip
torchvision	BSD	pip
uvicorn	BSD	Pip
astro	MIT	npm
tailwind	MIT	npm
svelte	MIT	npm

Tabla A.8: Licencias de las librerías usadas en el desarrollo de **Sign2Text**, la API y la demo web

Libreria	Licencia
VSCode	MIT
Jupyter Notebook	BSD Modificada

Tabla A.9: Licencias de las herramientas

Podemos ver una lista con todas licencias y sus características en [\[1\]](#)

Todas las licencias son licencias permisivas. Esto nos permite marcar todas las partes de este proyecto con la licencia MIT. Podemos ver las características de esta licencia en la figura A.1.

MIT		MIT License
Can	Use/reproduce, Distribute, Modify/merge, Sublicense, Commercial use	Licence comment:
Must	Incl. Copyright, Include licence	MIT is the most recommended permissive licence: short and very popular (probably the most used worldwide). OSI states it is supported by a strong community. Basically, you can do whatever you want as long as you include the original copyright and licence notice in any copy of the software/source. Another version MIT-0 does not mention the obligation of including the copyright notice. MIT is one of the permissive licences to be used (authorised) by French administrations.
Cannot	Hold liable	
Compatible	Permissive, GPL, Other copyleft, Linking freedom, For software	
Law	Not fixed/local	
Support	Strong Community, OSI approved, FSF Free/Libre	

Figura A.1: Características de la licencia MIT. Extraido de [3]

Apéndice B

Especificación de Requisitos

- B.1. Introducción**
- B.2. Objetivos generales**
- B.3. Catalogo de requisitos**
- B.4. Especificación de requisitos**

Apéndice C

Especificación de diseño

- C.1. Introducción**
- C.2. Diseño de datos**
- C.3. Diseño procedimental**
- C.4. Diseño arquitectónico**

Apéndice D

Documentación técnica de programación

D.1. Introducción

En esta sección vamos a describir la documentación técnica de programación. Vamos a explicar la estructura de directorios, el proceso de instalación del entorno de desarrollo y todas las librerías necesarias. Seguiremos explicando como ejecutar el proyecto y terminaremos con una breve explicación de las pruebas de sistema aplicadas.

Este proyecto está dividido en tres repositorios diferentes. Uno de ellos (**Sign2Text**), que es el proyecto principal, y dos de ellos (**Sign2Text-API** y **Sign2Text-Demo**), que son los repositorios de la API y la demo web respectivamente.

D.2. Sign2Text

Este es el proyecto principal, el que se ha estudiado en la memoria. Se puede acceder en alguno de los siguientes enlaces:

- Gitlab (Privado): <https://gitlab.com/HP-SCDS/Observatorio/2021-2022/sign2text/ubu-sign2text.git>
- Github (Público): <https://github.com/irg1008/Sign2Text.git>

El repositorio original es el de Gitlab, y contiene más tareas, así como los *milestones* y tareas realizadas.

Estructura de directorios

La estructura de este proyecto es:

- **/**: En la raíz del proyecto encontramos archivos como el «README», archivos de configuración de Python, la licencia o el archivo de configuración del *continuous integration*. También tenemos la carpeta con el entorno virtual de Python.
- **/docs**
 - **/assets**: En esta carpeta tenemos archivos usados en README, como los logos.
 - **/memoria**: Aquí encontramos la documentación creada en formato L^AT_EX sobre la memoria y los anexos.
- **/data**: En esta carpeta podemos encontrar archivos usados para entrada del modelo o entrenamiento de la red. Tenemos distintos *datasets* y una carpeta con videos demo para la web.
- **/models**: Aquí es donde guardamos los modelos entrenados.
- **/src**: En esta carpeta encontramos el código fuente de la aplicación.
 - **/nets**: Aquí guardamos todo el código fuente relacionado con las redes neuronales, tanto módulos de Python como cuadernillos de Jupyter.
 - **/common**: En esta carpeta tenemos código fuente común a todas las redes neuronales. En la raíz tenemos un archivo con útiles para la inferencia mediante *webcam*.
 - **/config**: En esta carpeta se albergan los archivos de configuración comunes a todas las redes, como la transformación de las imágenes a la entrada de la red.
 - **/utils**: Aquí tenemos archivos útiles como comprobaciones de balanceo de *datasets*, funciones comunes de *plot* y salida y similares.

- **/res_net:** Esta carpeta es la carpeta que alberga el primer modelo neuronal de clasificación de imágenes desarrollado. En la raíz tenemos dos cuadernillos de *Jupyter*. Con el primero llamamos a los módulos de **Python** de carga de datos y entrenamiento. Con el segundo inferimos las imágenes.
 - ◊ **/config:** En esta carpeta tenemos los archivos de configuración únicos a esta red, así como las variables de hiperparámetros y la lista de *datasets* disponibles.
 - ◊ **/lib:** En esta carpeta tenemos el modelo con la especificación de la red neuronal, y el script de entrenamiento de la red.
 - ◊ **/utils:** Esta carpeta alberga los archivos de carga de datos, salida y compresión de la red.
- **/simple_net:** Esta carpeta es la carpeta que alberga el segundo modelo neuronal de clasificación de imágenes desarrollado. En la raíz tenemos dos cuadernillos de *Jupyter*. Con el primero llamamos a los módulos de **Python** de carga de datos y entrenamiento. Con el segundo inferimos los videos.
 - ◊ **/config:** En esta carpeta tenemos los archivos de configuración únicos a esta red, así como las variables de hiperparámetros y la lista de *datasets* disponibles.
 - ◊ **/lib:** En esta carpeta tenemos el modelo con la especificación de la red neuronal, y el script de entrenamiento de la red.
 - ◊ **/utils:** Esta carpeta alberga los archivos de carga de datos, salida y compresión de la red.
- **/two_out_net:** Esta carpeta es la carpeta que alberga el tercer modelo neuronal de clasificación de imágenes desarrollado. En la raíz tenemos dos cuadernillos de *Jupyter*. Con el primero llamamos a los módulos de **Python** de carga de datos y entrenamiento. Con el segundo inferimos los videos.
 - ◊ **/config:** En esta carpeta tenemos los archivos de configuración únicos a esta red, así como las variables de hiperparámetros y la lista de *datasets* disponibles.
 - ◊ **/lib:** En esta carpeta tenemos el modelo con la especificación de la red neuronal, y el script de entrenamiento de la red.
 - ◊ **/utils:** Esta carpeta alberga los archivos de carga de datos, salida y compresión de la red.

- ◊ **tensorboard**: En esta carpeta almacenamos los eventos ocurridos en las distintas ejecuciones para poder ver gráficos a tiempo real, así como ver las imágenes generadas. También se proporciona un README para ver como abrir dichos *logs*.
- **/scripts**: Este directorio alberga herramientas útiles y reutilizables para el desarrollo de la aplicación.
 - **/common/utils**: En esta carpeta tenemos archivos comunes entre los *scripts*, como el manejo de archivos locales o el *log* de las salidas.
 - **/quantization**: Este directorio contiene un *script* para cuantización de modelos neuronales en formato ONNX.
 - **/SimpleVideo2Frame**: Esta carpeta tiene uno de los *scripts* usados para la transformación de videos a *frames*. Veremos como se ejecuta este script en el apartado de «ejecución».
 - **/WLASL2Frame**: Esta carpeta contiene un *script* adaptado a la transformación de datos con el formato único del *dataset* WLASL [9].

Manual del programador

Entorno

El entorno de ejecución de este proyecto es:

- Lenguajes: Python
- Versión: 3.9.8
- Conda: No

Las bibliotecas usadas en este proyecto se pueden ver en la tabla D.1. Todas estas librerías se detallan en el archivo `requirements.txt` para una instalación más rápida. Es muy importante mantener las versiones indicadas en D.1 para asegurar el correcto funcionamiento del proyecto. Es posible que funcione con versiones más modernas, pero se asegura con las versiones congeladas indicadas.

Por otro lado, necesitaremos algún modo de ejecutar cuadernillos de `Jupyter`. Para esto podemos usar `conda` o la extensión oficial para el

Biblioteca	Versión
pylint	2.14.3
black	22.3.0
black[jupyter]	22.3.0
numpy	1.22.4
mypy	0.961
mypy-extensions	0.4.3
types-PyYAML	6.0.8
typing	3.7.4.3
typing_extensions	4.2.0
PyYAML	6.0
matplotlib	3.5.2
matplotlib-inline	0.1.3
tensorboard	2.9.0
tensorboard-data-server	0.6.1
tensorboard-plugin-wit	1.8.1
protobuf	3.20.1
torch	1.10.2+cu113
torch-tb-profiler	0.4.0
torchensemble	0.1.7
torchinfo	1.7.0
torchvision	0.11.3+cu113
moviepy	1.0.3
onnx	1.12.0
onnxruntime-gpu	1.11.1
opencv-python	4.6.0.66
ipykernel	6.15.0
pandas	1.4.2
Pillow	9.1.1

Tabla D.1: Lista de librerías y versiones usadas en *Sign2Text*

IDE *VSCode*. En nuestro recomendamos usar *VSCode*, ya que es el que se ha usado para la realización de esta práctica.

Compilación, instalación y ejecución del proyecto

En estas secciones vamos a ver como instalar y ejecutar las distintas partes del proyecto. Antes de nada, debemos descargarnos el código fuente

de GitHub o Gitlab. Para hacer esto usamos el comando `git clone`. Puedes ver la url al comienzo de esta sección.

Una vez hecho, entramos en la carpeta recién descargada para comenzar el paso de instalación.

Instalación

1. Con el repositorio descargado, vamos a crear un entorno de Python virtual. De este modo no contaminamos las dependencias globales de pip. Para esto seguimos los siguientes pasos:
 - 1.1. Si no tenemos instalado con `pip install virtualenv`
 - 1.2. Ejecutamos el comando `virtualenv venv` o `python -m venv venv` para crear nuestro directorio `venv` con el entorno de Python virtual.
 - 1.3. Activamos el entorno virtual con `source venv/bin/activate` en bash o `./venv/bin/activate` en windows.
2. Una vez tenemos el entorno inicializado con el código fuente, podemos instalar las dependencias con el comando `pip install -r requirements.txt`.
3. (Opcional): Si nuestro sistema tiene una GPU, podemos ejecutar todos los scripts de PyTorch con ella. Para esto debemos descargar la versión de PyTorch preparada para CUDA.

Para hacer esto usamos el comando `pip install torch==1.10.2+cu113 torchvision==0.11.3+cu113 torchaudio==0.10.2+cu113 -f https://download.pytorch.org/whl/cu113/torch_stable.html`

Una vez hecho esto, ya tenemos todas las bibliotecas necesarias para ejecutar el proyecto.

Ejecución

Dentro de este proyecto, tenemos dos partes separadas que podemos ejecutar. La primera son los *scripts*, que aplicamos sobre los modelos ya entrenados o sobre los *datasets* que vamos a usar para el entrenamiento. Podemos acceder a ellos en la carpeta `/src/scripts`.

En este carpeta tenemos los siguientes *scripts*

- Quantización: Tenemos un *script* con el cual podemos cuantizar un modelo en formato ONNX. Para esto haremos `python quantize.py`, estando en el directorio del *script*. Si necesitamos cambiar la url del modelo a cuantizar, debemos entrar en el *script* y cambiar la variable dentro del *pipeline* principal del archivo.
- SimpleVideo2Frame: Este *script* se encarga de convertir un video en una serie de imágenes. Para ello, usamos el comando `python simpleVideo2Frame.py`, o si estamos en `windows`, también podemos usar `./run.bat`. Las opciones disponibles para este comando se deben insertar por parámetro. Los parámetros de este *script* son los siguientes:
 - `-i, --input`: El *path* donde se encuentran los videos de entrada a convertir. Tipo de entrada es *string*.
 - `-o, --output`: El *path* donde se guardarán las imágenes resultantes. Tipo de entrada es *string*.
 - `-l, --labels`: El número de etiquetas del dataset que se deben procesar y exportar. Cuanto mayor sea el número de etiquetas, más tiempo tardará en ejecutarse el script. Tipo de entrada es entero.
 - `-c, --convert`: Si se debe convertir el video a imágenes o se debe transformar a formato mp4 a la salida. Tipo de entrada es booleano. Por defecto es verdadero.

Un ejemplo de ejecución de este *script* es: `python ./simplevideo2frame.py -i /data/actions/raw_videos -o /data/actions/frames -l 8`

- WLASL2Frame: Este *script* es una versión del anterior adaptado a la transformación de *datasets* de lenguaje de signos. Lo podemos ejecutar con `python video2frame.py`. En este *script* no solo tenemos entrada por argumentos, como veremos a continuación; si no que también tenemos un archivo de configuración `config.yml` en el que podemos especificar las variables necesarias. En caso de que insertemos los datos por argumentos, tenemos las siguientes opciones:

- Opciones globales
 - `-i, --input`: Carpeta con los videos a procesar (Opcional).
 - `-o, --output`: Carpeta donde se guardaran los frames (Opcional).

- -l, -labels: Número de labels a procesar. Puede ser un número o “all” para procesar todos (Opcional).
- -c, -config: Archivo de configuración con las etiquetas (Opcional).
- -h, -help: Muestra esta ayuda.
- Extracción de *frames*
 - -m, -merge: Indica si se debe unir los frames en una imagen (Opcional).
 - -f, -frames: Número de frames aleatorios del video (Opcional).
- Extracción de videos
 - -v, -video: Extrae todos los frames del video en una carpeta única para ese video. Deshabilita las opciones -f y -m (Opcional).

Se puede obtener más información, así como varios ejemplos sobre este *script* en el «README» dedicado bajo el mismo directorio.

Los siguientes archivos que podemos ejecutar son los cuadernillos de *Jupyter* para cada uno de los modelos neuronales creados.

- **ResNet:** este primer modelo se encuentra en la carpeta `/src/models/res_net`. En esta carpeta podemos ejecutar dos cuadernillos.
 1. El cuadernillo de entrenamiento se puede ejecutar usando *VSCode*. Este cuadernillo nos permite actualizar pesos de una red preentrenada. Podemos elegir uno de los datasets que se nos ofrece para entrenar la red. En este cuaderno veremos un ejemplo de uno de los datos del *dataset* antes de comenzar el entrenamiento. Se hará una comprobación de balanceo del *dataset* cargado. Cuando se entrene la red se mostrará una gráfica de costes, así como alguna comprobación del *accuracy* en los *sets* de entrenamiento, test y validación. Por último se exportará el modelo para su reutilización en el siguiente cuadernillo.
 2. Por otro lado tenemos el cuadernillo de inferencia. Con este podemos cargar el modelo entrenado en el cuadernillo anterior para ejecutar inferencia sobre una imagen o mediante *webcam*.
- **SimpleNet:** este segundo modelo se encuentra en la carpeta `/src/models/simple_net`. En esta carpeta podemos ejecutar dos cuadernillos.

1. el cuadernillo de entrenamiento: Al igual que en la red anterior, tenemos un cuaderno para entrenar el modelo. La diferencia con el anterior es que el *dataset* que se debe alimentar a este modelo, es un *dataset* de video.

Por otro lado, este modelo no usa una red preentrenada, sino que implementa una estructura CNN. El programador podrá cambiar los hiperparámetros en el archivo de configuración en `./config/consts`.

2. Muy similar al anterior cuaderno de inferencia en la red «res_net», podemos inferir a tiempo real entradas de video usando la *webcam*.

- **TwoOutNet:** este tercer modelo se encuentra en la carpeta `/src/models/two_out_net`. En esta carpeta podemos ejecutar dos cuadernillos y un *log* a tiempo real.

1. Este cuaderno de entrenamiento se parece mucho al anterior en cuanto a que aplica una red customizada de CNN y carga *dataset* de video. La diferencia está en que a ser la red más compleja, tiene un *log* a tiempo real. Por otro lado, en este archivo se muestra no sol información del tipo de signo que se debe detectar, sino de la pose de los individuos. Puedes ver un ejemplo de esto en la figura D.1.
2. Por otro lado tenemos el cuadernillo de inferencia. Con este podemos cargar el modelo entrenado en el cuadernillo anterior para ejecutar inferencia sobre una imagen o mediante *webcam*. La diferencia con el anterior está en que también se encuentra disponible la inferencia usando el *runtime* de ONNX.
3. Visualización del *log* de ejecución. Para visualizar el *log* debemos entrar en la carpeta situada en `/src/models/two_out_net/tensorboard`. Aquí podemos leer el «README» o podemos ejecutarlo con `python run.py`. Al ejecutar esto, se nos abrirá una página web que nos permitirá visualizar el *log* de ejecución. Puedes ver un ejemplo en la figura D.2.

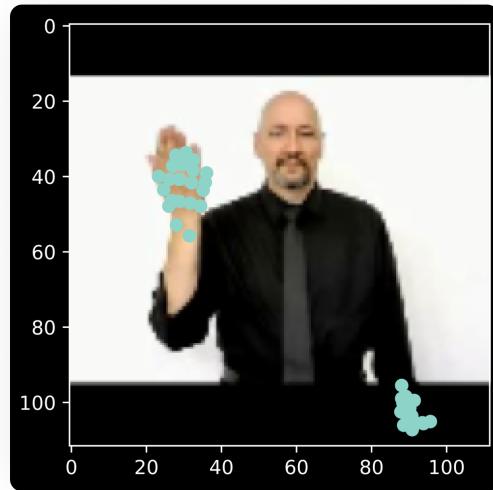


Figura D.1: Superposición de información de pose sobre *frame* estático de una instancia de entrada en formato video. Imagen del *dataset* WLASL [9]

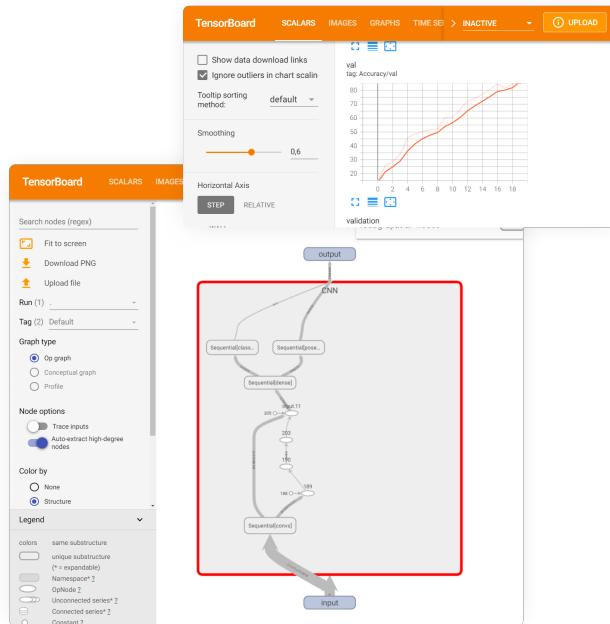


Figura D.2: Utilización de la librería Tensorboard para visualizar *logs* y estructuras de redes neuronales a tiempo real

Pruebas del sistema

Debido a la naturaleza de prueba/error del proyecto, no se han realizado pruebas unitarias ni de integración entre módulos. Sin embargo se ha usado *continuous integration* para asegurar el desarrollo de código de calidad y así evitar errores básicos que puedan derivar en otros de mayor peligro.

D.3. **Sign2Text - API**

Este es uno de los dos proyectos secundarios. Este proyecto implementa una API en Python con **FastAPI**. El objetivo del proyecto es crear una plataforma con la que los desarrolladores puedan consumir del modelo sin necesidad de tener conocimientos relacionados con *deep learning* o el framework PyTorch.

- API: La API se puede acceder a través de <https://api.sign2text.com>
- O descargar el repositorio de Github: <https://github.com/irg1008/Sign2Text-API.git>

Estructura de directorios

La estructura de este proyecto es:

- **/**: En la raíz del proyecto encontramos archivos como el «README» y archivos de dependencias de Python. Por otro lado tenemos los archivos de configuración como el `.dockerignore` y el `Dockerfile`.
- **/models**: En esta carpeta tenemos los modelos usados en el desarrollo de la API. Estos modelos solo se usan en la fase de desarrollo, ya que en producción, los modelos se consiguen haciendo una descarga a la hora de construir la imagen de docker. Esto se hace así para ahorrar tiempos de transferencias en el momento de subir o bajar una actualización de la imagen.
- **/src**: En esta carpeta encontramos el código fuente de la aplicación. El archivo `main` contiene la inicialización del servidor y el *endpoint* en el que se recibe el vídeo. El *script* `dev` se usa para arrancar la aplicación en modo desarrollo.

- **/utils**: Esta carpeta contiene archivos que no están directamente relacionados con la funcionalidad de **FastAPI**. Están son la transformación del video recibido en *frames* o la creación de una sesión de **onnxruntime**.

Manual del programador

Una vez conocemos la estructura del proyecto, vamos a ver como podemos hacer para levantar el servidor por nuestra cuenta, o en su defecto, que debemos hacer para consumir el *endpoint* ya creado y accesible en <https://api.sign2text.com>.

Entorno

El entorno de ejecución de este proyecto secundario es:

- Lenguajes: Python
- Versión: 3.9.8
- Conda: No

Las bibliotecas usadas en este proyecto se pueden ver en la tabla D.2. Todas estas librerías se detallan en el archivo **requirements.txt** para una instalación más rápida. Es muy importante mantener las versiones indicadas en D.2 para asegurar el correcto funcionamiento del proyecto. Es posible que funcione con versiones más modernas, pero se aconseja usar las versiones indicadas.

En este caso no se recomienda ningún IDE en concreto, ya que basta con tener las dependencias instaladas. La más importante es **uvicorn**. Con esta dependencia iniciamos el servidor en el puerto que indiquemos.

Compilación, instalación y ejecución del proyecto

En estas secciones vamos a ver como instalar y ejecutar las distintas partes del proyecto. Antes de nada, debemos descargarnos el código fuente de GitHub. Para hacer esto usamos el comando **git clone**. Puedes ver la url al comienzo de esta sección.

Una vez hecho, entramos en la carpeta recién descargada para comenzar el paso de instalación.

Biblioteca	Versión
fastapi	0.78.0
gdown	4.5.1
gunicorn	20.1.0
httpx	0.4.0
numpy	1.23.0
onnx	1.12.0
onnxruntime	1.11.1
Pillow	9.1.1
protobuf	3.20.1
pydantic	1.9.1
PySocks	1.7.1
python-multipart	0.0.5
PyYAML	6.0
requests	2.28.0
torch	1.11.0
torchvision	0.12.0
typing_extensions	4.2.0
uvicorn	0.18.1
watchfiles	0.15.0
websockets	10.3

Tabla D.2: Lista de librerías y versiones usadas en *Sign2Text-API*

Instalación

La instalación comienza igual que el proyecto anterior, creando un entorno virtual e instalando las dependencias:

1. Con el repositorio descargado, vamos a crear un entorno de Python virtual. De este modo no contaminamos las dependencias globales de pip. Para esto seguimos los siguientes pasos:
 - 1.1. Si no tenemos instalado con `pip install virtualenv`
 - 1.2. Ejecutamos el comando `virtualenv venv` o `python -m venv venv` para crear nuestro directorio `venv` con el entorno de Python virtual.
 - 1.3. Activamos el entorno virtual con `source venv/bin/activate` en bash o `./venv/bin/activate` en windows.

2. Una vez tenemos el entorno inicializado con el código fuente, podemos instalar las dependencias con el comando `pip install -r requirements.txt`.

Ejecución

Podemos ejecutar el servidor de dos maneras, de forma directa, como vemos a continuación o construyendo el contenedor de docker y corriendo la imagen generada.

Ejecutando de forma directa

1. Con el repositorio descargado, vamos a crear un entorno de Python virtual. De este modo no contaminamos las dependencias globales de pip. Para esto seguimos los siguientes pasos:
 - 1.1. Si no tenemos instalado con `pip install virtualenv`
 - 1.2. Ejecutamos el comando `virtualenv venv` o `python -m venv venv` para crear nuestro directorio `venv` con el entorno de Python virtual.
 - 1.3. Activamos el entorno virtual con `source venv/bin/activate` en bash o `./venv/bin/activate` en windows.
2. Una vez tenemos el entorno inicializado con el código fuente, podemos instalar las dependencias con el comando `pip install -r requirements.txt`.
3. Con las dependencias descargadas, ya podemos inicializar el servidor entrando en `/src` y ejecutando `python dev.py`. Esto nos abrirá un navegador con el puerto establecido. En este punto ya tenemos nuestra api disponible para consumir de forma local. Podemos usar cualquier cliente para probarlo, como por ejemplo `curl`.
4. Alternativamente podemos ejecutar el servidor con `uvicorn main:app -reload -port <PORT>` para desarrollo, o `uvicorn main:app -port <PORT>` para producción.

Ejecutando con docker

1. Lo primero que debemos hacer es descargar docker si no lo tenemos. Se puede hacer desde <https://docs.docker.com/get-docker/>.

2. Una vez descargado, podemos crear un contenedor con el comando `docker build -t <IMAGE_NAME> ..`. Se aconseja asignar un nombre a la imagen como «Sign2Text» o «sign2Text-API».
3. Con la imagen creada, podemos ejecutarla con `docker run -p <PORT>:<PORT> <IMAGE_NAME>`. El puerto en el lado izquierdo debe ser el asignado en el comando de *unicorn*. El puerto de la derecha marcará el puerto en el que estará disponible la API.
4. Tras esto, ya podemos acceder a la web desde el navegador.

si se desea información más extensa sobre el proceso de instalación o sobre el proceso de *deploy* del contenedor o servicio, se puede obtener en el archivo «*README*» de la raíz del proyecto.

Consumo de la API

Al ser una API cuyo intención es consumir de un modelo neuronal, se ha creado un único *endpoint*. Este *endpoint* se encarga de recibir una petición en formato «datos de formulario» [4], incluyendo un *input* de tipo «archivo». En este *input* se recibirá el archivo con el vídeo necesario para la inferencia y devolverá el resultado extraido de la clasificación.

En la figura D.3 podemos ver la estructura necesaria para hacer la petición de forma correcta. En este *endpoint* se valida que el archivo sea de tipo «mp4».

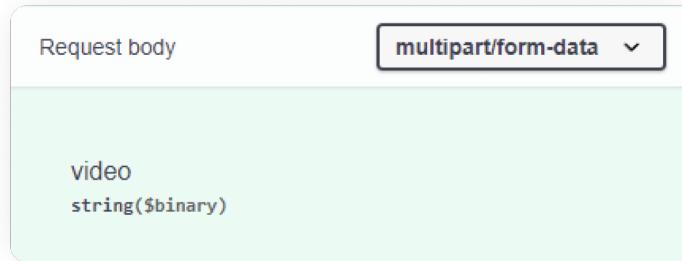


Figura D.3: Esquema del cuerpo de la petición de entrada, con formato *multipart* [4]

Por otro lado, la respuesta del servidor al cliente se puede observar en la figura D.4. El servidor nos devolverá un objeto JSON con una entrada con el valor de la etiqueta clasificada.

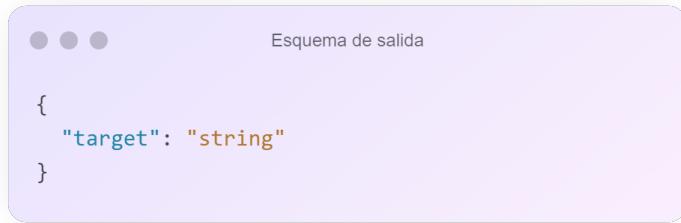


Figura D.4: Esquema de salida devuelto por el servidor. En el *target* se devolverá la etiqueta clasificada

Pruebas del sistema

En este caso, al ser un proyecto tan pequeño en número de *endpoints*, se han obviado las pruebas unitarias y de integración.

Para poder probar entonces el correcto funcionamiento del servidor se ha usado un cliente HTTP llamado Insomnia [5]. Podemos ver un ejemplo de una petición en la que enviamos un archivo «.pdf» en vez de «.mp4» en la figura D.5. El servidor comprueba correctamente esto y devuelve un objeto JSON con una entrada «detalle» con el error.

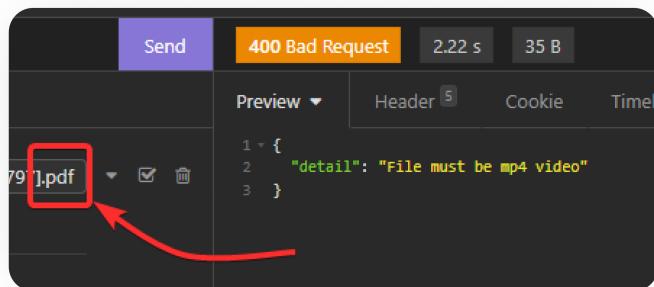


Figura D.5: Ejemplo de uso de Insomnia [5] para probar si el *endpoint* comprueba el tipo de archivo recibido. Podemos ver señalado de rojo que la extensión del archivo es «.pdf» y no «.mp4»

D.4. Sign2Text - Aplicación de cliente

Por último, el tercer repositorio del proyecto implementa una aplicación de cliente en **JavaScript** con **Astro** [7] y **Svelte** [2]. El objetivo del proyecto es crear una web fácil de usar para que los usuarios puedan probar el modelo neuronal con solo arrastrar y soltar. Veremos como funciona en el apartado «Manual de usuario». En esta sección estudiaremos la estructura del código, así como las fases de instalación y uso desde la perspectiva del desarrollador.

- El repositorio está disponible en <https://github.com/irg1008/Sign2Text-Astro.git>
- La demo web está disponible libre para todo el mundo en <https://sign2text.com>

Estructura de directorios

La estructura de este proyecto es:

- **/:** En la raíz del proyecto encontramos archivos como el «**README**» y archivos de dependencias de **JavaScript** y **NPM**. También tenemos archivos de configuración del *deployment*, así como de las librerías principales del proyecto. Por último tenemos los archivos «**.env**». En estos guardamos variables secretas que no deben ser publicadas en el repositorio.
- **/node_modules:** Esta carpeta es la que alberga las dependencias del proyecto. No está subida a Github porque el peso es muy grande y no tiene sentido hacerlo, como veremos en el apartado de instalación.
- **/public:** En esta carpeta se guardan todos los archivos que quedan públicos en una página o aplicación web. Estos suelen ser el *favicon*¹, algún archivo estático como logos y otros archivos.
- **/src:** En esta carpeta encontramos el código fuente de la aplicación. El único archivo que encontramos aquí es la declaración de tipos de las variables de entorno.

¹Logo que se ve en la pestaña de la página web, a la izquierda del título

- **/components**: Esta carpeta contiene componentes de distintos *frameworks* y librerías de JavaScript. Las librerías disponibles son **React**, **Svelte**, **Vue**, entre otras [7]. Los componentes son las unidades mínimas que componen una UI de una aplicación. Estos deben mantener la lógica interna sin depender de los demás.
- **/layout**: Un *layout* representa una estructura de la UI que se repite en todas las páginas. En este caso, el *layout* contiene un *header* y un *footer*.
- **/pages**: En esta carpeta podemos crear nuestras páginas de la aplicación. Cada archivo creado aquí generará una ruta única en la *URL*. En las páginas se pueden usar los componentes y los *layouts*.
- **/services**: En esta carpeta albergamos la lógica de negocio relativa a las peticiones HTTP al servidor (**Sign2Text-API**). Podemos observar la lógica de creación de un formulario con un video (como vimos en el proyecto del servidor), el envío a la API y la espera de la respuesta.

Manual del programador

Entorno

El entorno de ejecución de este proyecto es:

- Lenguajes: **TypeScript** (JavaScript con tipos)
- Versión: **4.7.4**

Las bibliotecas usadas en este proyecto se pueden ver en la tabla [D.3](#). Todas estas librerías se detallan en el archivo `package.json` para una instalación más rápida. Es muy importante mantener las versiones indicadas en [D.3](#) para asegurar el correcto funcionamiento del proyecto. Es posible que funcione con versiones más modernas, pero se aconseja usar las versiones indicadas.

Para la realización de este proyecto se recomienda usar **VSCode**, ya que es un editor orientado a la programación web. Además, gracias al ecosistema de extensiones, tenemos muchas herramientas que nos ayudan a desarrollar proyectos con las herramientas listadas en [D.3](#).

Biblioteca	Versión
@astrojs/react	0.1.3
@astrojs/svelte	0.1.5
@astrojs/tailwind	0.2.1
@astrojs/vercel	0.2.3
@types/react	18.0.14
astro	1.0.0-beta.53
react	18.2.0
react-dom	18.2.0
svelte	3.48.0

Tabla D.3: Lista de librerías y versiones usadas en *Sign2Text-Cliente*

Compilación, instalación y ejecución del proyecto

En estas secciones vamos a ver como instalar y ejecutar las distintas partes del proyecto. Antes de nada, debemos descargarnos el código fuente de GitHub. Para hacer esto usamos el comando `git clone`. Puedes ver la url al comienzo de esta sección.

Una vez hecho, entramos en la carpeta recién descargada para comenzar el paso de instalación.

Instalación

Comenzamos instalando las dependencias. No debemos preocuparnos de que se instalen localmente, ya que por defecto los paquetes se instalan en la carpeta `node_modules` en el propio directorio. Si queremos instalar un paquete de forma global, debemos usar el comando `npm install` con la opción `-g`.

1. Instalamos las dependencias del `package.json` con `npm install` o `npm i`:

Ejecución

- La ejecución del proyecto es muy sencilla. Lo primero que debemos hacer es ejecutar `npm run dev` desde el directorio raíz. Esto nos abrirá el navegador (por defecto en el puerto 3000).

- Una vez hecho esto, ya podemos consumir nuestra API (desarrollo o producción). La URL de la API se debe cambiar en el archivo de configuración «.env».

Compilación

- Si queremos crear una versión de producción, debemos entonces ejecutar el comando `npm run build` para compilar todo el código. Tras esto ya podemos ejecutar nuestra versión de producción.
- Para ejecutar la versión de producción ejecutamos ahora `npm run start`. Entonces se abrirá el navegador como en la versión de desarrollo. Observaremos que la aplicación funciona de forma más rápida y sin menos saltos que la versión de desarrollo.

Se puede obtener más información sobre este proyecto en el «manual del usuario», en el cual detallaremos como se puede usar la aplicación para enviar vídeos con un simple gesto.

Pruebas del sistema

No se han realizado pruebas del sistema para el front, ni de E2E ni de ningún tipo. Esto es algo que se planea hacer como tarea en el futuro. De momento, debido a la simplicidad de la aplicación, se ha optado por hacer *testing* manual.

Otro de los aspectos que no se ha probado y también entra en las líneas futuras, es la comprobación de uso en dispositivos móviles. Se ha comprobado que el sitio es *responsive* (adapta su tamaño correctamente al ancho de la pantalla) pero no se ha comprobado que un vídeo se envíe de forma correcta.

Apéndice E

Documentación de usuario

- E.1. Introducción**
- E.2. Requisitos de usuarios**
- E.3. Instalación**
- E.4. Manual del usuario**

Bibliografía

- [1] 2014.
- [2] 2019.
- [3] 2020.
- [4] Jun 2022.
- [5] 2022.
- [6] 2022.
- [7] 2022.
- [8] abdonrd. abdonrd/github-labels: A list of github labels for reuse across projects, Apr 2018.
- [9] Dongxu Li, Cristian Rodriguez, Xin Yu, and Hongdong Li. Word-level deep sign language recognition from video: A new large-scale dataset and methods comparison. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 1459–1469, 2020.
- [10] Ediciones El País. Calculadora de irpf, Jul 2022.