Grado en Ciencia de Datos
Departamento de Ingeniería Electrónica

Trabajo Fin de Grado

Estimación de Pose y sus Aplicaciones

Autor: Javier Gaviña Rueda

Tutor: Valero Laparra Pérez-Muelas

Fecha Convocatoria: Pendiente

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

Trabajo Fin de Grado

Estimación de Pose y sus Aplicaciones

Autor: Javier Gaviña Rueda

Tutor: Valero Laparra Pérez-Muelas

TRIBUNAL

PRESIDENTE/A:

VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

Declaración de autoría:

Yo, Javier Gaviña Rueda, declaro la autoría del Trabajo Fin de Grado titulado "**Estimación de Pose y sus Aplicaciones**" y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual. El material no original que figura en este trabajo ha sido atribuido a sus legítimos autores.

Valencia, 11 de septiembre de 2022

Fdo: Javier Gaviña Rueda

Resumen:

El TFG presentado está focalizado en una rama de la Inteligencia Artificial conocida como visión por computador, con una amplia gama de aplicaciones en campos como medicina, fisioterapia, automatización industrial, robótica, etc.

En el TFG trataremos abordamos campos de estudio de la visión por computador como la detección de objetos y la segmentación.

Para ello describiremos varios tipos de arquitecturas comunmente empleadas para este tipo de problemas y de librerías avanzadas en este campo como OpenPose, Detectron2 y del modelo MoveNet de la API de Tensorflow.
Asimismo, realizaremos una descripción del tipo de tecnologías hardware necesarias para afrontar el problema.

Finalmente, profundizaremos en la estimación de pose y en como aplicarla para resolver dos casos prácticos estrechamente relacionadas con el campo de la fisioterapia. Para cada uno de los casos prácticos, explicaremos que metodologías han sido empleadas para afrontar el problema, así como resultados obtenidos

Abstract:

The current FDP is focused in a branch of the Artificial Intelligence known as Computer Vision, with a wide range of possibilities in fields such as medicine, phisiotherapy, industrial automatization, robotics, etc.

In the paper, we'll try to deal with fields of study of computer vision like object detection and segmentation.

For this, we will describe several types of architectures that are commonly used for this type of problems, and advanced libraries used in this field as OpenPose, Detectron, and the Tensorflow API MoveNet mode.
In addition, we'll describe the type of Hardware technologies needed to deal with the problem.

Finally, we will delve deeper into pose estimation and how to apply it to solve two practical cases closely related with the field of phisioterapy. For each of the practical cases, we'll explain which methodologies has been used to deal the problem, as well as the results obtained

Resum:

El TFG presentat està focalitzat en una branca de la Intel·ligència Artificial coneguda com a visió per ordinador amb una àmplia gamma d'aplicacions a camps com medicina, fisioteràpia, automatització industrial, robòtica, etc

En l'article tractarem d'abordar camps d'estudi de la visió per computador com la detecció d'objectes i la segmentació.

Per fer-ho, descriurem diversos tipus d'arquitectures emprades comunament per aquest tipus de problemes i de llibreries avançades en aquest camp com OpenPose, Detectron2 i del model MoveNet de l'API de Tensorflow.
Així mateix, realitzarem una descripció dels tipus de tecnologies Hardware necessàries per afrontar el problema.

Finalment, aprofundirem en l'estimació de pose i com aplicar-la per a resoldre dos casos pràctics estretament relacionades amb el camp de la fisioteràpia. Per a cadascú del casos pràctics, explicarem quines metodologies han sigut emprades per afrontar el problema, així com els resultats obtinguts

Agradecimientos:

En primer lugar quiero agradecer a todos aquellos que me han apoyado durante todos estos años. Entre ellos a mis padres por poder subvencionarme en los estudios y haberme apoyado desde el principio.
Por otro lado, quisiera agradecer a compañeros de clase como Adrián Salcedo Puche, Moisés Ibáñez Henein y Carlos Nácher Collado, por haberme acompañado cada dia en la biblioteca y habernos ayudado mutuamente a afrontar cada asignatura y cada problema.
Quisiera agradecer también a Jose Oriol Moreno Usó por haberme ayudado con la instalación de la TPU, sin él no hubiera sido posible poder implementarla.
Desde mi experiencia realizando prácticas en el IDAL, quisiera agradecerle a Regino Barranquero Cardeñosa por ayudarme con ciertos problemas y, sobretodo, por instalarme tensorflow con GPU, lo cual ha ayudado bastante a poder realizar el trabajo programando en local. Agradecerle también a Yasser Alakhdar por proporcionar los videos con las métricas extraidas con un sensor de los corredores de cinta, a todos los que se prestaron voluntarios para realizar la prueba y, finalmente, a Emilio Soria Olivas por ser él quien me asignó y motivó a realizar el proyecto de estimación de pose en corredores de cinta, guiarme con aspectos básicos y prestarme la TPU.
Respecto a los docentes que he tenido, quisiera agradecer a todos los profesores que nos han aportado conocimientos fundamentales en machine learning como Jose David Martín Guerrero, Valero Laparra Pérez-Muelas, Antonio José Serrano, Emilio Soria Olivas, Joan Vila Francés, Francisco Martinez Gil.
Finalmente, quisiera agradecer a Valero Laparra Pérez-Muelas por tutorizarme con el proyecto, enviarme papers, dejarme sus apuntes del drive de Keras y, sobretodo, por la paciencia que ha tenido conmigo.

Índice general

Capítulo 1

Introducción

1.1. Objetivos

El objetivo de este trabajo de fin de grado, se enfoca principalmente en describir en que consiste la estimación de pose y en mostrar como se logran afrontar los dos casos prácticos de estimación de pose: la extracción de métricas de corredores de cinta y la corrección de la pose.
Para ello, se explican en los capítulos anteriores a los de estimación de pose las tecnologías necesarias para afrontar el problema, junto con la definición de arquitecturas de Deep Learning empleadas en la detección de objetos y segmentación.

De manera esquematizada, estos serían los objetivos más importantes del artículo:

- Describir las arquitecturas más empleadas en la Detección de objetos y segmentación
- Describir las librerías OpenPose, Detectron2 y el modelo MoveNet de la API de Tensorflow. Explicar porque utilizaremos MoveNet
- Explicar diferencias entre CPU, GPU y TPU
- Describir en qué consiste el dispositivo Google Coral TPU y mostrar una guia de como implementarlo para Tensorflow
- Casos Prácticos Estimación de Pose: obtener métricas de corredores en cinta

1.2. Motivación

Una de las motivaciones del TFG es dar a conocer algunos de los métodos empleados en la Detección de Objetos y en la segmentación, así como los modelos de estimación de pose.

Asimismo, facilitar a otros usuarios a la instalación del dispositivo Google Coral TPU.

Por último, en lo que se refiere a los casos prácticos en Estimación de Pose, el caso expuesto tiene un motivo clave para hacer uso de estos modelos de estimación. Consiste en la **estimación de métricas en corredores de cinta** y tiene como finalidad poder sustituir el uso de sensores que midan las métricas por solamente una cámara con un algortimo de visión por computador, con el objetivo de reducir costes económicos.

1.3. Organización del TFG

Para explicar con detalle que es exactamente la estimación de pose y profundizar en la aplicación práctica que se ha investigado, el artículo consta de 3 bloques distintos para contextualizar elementos básicos del problema.

El **primer bloque** está formado por el *Capítulo 2*.

En este primer apartado, se introduce que es la detección y/o segmentación de objetos, a la vez que se describen brevemente distintas arquitecturas empleadas en ese campo de estudio bastante comunes.

El **segundo bloque** se encuentra en el *Capítulo 3*.

En este punto, pasamos a definir componentes Hardware básicos con los que se puede afrontar este problema y introducimos el concepto de la TPU (*Tensor-Processor-Unit*), seguido de una guía rápida de instalación del dispositivo Google Coral TPU y para emplearlo con Tensorflow.
Con dicho dispositivo, implementaremos las aplicaciones de la estimación de pose descritas en el siguiente bloque

El **tercer bloque** forma parte de los *Capítulos 4 y 5*

En este último apartado, se profundiza más en el paradigma de la estimación de pose y las librerías que nos van a permitir realizar los modelos de estimación de pose. Posteriormente se expone un caso de estudio aplicado en la que exponemos como resolver el problema de la estimación de métricas en corredores de cinta.
Para este problema se realizará una breve introducción de en qué consiste y las metodologías aplicadas en el desarrollo.

Capítulo 2

Arquitecturas en detección y/o segmentación de objetos

La detección y segmentación de objetos es un campo interdisciplinario que ha estado ganando una gran cantidad de atracción en los últimos años (desde las CNN). Este campo ayuda en diversos problemas como la estimación de poses, detección de vehículos, vigilancia, etc. Este tipo de problema consiste en poder proporcionar tanto la clasificación de un objeto, como su ubicación espacial o descriptores locales necesarios para identificar el objeto.Una ligera aproximación para resolver este problema podría ser tomar diferentes regiones de interés de la imagen y utilizar una CNN para clasificar la presencia de objetos en esa región. Sin embargo, los objetos podrían tener diferentes localizaciones espaciales y/o descriptores. Para ello, se desarrollaron una serie de algoritmos capaces de localizar varias regiones de interés en una imagen y de proporcionar las ubicaciones espaciales necesarias para localizar el tipo de objeto

2.1. R-CNN

Las R-CNN, conocidas como *Region Convolutional Neural Networks* [1] fueron presentadas por *Ross Girshick* quien propuso un método que aplicaba un algoritmo de búsqueda selectiva (*Selective Search*) [2] para extraer 2000 regiones conocidas como *región proposals*

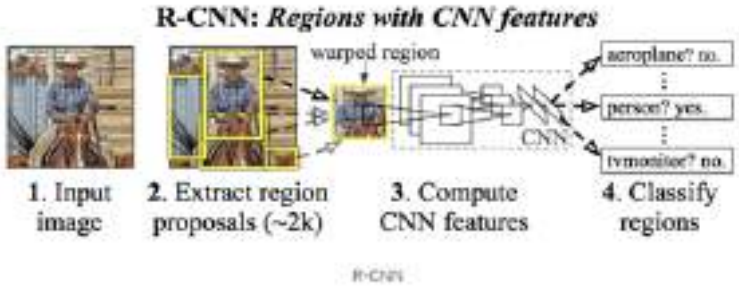

Figura 2.1: Pasos de una red convolucional basada en regiones [1]

Una vez se extraen todas las Region Proposals con el algoritmo de búsqueda selectiva [2], el algoritmo R-CNN [1] aprende a claisficar la presencia del objeto dentro de la región candidata y a predecir lo que conocemos como *Bounding-Box*.

Esto lo hace con 3 componentes principales:

- **CNN**: actúa como un extractor de características de cada región
- **Fully-Connected**: obtiene las coordenadas espaciales descriptoras de la imagen (*BoundingBox Regressor*)
- **SVM**: clasifica la presencia del objeto dentro de la región de propuesta candidata reemplazando la clasificación *softmax*, aprendida por *fine-tuning*

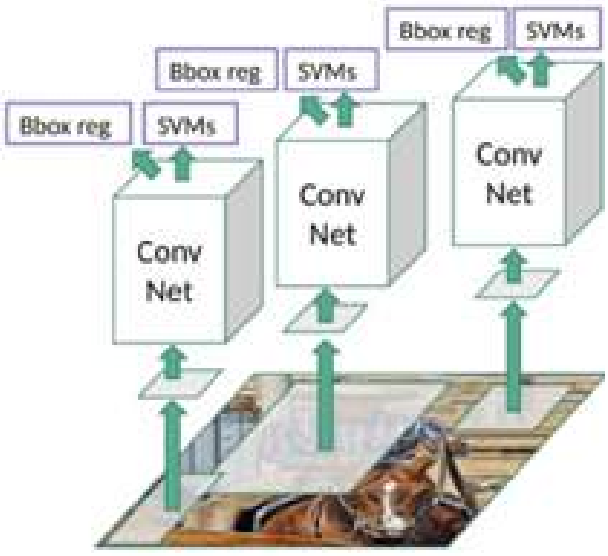

Figura 2.2: Arquitectura RCNN [3]

Sin embargo, presenta un problema en cuanto al tiempo de ejecución, dado que tiene que aplicar el algoritmo de *búsqueda selectiva* [2] para cada imagen para encontrar las 2000 regiones propuestas. Lo que conlleva unos 47 segundos por imagen

2.2. Fast R-CNN

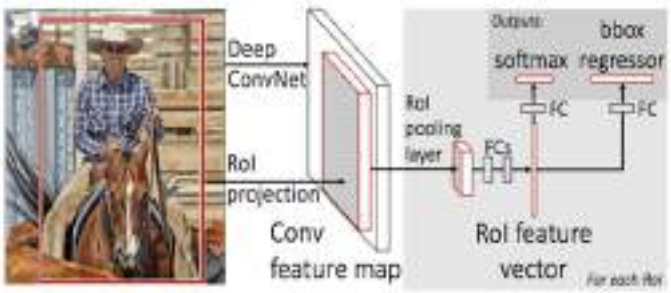

Figura 2.3: Arquitectura Fast-RCNN [4]

Este método propuesto por el mismo autor que el R-CNN [1] original.
La arquitectura 2.3 consiste en los siguientes pasos:

1. Se emplea el algoritmo de búsqueda selectiva [2] en la imagen de entrada para obtener un conjunto de *proposal regions*
2. Se aplican una serie de capas convolucionales y de capas *MaxPooling* para obtener el mapa de características (*feature map*)
3. Con el *feature map* se aplica *MaxPooling* para obtener las regiones de interés (*RoI*) de cada objeto
4. Los RoI sirven de entrada a una serie de capas *fully-connected* para obtener un vector de características de la región de interés
5. Finalmente, una rama aplica otra *fully-connected* para aplicar la clasificación *softmax* del objeto y por otro lado, otra rama emplea otra *FC* para realizar la ***regresión de la Bounding-Box***

2.3. Faster R-CNN

Esta arquitectura fue planteada por el autor *Shaoquing Ren*, donde consigue modificar la arquitectura Fast R-CNN [4] para reducir el tiempo de ejecución.

Para ello, lo primero que se plantea es sustituir el algoritmo de *Selective Search* [2] (lento y de alto coste computacional), de modo que son las propias CNN quienes aprenden las *Region Proposals*

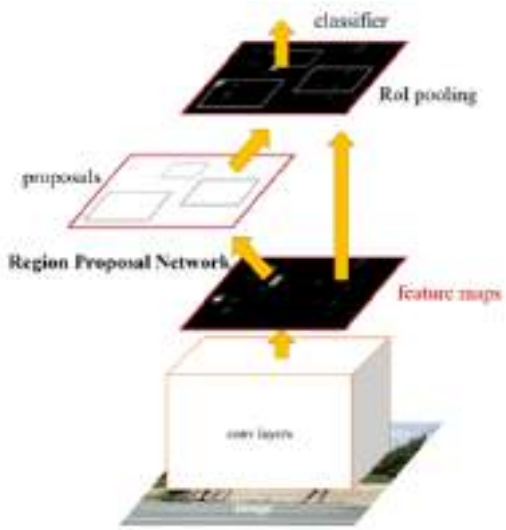

Figura 2.4: Arquitectura del Faster-RCNN [5]

La arquitectura toma una imagen como entrada y le aplica una serie de capas convolucionales profundas para extraer mapas de características (*feature maps*).

A partir de las *feature maps*, otra rama de la red predice las *región proposals*, cuyo tamaño es reescalado usando capas pooling de *RoI*, las cuales son usadas posteriormente para clasificar el objeto dentro de cada región propuesta, al mismo tiempo que proporciona las coordenadas espaciales de los *BoundingBoxes*.

2.4. Masked R-CNN

Esta arquitectura fue desarrollada por Facebook para implementarla en una librería de visión por computador conocida como Detectron (actualmente Detectron2), por el autor *Ross Girshick*.

Parte del *Faster R-CNN* [5], la cual implementa *BBox Regression* y *Object Classification*, con la adición de implementar otra rama en la arquitectura para predecir la máscara de un objeto en paralelo, empleando la ya existente rama de *bounding box regression*

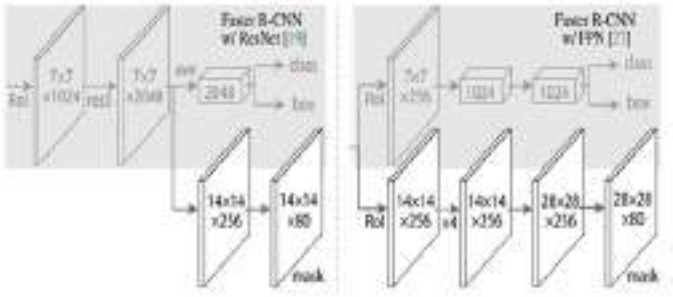

Figura 2.5: Arquitectura Masked RCNN [6], la rama superior para la regresión bbox y clasificación de objeto, y la inferior para la estimación de la segmentación como imagen binaria

Se parte de las RPN (*Region Proposal Networks*) [7], desde donde se obtienen las RoI de cada objeto, con ello se emplean una serie de capas convolucionales hasta finalmente obtener por cada RoI una salida de una máscara binaria.

Capítulo 3

Comparativa de tipos de Hardware para la inferencia

El hardware del ordenador es el soporte físico de un computador necesario para algún tipo de necesidad básica, además incluye tanto soporte interno (HDD, SDD, RAM, ...) como externo (periféricos). El elemento de hardware más importante en un ordenador convencional es lo que se conoce como unidad central de procesamiento o *CPU*, encargada principalmente de la ejecución de procesos y la interpretación de instrucciones del programa. En el campo del machine learning, la *CPU* era la única herramienta de procesamiento del ordenador para realizar entrenamiento e inferencia sobre modelos, sin embargo el principal motivo por el avanze del machine learning fue el uso de nuevos elementos hardware para incrementar la velocidad de cálculo tanto en entrenamiento como en inferencia, entre ellos el uso de la unidad de procesamiento gráfica (*GPU*) y la unidad de procesamiento de tensores (*TPU*). La ventaja que tienen algunos de los distintos modelos de estimación de pose es que a la hora de realizar **inferencia** (usar modelos con pesos ya entrenados para la obtención directa del resultado), ya tiene implementados pesos para cada una de estas tres herramientas: *CPU*, *GPU* y *TPU*, en este último caso, para el uso de la Google Coral TPU, herramienta con la que finalmente realizaremos el caso práctico de estimación de pose.

A continuación, en este capítulo trataremos de explicar de manera resumida en que consiste cada una de esas tres herramientas y como estas se diferencian.

3.1. CPU

La **CPU** controla el flujo de datos y gobierna la secuencia de acciones dentro de todo el sistema. Para poder sincronizar las operaciones hace uso de un reloj, con lo que sabe la velocidad de cada proceso y puede calcular la frecuencia de reloj, sin embargo la frecuencia de reloj está limitada por la tecnología de la CPU.

De manera resumida, podemos reflejar el diagrama de funcionamiento de la CPU en base a dos componentes principales:

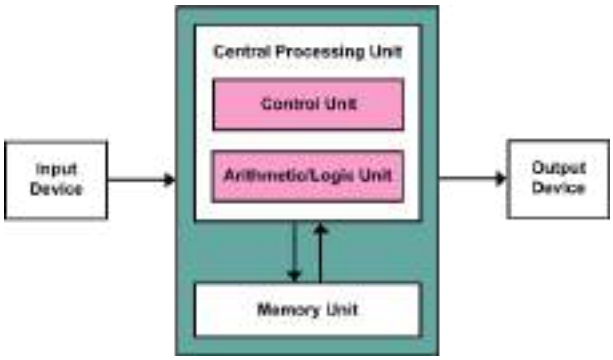

Figura 3.1: Diagrama simplificado arquitectura CPU [8]

- **ALU**: se trata de una unidad aritmético-lógica, es la encargada en la CPU de llevar a cabo las operaciones aritméticas (sumas, restas,...) y las operaciones lógicas (and, or, xor,...). Por lo general, la ALU tiene acceso directo de entrada y salida al controlador del procesador de memoria principal (RAM), donde las entradas y salidas fluyen por un camino electrónico, conocido como bus.
- **CU**: es la unidad de control de la CPU. Se encarga de dirigir las operaciones del procesador, mediante el uso del timing y de señales de control manejan el flujo de los datos entre la CPU y otros dispositivos.

La principal diferencia de la *CPU*, frente a la *GPU*, consiste en que está optimizada para minimizar la **latencia**, la cual refleja el tiempo requerido para completar una tarea. En las CPUs, gracias al *pipelining* se puede llegar a incrementar la frecuencia de reloj, lo que permite ejecutar varias operaciones simultameante y aumenta la capacidad computacional. Por lo tanto, en las CPUs se puede llegar a lograr una alta velocidad en ejecuciones secuenciales, permitiendo cierto nivel de paralelismo de varios cores del procesador. Por lo que si tenemos un algoritmo secuencial o paralelizable con un bajo grado (de 2 a 8 hilos), la CPU puede ser la mejor opción.

3.2. GPU

La *GPU* es un procesador gráfico dedicado al procesamiento de gráficos y las operaciones matriciales, y el motivo de su gran éxito es principalmente al uso de videojuegos o en aplicaciones 3D, en operaciones como el renderizado. Es por ello, que la GPU tiene implementada operaciones gráficas primitivas optimizadas para el procesamiento gráfico, con el objetivo de complementar a la *CPU* liberándole de cálculos muy intensivos. Una de las características de la GPU es que puede ser programable con la ayuda de APIs como *OpenGL*, *DirectX* y *NVIDIA*, la cual es la más empleada actualmente.Dentro del ámbito del Deep Learning, tuvo especial éxito con la implementación del *General Purpose-GPU* (**GPGPU**), que logra proporcionar aspectos de cálculos matriciales paralelizables, lo cual incrementa enormemente el rendimiento del entrenamiento y inferencia en redes profundas.

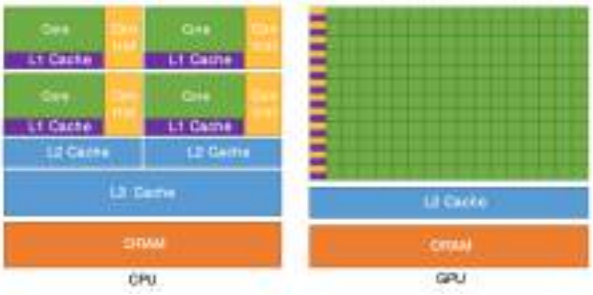

Figura 3.2: Arquitectura de una CPU frente a una GPU [9]

Tal y como podemos observar en la figura, la GPU incrementa enormemente la paralelización gracias a la elevada cantidad de cores que poseen, lo que conlleva a poder hacer uso de un número mucho más elevado de hilos. Es por ello por lo que la GPU logra maximizar el *througput* (número de operaciones por segundo), lo que la convierte adecuada para operaciones altamente paralelizables, como las redes neuronales profundas. Sin embargo, en el cálculo de algoritmos de Machine Learning simples, puede llegar a ser más óptimo el uso de la CPU.

3.3. TPU

Una **TPU** o unidad de procesamiento tensorial, trata de un *ASIC* desarrollado por Google, personalizado para el aprendizaje automático y adaptado para *Tensorflow*, aunque también es compatible con *PyTorch* [10]. Esta adaptación permite que el chip sea más tolerante a la precisión computacional reducida, lo que significa que requiere menos transitores por operación.

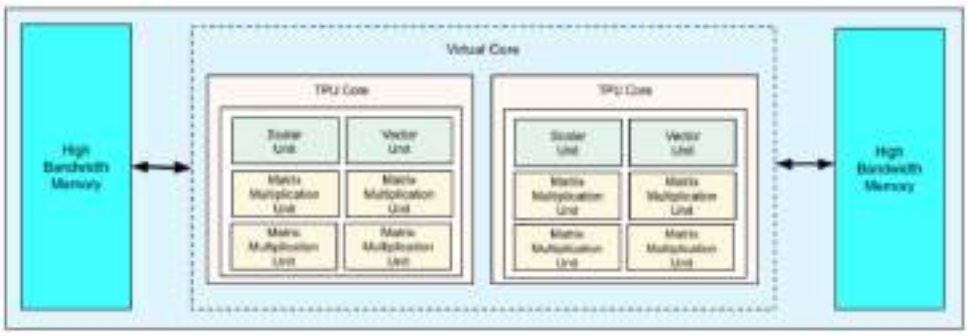

Figura 3.3: Arquitectura de una TPU (la de la imagen: TPU-v4) [11]

Si nos fijamos en la estructura 3.3, la TPU es capaz de acelerar el proceso del cálculo matricial, dado que en cada Core posee una o varias unidades multiplicativas matriciales, además de las unidades escalares y las unidades vectoriales, cada una de estas unidades especificadas para su operación. Reduciéndolo a una frase, podríamos decir que una TPU es un circuito con gran número de núcleos, muchos más que los ofrecidos por una GPU, especializados en realizar operaciones compuestas sobre tipos de datos simples. Esto les permite incrementar el rendimiento respecto a las GPUs y las CPUs [12].

En la investigación de la aplicación práctica de estimación de pose, en los corredores de cinta, hemos podido aplicar cálculos en una TPU con un dispositivo de Google conocido como **Google Coral Edge TPU**, un tipo de TPU externo que vamos a describir a continuación.

3.3.1. Introducción a la Edge TPU Coral

Edge TPU coral es un pequeño ASIC desarrollado por Google y enfocado a realizar inferencia de algoritmos de machine learning de **Tensorflow-Lite** sin la necesidad de utilizar herramientas cloud, lo que se conoce como **edge computing**.

Figura 3.4: Placa Google Coral Edge TPU [13]

Las características del Google Edge TPU 3.4 son las siguientes:

- Rendimiento: 4 billones de operaciones por segundo (TOPS).
- 2 TOPS por watt.
- Conexión: USB 3.0 (USB 3.1 Gen 1) enchufe Tipo-C.
- Soporte en Linux, Max y Windows.
- LED: si se encuentra encendido significa que la TPU está inicializada y si se encuentra parpadeando indica que la TPU está operando.
- Se alimenta con 5 Voltios por el conector USB.
- La fuente de alimentación ha de proporcionar como mínimo 500mA a 5V.

Este tipo de placa, destaca en que es capaz de reducir en gran cantidad la latencia, siendo capaz de ejecutar modelos de visión por computador a tiempo real como el modelo de TF-Lite **MobileNetv2** a 400 frames por segundo. Además, también nos permite realizar inferencia de algoritmos ML sin la necesidad de conexión a internet. La placa necesita la descarga de los drivers **Edge TPU Runtime** y una API para trabajar con ella que, en el caso de Python, es **PyCoral**. En la descarga de Edge TPU Runtime, nos da la opción de poder incrementar la máxima frecuencia reloj, lo cual incrementará la velocidad de inferencia, pero consumirá mucha más energia, por lo que puede sobrecalentarse.

3.3.2. Instalación

Los pasos de instalación de la placa que se han seguido para el sistema operativo de Windows, y la guia de implementación en Tensorflow los podemos encontrar en la página oficial de Coral [14].

En primer lugar, son necesarias unas instalaciones previas para el caso de Windows:

1. Descarga de la última versión de: Microsoft Visual C++ 2019 redistributable.
2. Descargamos el contenido del siguiente. link y descomprimimos.
3. Instalamos Git Bash.
4. Abrimos consola de Git Bash, accedemos a la carpeta descomprimida `edgetpu_runtime` y ejecutamos el archivo **install.bat**.

   ```
   >C:/Users/javie: cd Desktop/edgetpu_runtime
   >C:/Users/javie/Desktop/edgetpu_runtime: ./install.bat
   ```

5. Reiniciamos.

Después de las instalaciones previas, realizamos los siguientes pasos:

1. Usando **Anaconda Prompt**, creamos un entorno virtual para uso exclusivo de la TPU.

   ```
   (base) C:/Users/javie>conda create -n MoveNetTPU
   ```

   Nos aparecerá el siguiente mensaje: Proceed ([y]/n)? , y tecleamos **y**

2. Activamos el entorno:

   ```
   (base) C:/Users/javie>conda activate MoveNetTPU.
   ```

3. Instalación de la librería **PyCoral 2.0**.

   ```
   pip install --extra-index-url https://google-coral.github.io/py-repo/
   pycoral~= 2,0
   ```

4. Instalación de librería **tflite_runtime** para Python.

   ```
   pip install --extra-index-url https://google-coral.github.io/py-repo/
   tflite_runtime
   ```

Finalmente, probamos a ejecutar un modelo de prueba para comprobar que funciona.

1. Descarga del código de ejemplo de Google Coral.

```
mkdir coral  cd coral
git clone https://github.com/google-coral/pycoral.git
cd pycoral
```

2. Descargamos los datos y el modelo de ejemplo.

```
bash examples/install_requirements.sh classify_image.py
```

3. Ejecutamos el clasificador de imagenes con la imagen del loro.

```
python3 examples/classify_image.py \
--model test_data/mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite\
--labels test_data/inat_bird_labels.txt \
--input test_data/parrot.jpg
```

Si no hemos obtenido el error *Failed to load delegate edgetpu.dll* y vemos que la luz de la Coral parpadea, significará que hemos logrado instalar la placa con sus corrientes drivers de manera correcta y implementarla en Python con éxito.

3.3.3. Implementación y creación modelos con Edge TPU Coral

Edge TPU es capaz de entrenar y hacer inferencia con redes neuronales profundas como CNNs. Solo admite modelos de **Tensorflow Lite** cuyas entradas sean tensores cuantificados a 8 bits y luego compilados específicamente para Edge TPU

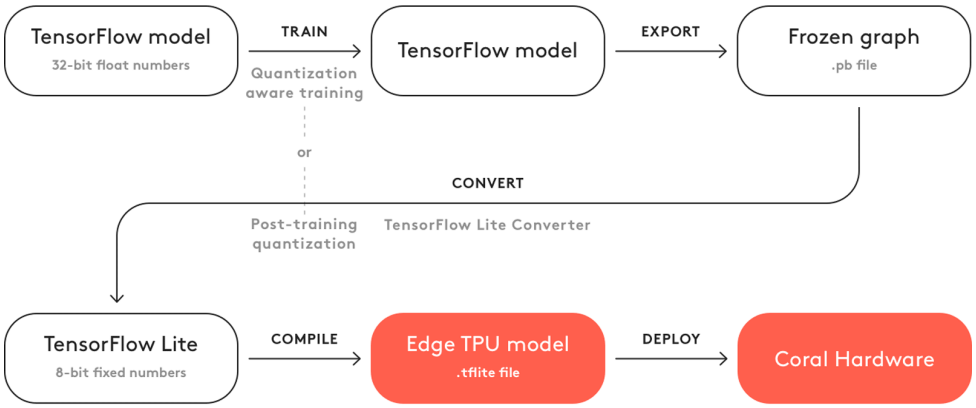

Figura 3.5: Flujo de creación de un modelo para Edge TPU [15]

Los pasos, tal y como observamo en la figura 3.5, serían cuantificar un modelo de tensorflow a 8 bits y cuantificarlo, una vez entrenado se exporta como modelo Tensorflow en un grafo congelado (.pb). Lo siguiente sería exportarlo a Tensorflow Lite (.tflite), finalmente, ya podríamos realizar inferencia con Edge TPU usando los pesos en formato **tflite**.

Edge TPU tiene definidas una serie de operaciones las cuales están optimizadas para su implementación, algunas de ellas son las siguientes:

- Add, AveraPool2D, MaxPool2D, Conv2D, PReLU
- Concatenation, Mean, Maximum, Minimum, Rsqrt, Tanh
- DepthWiseConv2D, FullyConnected, L2Normalization
- ExpandDims, Reshape, Transpose, TransposeConv, SoftMax
- Logistic, LSTM, Mul, Pack, Slice, Split

Para aprovechar al máximo la operabilidad de Edge TPU, los modelos han de cumplir con los siguientes requisitos:

- Parámetros de tensores cuantificados (int8 o uint8).
- Los tamaños de tensores han de ser constantes mientras se compila.
- Parámetros de los modelos (como los sesgos en los tensores) han de ser constantes mientras se compila.
- Los tensores han de ser de 1, 2 o 3 dimensines, si la dimensión es mayor a 3, solo las 3 dimensiones más internas pueden tener un tamaño mayor a 1.
- El modelo utiliza solamente las operaciones soportadas por Edge TPU.

El incumplimiento de alguna de estas condiciones puede hacer que, o bien no funcionen con Edge TPU, o bien el rendimiento empeore notablemente.

Capítulo 4

Modelos de Estimación de Pose

La estimación de pose es un campo de estudio de visión por computador focalizado a inferir la pose de una persona o un objeto en una imagen. Se trata en un problema de determinar la posición espacial en una imagen (en coordenadas xy) en relación con una persona u objeto determinado. Esta técnica se realiza identificando, ubicando y rastreando una cantidad de Keypoints que definan a la persona o al objeto. Para los objetos, esto podría ser esquinas u otras características específicas *Feauture Extraction*, mientras que para los humanos, estos representarían partes del cuerpo y articulaciones importantes para poder definir un esqueleto como un tobillo, un hombro o una muñeca. El entrenamiento de los modelos de estimación de pose se requiere un etiquetado previo de cada Keypoint de interés en cada imagen, de modo que la función de coste sería algo así como:

$$arg\min_{\theta} \sum_{(x,y)\in D_N} \sum_{i=1}^{k} \left|\left| y_i - \psi_i(x;\theta) \right|\right|_2^2$$

Dicha función consiste en el error cuadrático medio de las coordenadas (x,y) de la etiqueta a las coordenadas predecidas por el modelo.
En la aplicación del caso práctico, aplicamos modelos de estimación de pose a videos de corredores de cinta con el objetivo de quedarnos con la ubicación espacial (en un instante o frame determinado) y temporal (evolución de la posición a lo largo del video) de determinadas partes de interés del cuerpo humano como son: el tobillo izquierdo y derecho, rodillas, pelvis,... con el objetivo de poder realizar una estimación con técnicas de procesado de series temporales univariantes para calcular métricas de interés en los corredores de cinta. Para ello, se ha investigado varios frameworks que implementan modelos de estimación de pose como **OpenPose** y **Detectron2**, y un modelo de la API de Tensorflow, TFLite, conocida como **MoveNet**, donde la API ya proporciona los pesos para que el modelo pueda realizar inferencia con **CPU**, **GPU** y con el dispositivo **Edge TPU**

4.1. OpenPose

Es una librería, originalmente escrita en C++ y en Caffe y actualmente con pequeñas implementaciones en Python, basada en la detección a tiempo real de múltiples personas (Real-time multiple-person detection). Se puede ejecutar en diferentes plataformas, incluyendo Ubuntu, Windows, Mac OSX y sistemas integrados (p.e. NVIDIA Tegra TX2). También proporciona soporte para diferentes tipos de hardware, como GPU CUDA, GPU OpenCL y dispositivos only-CPU.

OpenPose consiste en tres bloques diferentes: detección de cuerpo+pies, detección de manos y detección de rostro. El bloque principal es el detector de keypoints de cuerpo y pies combinados. De manera alternativa, también puede utilizar modelos de solamente el cuerpo que han sido entrenados con los datasets de COCO y MPII. La librería emplea un método no paramétrico conocido *Part Affinity Fields* (PAF: Campos de afinidad de piezas), con lo que logra aprender a asociar partes del cuerpo con individuos en la imagen, a la vez que puede obtener un score o puntuación para cada estimación de pose, con ello, la librería es capaz de detectar poses 2D de múltiples personas, incluidos keypoints del cuerpo, el pie, la mano y la cara. OpenPose emplea un método para le estimación de pose conocida como Top-Down, donde en primer lugar se aplica un detector de objetos para detectar a las personas, y la ubicación detectada con un BoundingBox es utilizada para obtener la estimación de pose. Este método puede obtener scores por cada par de PAFs sin la necesidad de un paso de entrenamiento más adicional, con resultados de alta calidad con rendimiento a tiempo real en la estimación de pose.

Figura 4.1: Pipeline de estimación de pose de OpenPose [16]

Los pasos del algoritmo 4.1 son los siguientes:

1. Se toma la imagen completa como entrada para una CNN para la predicción conjunta (*jointly prediction*).
2. Se utilizan *Part Confidence Maps* para la detección de las partes del cuerpo.
3. Emplea *Part Affinity Fields* para la asociación de partes.
4. Mediante los *Part Confidence Maps* y las PAFs, se extraen un conjunto de coincidencias bipartidas (bipartite matchings) para asociar candidatos a partes del cuerpo 4.2.
5. Se ensamblan los candidatos en poses de cuerpo completo para todas las personas de la imagen.

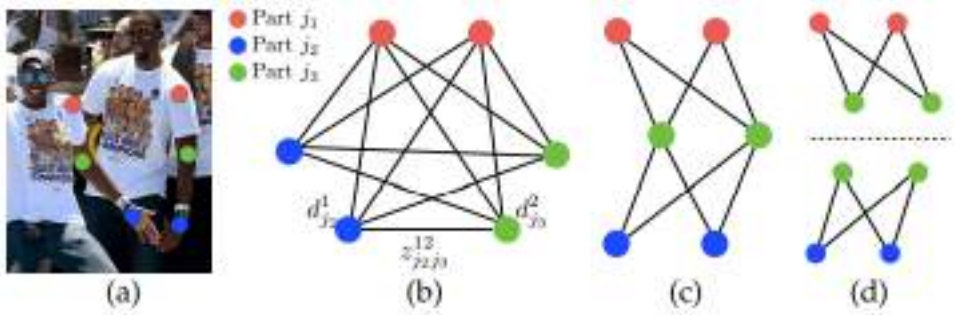

Figura 4.2: Grafos encontrados. (a) Imagen Original con las detecciones de partes. (b) grafo K-partido. (c) Estructura de arbol. (d) Un conjunto de grafos bipartidos [16]

El sistema toma como entrada una imagen en color de tamaño $w \times h$ y produce las ubicaciones 2D de los puntos clave anatómicos para cada persona en la imagen. Primero, una red feedforward predice un conjunto de mapas de confianza 2D **S** de ubicaciones de partes del cuerpo y un conjunto de campos vectoriales 2D **L** de campos de afinidad de partes (PAF), que codifican el grado de asociación entre las partes. OpenPose emplea un tipo de arquitectura convolucional para asociar las partes del cuerpo con los individuos conocida como **multi-stage CNN** 4.3

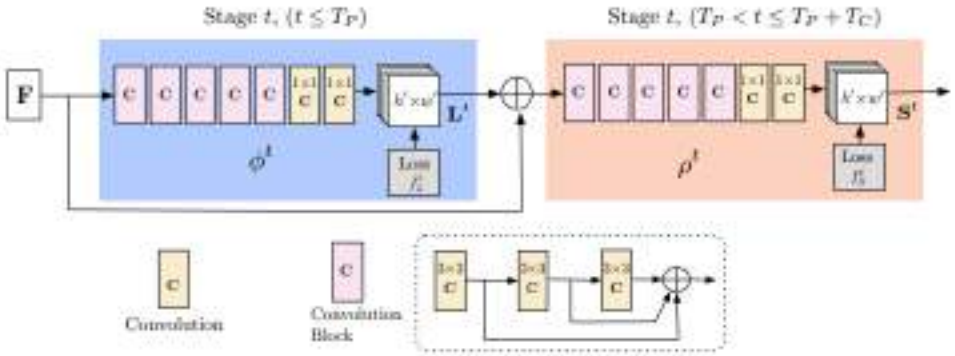

Figura 4.3: Arquitectura de un multi-stage CNN

La arquitectura 4.2 utiliza bloques convolucionales de 3 capas de kernels 3x3, las cuales son concatenadas al final de cada stage. En los primeros stages se predicen los PAFs L^t, mientras que en los últimos se predicen los *Confidence Maps* S^t. Las predicciones de cada stage y sus correspondientes características de la imagen se concatenan con su stage consecuente. El conjunto $\mathbf{S} = (S_1, S_2, \cdots S_j)$ tiene J mapas de confianza, uno por parte del cuerpo, donde $S_j \in \mathbb{R}^{wxh}, j \in \{1 \cdots, J\}$. El conjunto $\mathbf{L} = (L_1, L_2, \cdots, L_C)$ tiene C campos de vectores, uno por extremidad, donde $L_C \in \mathbb{R}^{wxhx2}, c \in \{1, 2, \cdots, C\}$, donde cada pareja de partes es considerada como extremidad, pero no todos los pares son extremidades humanas reales. Cada localización de la imagen en L_c codifica un vector de dos dimensiones. Finalmente, los mapas de confianza y los PAFs son analizados por *Greedy Inference* para obtener los Keypoints en 2D para cada persona de la imagen.

La imagen es analizada por una CNN (inicializada con las 10 primeras capas de **VGG-19** y ajustada por *fine tunning*), generando un conjunto de mapas de características **F** que se introducen como entrada en la **multi-stage CNN** en la primera etapa. En este punto, la red produce un conjunto de campos de afinidad de partes (PAFs) $L^1 = \phi^1(F)$, donde ϕ^1 se refiere a la inferencia obtenida por las CNNs en el Stage 1. En cada estado, las predicciones de los estados anteriores y las características de la imagen orginal F son concatenadas y utilizadas para producir predicciones refinadas.

$$L^t = \phi^t(F, L^{t-1})$$

Después de T_P iteraciones, el proceso se repite para la detección de mapas de confianza, empezando con la predicción más actualizada del PAF

$$\begin{aligned} S^{T_p} &= \rho^t(F, L^{T_P}, \forall t = T_P \\ S^t &= \rho^t(F, L^{T_P}, S^{t-1}), \forall T_P < t \leq T_P + T_C \end{aligned}$$

Donde ρ^t se refiere a la inferencia obtenida por las CNNs en el Stage t y T_C al número total de stages de *mapas de confianza*.

Finalizando, tenemos que la función de coste de las **PAFs** es la siguiente:

$$f = \sum_{t=1}^{T_P} {}_L^t + \sum_{t=T_P+1}^{T_P+T_C} f_S^t$$

Donde:

$$\begin{aligned} f_L^{t_i} &= \sum_{c=1}^{C} \sum_{p} W(p) \cdot ||L_C^{t_i}(p) - L_c^*(p)||_2^2 \\ f_S^{t_k} &= \sum_{j=1}^{J} W(p) \cdot ||S_j^{t_k}(p) - S_j^*(p)||_2^2 \end{aligned}$$

Siendo L_C^* el PAF del *groundtruth*, S_j^* el *groundtruth* del *Part Confidence Map* y **W** la máscara binaria

4.2. Detectron2

Detectron2 es un framework de última generación de FAIR (Facebook AI Research), publicado en GitHub en el siguiente link [17] para la detección y segmentación de objetos. FAIR creó este marco para proporcionar la implementación de CUDA y PyTorch de arquitecturas de redes neuronales de última generación. También proporcionan modelos preentrenados para la detección de objetos (RCNN [1], Fast-RCNN [4], Faster-RCNN [5]), de segmentación (Masked-RCNN [6]), de detección de keypoints en humanos y otros usos. Una de las desventajas de esta plataforma es que no está habilitada para emplearse en Windows y solo admite su uso para la GPU. Necesita para utilizarse un sistema Linux, GPU habilitada para CUDA, Pytorch >= 1.4 y Python 3.5, 3.6 o 3.7.

Detectron2 está orientado a la inferencia, de modo que en la librería están incorporados modelos pre-entrenados de **model zoo**, que es una plataforma de Deep-Learning que proporciona redes neuronales entrenadas previamente en el dataset COCO y MPII. Desde Detectron2 se disponen de conjuntos de modelos para cada tipo de aplicación de visión por computador, dado que tiene sus conjuntos de modelos pre-entrenados para emplearse en extracción de keypoints (la que nos interesa en estimación de pose), detección de objetos, segmentación panóptica y segmentación de instancias. Sin embargo, la librería no está solamente orientada a realizar inferencia, dado que también permite poder usar estos modelos de model zoo para entrenar, empleando las configuraciones de Detectron2 ya programadas para utilizar modelos en Pytorch. Su principal ventaja respecto a **OpenPose** [16] se encuentra justamente en la enorme variedad de aplicaciones que proporciona Detectron2 dentro de la visión por computador.

En cuanto al tipo de arquitecturas que se pueden encontrar de Detectron2, nos permite este tipo de combinaciones:

- Backbones (o extractores) de características: ResNet 18, ResNet 34, ResNet 50, VGG16, VGG19,..
- Backbones + FPN [18] (Feature Pyramid Networks)
- Red convolucional basada en regiones: RCNN, Fast-RCNN, Faster-RCNN, Masked-RCNN

.

El modelo de Detectron2 que hemos empleado para la estimación de pose es keypoint_rcnn_R_50_FPN_3x.yaml , que como su nombre indica, es un algoritmo extractor de keypoints, necesario para la estimación de pose, que utiliza como backbone la **ResNet 50** combinada con una **Feature Pyramid Network**, y como red convolucional extractora de keypoints utiliza la **RCNN**

Probamos a ejecutar un modelo de estimación de pose de **OpenPose** y el modelo ya comentado de **Detectron2** en Google Colab Pro, con una *GPU Tesla P100-PC1E-16GB*, con el objetivo de poder comparar los resultados y esto fue lo que obtuvimos:

Figura 4.4: Comparación OpenPose con Detectron2. Arquitectura de detectron empleada: keypoint_rcnn_R_50_FPN_3x

Tal y como hemos comentado en la sección anterior, si observamos la figura 4.4 vemos que detecta varios keypoints más en el pie, a la vez que parece estimar mejor la pose humana. Por otro lado, podemos observar en Detectron2 4.4 una estimación ligeramente peor que en OpenPose. Sin embargo, nos proporciona la bounding-box obtenida debido a la RCNN. Se probó a estimar 50 veces el mismo frame para estimar el tiempo de ejecución con los dos modelos y obtuvimos que OpenPose tarda 0.9 segundos/frame, mientras que el tiempo de Detectron2 es de unos 0.23 segundos/frame, dado que OpenPose está orientado a la multi-detección de pose (a más de una persona), puede que ese sea el motivo por el cual es más lento. Esto puede representar un problema a la hora de querer hacer una implementación a tiempo real, por lo que si tuviérámos que emplear uno de los dos modelos, de momento sería el modelo de Detectron2.

4.3. MoveNet

MoveNet [19] es un modelo desarrollado por la API de Tensorflow, TF Hub, con su adaptación para dispositivos como Edge TPU [15] en **TF-Lite**, con dos variantes del mismo modelo, **lightning** [20] y **thunder** [21], orientado a la detección de pose. Destaca principalmente por su rápida inferencia. El modelo detecta un total de 17 keypoints del cuerpo y está capacitado para funcionar más rápido que en tiempo real (a más de 30 fps). Es compatible con Linux, Windows y MAC y ofrece los pesos del modelo adaptados para CPU, GPU y TPU. La arquitectura de MoveNet consiste en un **MobileNetV2** [22] utilizado como extractor de características, combinado con una **Feature Pyramid Network** [18] como decoder (con strides de 4), seguido de una **CenterNet** [23] para la predicción de cabezas utilizando un post-procesado lógico propio.

MobileNetV2 es una arquitectura convolucional con alto rendimiento para emplearse en dispositivos móviles, que consiste en una estructura invertida residual donde las conexiones residuales se realizan entre las capas del cuello de botella (espacio latente). Utiliza un tipo de convoluciones conocidas como *Depthwise Convolutions* [24], un tipo de convolución especial en el que se aplica un único filtro convolucional por cada canal de colores de la imagen de entrada. En conjunto, la arquitectura de MobileNetV2 contiene la capa inicial de convolución completa con 32 filtros, seguida de 19 capas de cuello de botella residuales.

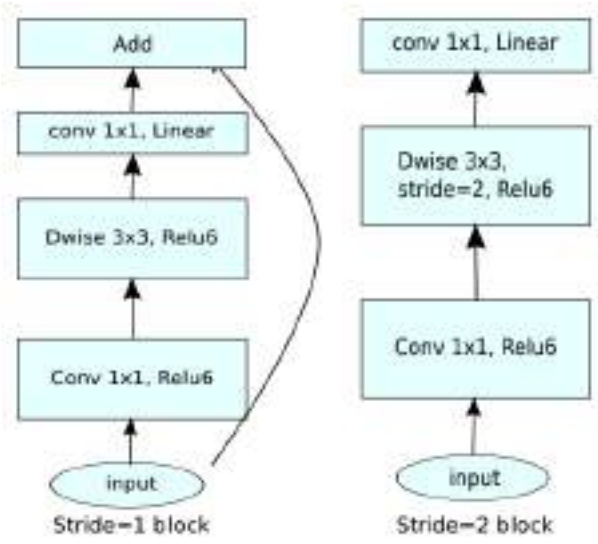

Figura 4.5: Arquitectura MobileNetV2

Por otro lado, la **Feature Pyramid Network** [18] o FPN consiste en un extractor de características que coge una imagen de escala única de un tamaño arbitrario como entrada, y predice mapas de características de tamaños proporcionales al inicial en múltiples niveles, empleando convolucionales.

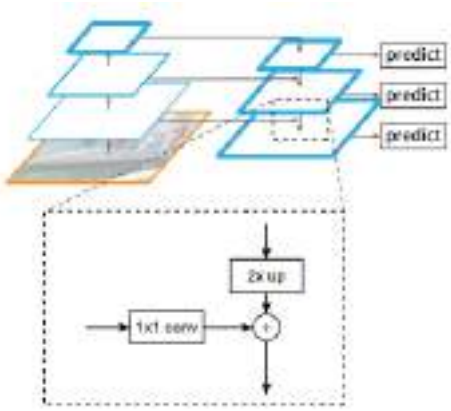

Figura 4.6: Arquitectura de una Feature Pyramid Network [18]

Destacan bastante por el hecho de que se suelen combinar con backbones como ResNet50 o VGG16, ya que también admite como entrada, mapas de características de escala única. La arquitectura se divide en dos partes conectadas por conexiones laterales de sumas de convoluciones 1x1 del bottom-up y del mapa de características resultante de incrementar por 2 el tamaño por parte del top-down, tal y como se visualiza en el zoom de la figura 4.6. El primero es el **bottom-up**, que realiza un cálculo feed-forward para obtener una jerarquía de características en varias escalas que constituyen el mapa de características. La segunda parte es el **top-down**, que obtiene características de mayor resolución mediante el muestreo superior de características de niveles piramidales más altos. Estas características se mejoran con las conexiones laterales que podemos observar en la figura 4.6.

Por último, las **CenterNet** [23] son detectores de objetos de una etapa que detectan cada objeto como un triplete, en lugar de un par, de keypoints. Utiliza dos módulos personalizados que son *cascade corner pooling* y *center pooling*, que desempeñan el papel de enriquecer la información recopilada por las esquinas superior izquierda e inferior derecha y proporcionar información más reconocible en las regiones centrales, respectivamente.

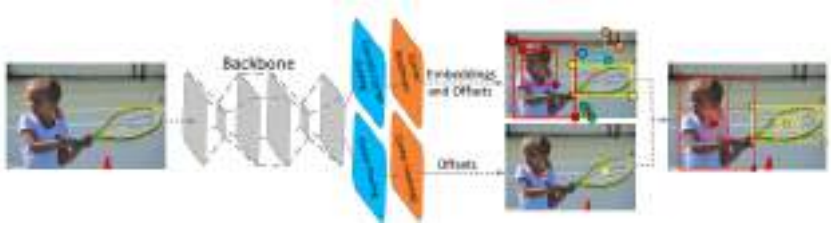

Figura 4.7: Arquitectura CenterNet

Retomando el modelo de MoveNet, hemos comentado previamente que TFHub a desarrollado el modelo con dos variantes, lightning y thunder. En cuanto a arquitectura son bastante similares aunque presentan unas pequeñas diferencias:

El tamaño de las imágenes de entrada de lightning es de 192 x 192 x 3, mientras que en thunder es de 256 x 256 x 3. Además, respecto a las *Depthwise-convolutions* [24] empleadas en la arquitectura de *MobileNetV2* [22], lightning emplea un Depth Multiplier de 1, mientras que thunder de 1.75. Siendo *Depth Multiplier* un parámetro de la Depthwise Convolution que escala el número de salidas del canal RGB. Respecto a las prestaciones que ofrece ambos modelos, lightning realiza estimaciones con peor calidad que thunder a costa de ser más rápido, de modo que el primero puede ejecutarse a tiempo real en grabaciones de más de 50fps mientras que el segundo a más de 30fps. Si realizamos una comparación de cada variante de MoveNet para CPU, GPU y TPU, podemos observar lo siguiente 4.8:

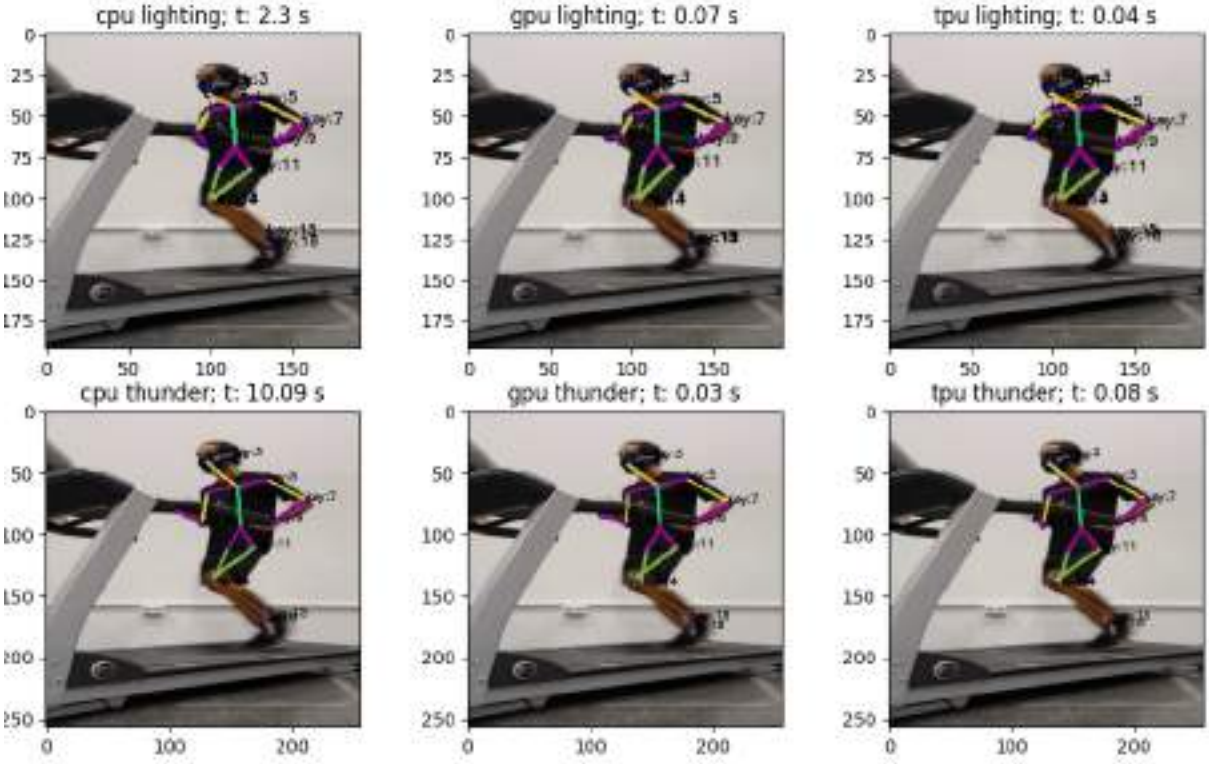

Figura 4.8: Comparación MoveNet de lightning y thunder, para CPU, GPU y TPU

Podemos ver en la figura 4.8 que para ese frame determinado, lightning tiene peor calidad a la hora de predecir algunas partes del cuerpo como la parte del brazo derecho. Comprobemos si por tiempos de ejecución, a costa de tener mayor coste computacional, si sería más rentable incrementar la calidad de la predicción con thunder 4.9:

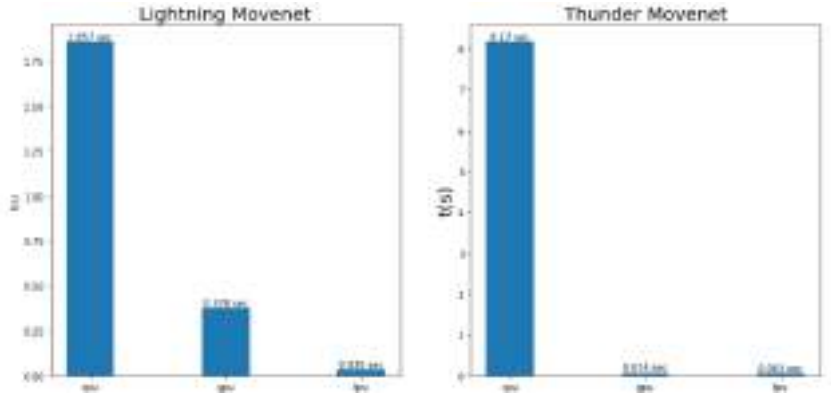

Figura 4.9: Comparativa de tiempos de ejecución de lightnig y thunder para cada tipo de tecnología

Podemos observar en la figura 4.9 que, aunque no haya mucha diferencia de tiempos de ejecución entre la GPU y la TPU en thunder, en Lightning el uso de la TPU incrementa la velocidad de ejecución 10 veces más que la GPU, por lo que utilizarla nos va a ayudar bastante en la implementación a tiempo real. No obstante, el uso de la CPU en la aplicación de tiempo real, queda completamente descartada para ambos modelos dado su enorme coste computacional. Dado que no hay unas diferencias significativas en tiempos de ejecución entre el modelo lightning y thunder 4.9, nos resulta más beneficioso utilizar thunder para incrementar la calidad de la estimación de pose.

Capítulo 5

Estimación métricas corredores en cinta

En este apartado se ha realizado una investigación sobre un caso concreto en estimación de pose, en el que se trata de extraer métricas de corredores en cinta empleadas en el campo de la Fisioterapia. El objetivo consiste en poder extraer las métricas obtenidas con un sensor mediante visión por computador, de modo que nos permite reducir el coste económico de la adquisición del sensor a la hora de realizar una monitorización del sujeto. En el estudio realizado, se logró tener una estimación de las siguientes métricas:

- Pasos Totales
- Cadencia: pasos/minuto
- Máximo ángulo de separación de piernas: medido en grados sexagesimales
- Tiempo de Contacto: tiempo en el que el sujeto está en contacto con el suelo, medido en milisegundos

Para realizar el estudio se consiguieron a 9 pacientes voluntarios y se grabaron 2 videos con 8 de ellos, uno con la velocidad de la cinta fijada a 10km/h y el otro con la velocidad fijada a 12 km/h. Con el paciente restante solo se grabó un video con la cinta a 10km/h de velocidad. El experimento consistía en poner a correr en la cinta a un sujeto y emplear una cámara en el suelo para grabar al paciente y se le colocaba en los pies un sensor para poder extraer la información de las métricas de interés. El sensor empleado es RunScribe [25] y actualmente es utilizado por entrenadores, podólogos, fisioterapeutas, laboratorios de marcha, etc. El sensor captura los datos de cada pie en cada paso además de que proporciona una vista detallada de las mecánicas de marcha de un sujeto. RunScribe cuenta, además de con los sensores 5.1, con una aplicación para el móvil y un dashboard al que se puede acceder desde la propia página web, donde se pueden consultar las mecánicas de marcha del corredor

Figura 5.1: Sensor RunScribe [25]

No obstante, el principal incoveniente del dispositivo es su elevado coste económico, encontrando unos precios de entre 599€ y 799€ en la página web oficial. De aquí surge el interés, para el caso de los corredores de cinta, en poder sustituir el uso de estos sensores por un algoritmo basado en visión por computador, donde el único coste sería el uso de la propia cámara. El programa realizado para abordar el estudio de las métricas de corredores se ha desarrollado en base a dos enfoques importantes:

- **Enfoque Off-Line**: Se indica al algoritmo la ruta de una carpeta que contiene videos de corredores en cinta y la ruta de salida de los videos. En este caso, el algoritmo te devuelve como salida los mismos videos con el esqueleto del sujeto (obtenido con la estimación de pose) pintado y con las métricas descritas previamente apareciendo en el propio video, también genera un reporte en pdf con un promediado de cada una de estas médidas por sujeto, seguida de una gráfica con la evolución de cada una de las métricas.
- **Enfoque On-Line**: El programa comienza a grabar a tiempo real, mientras muestra por pantalla al corredor de cinta, con el esqueleto pintado, y en la esquina superior derecha del video las métricas calculándose a tiempo real. Al finalizar el video genera el mismo reporte en pdf que el del enfoque Off-Line.

Para ambos enfoques, el dibujo del esqueleto y la escritura de las métricas en la imagen lo conseguimos utilizando la librería de **OpenCV**. En cuanto a con que tipo de tecnología utilizar para este caso, tal y como hemos comentado en el capítulo anterior, utilizaremos el dispositivo Edge TPU, dada su principal ventaja en lo referido al tiempo de ejecución, lo que nos permite poder utilizar el programa con el **enfoque On-Line**

A la hora de enfrentarse a la estimación cuantitativa de las mecánicas de movimiento existen dos ramas en las que abordar el estudio, tal y como vemos en la figura 5.2

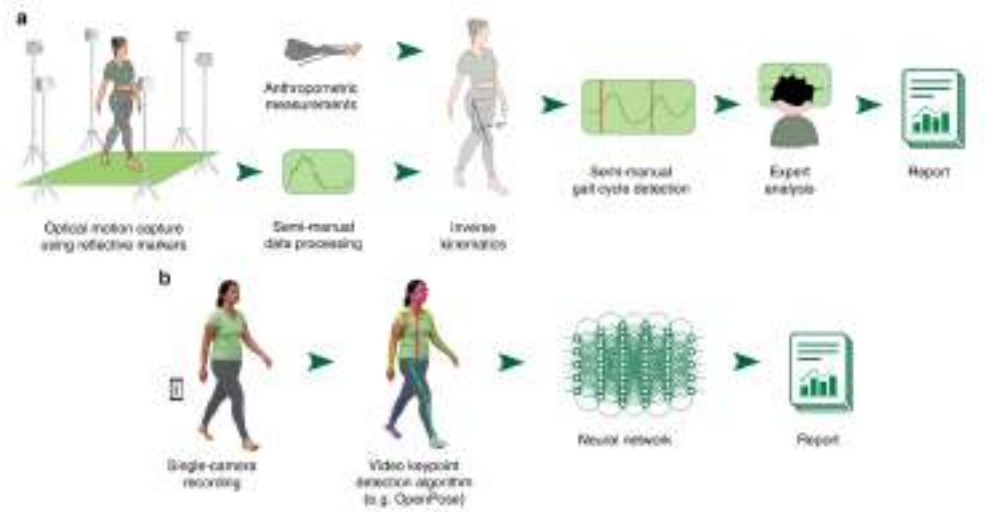

Figura 5.2: Formas de abordar la extracción de métricas de los corredores. (a) Procesado semi-manual: etiquetado manual de los puntos clave en cada frane, obtención de serie temporal y cálculo de métricas con procesado de serie temporal univariante. (b) Procesado automático: extracción de Keypoints, obtención de series temporales de cada Keypoint, uso de redes neuronales profundas que aprenderán a predecir cada una de las métricas [26]

En nuestro caso, se ha utilizado el primer método con la diferencia de que la obtención de keypoints se ha realizado con inferencia utilizando el modelo de **MoveNet**. Este método tiene como ventaja que es más determinista y que con el procesado adecuado puede obtener resultados bastante eficientes, sin embargo es un método poco generalizable. El uso de redes neuronales profundas que sean capaces de aprender a estimar las métricas, no ha podido ser posible para este caso debido a la poca cantidad de datos, de modo que la única forma de realizar un entrenamiento con éxito sería sobreentrenando nuestro pequeño conjunto de datos

5.1. Extracción de Keypoints

La salida de MoveNet 4.8, al recibir una imagen como entrada, es un array donde cada fila corresponde con una parte del cuerpo humano, de modo que:

`fila 0: nariz; fila 1: ojo izquierdo; fila 2: ojo derecho; fila 3: oreja izquierda; fila 4: oreja derecha; fila 5: hombro izquierdo; fila 6: hombro derecho; fila 7: codo izquierdo; fila 8: codo derecho; fila 9: muñeca izquierda; fila 10: muñeca derecha; fila 11: cadera izquierda; fila 12: cadera derecha; fila 13: rodilla izquierda; fila 14: rodilla derecha; fila 15: tobillo izquierdo; fila 16: tobillo derecho`

Donde cada fila consiste en un vector 2D que indica la posición del keypoint en la imagen: (x, y). Con esta información, podemos obtener una estimación de las coordenadas de keypoints de las siguientes partes del cuerpo:

- cuello $= \left(\frac{Kp_{5,x} + Kp_{6,x}}{2}, \frac{Kp_{5,y} + Kp_{6,y}}{2} \right)$; distancia media entre hombro izquierdo y derecho
- ombligo $= \left(\frac{Cuello_x + Kp_{11,x} + Kp_{12,x}}{3}, \frac{Cuello_y + Kp_{11,y} + Kp_{12,y}}{3} \right)$; distancia media entre cuello, cadera izquierda y cadera derecha
- pelvis $= \left(\frac{Kp_{11,x} + Kp_{12,x}}{2}, \frac{Kp_{11,y} + Kp_{12,y}}{2} \right)$; distancia media entre cadera izquierda y cadera derecha

Siendo Kp el array de dimensiones 17 x 2 de keypoints que nos indican cada parte del cuerpo. De este modo, podríamos guardarnos un total de 34 series temporales (17 partes del cuerpo x 2 ejes) si nos guardaramos la posición espacial de cada keypoint a lo largo de cada frame del video.

La evolución de un Keypoint a lo largo del video, lo podemos representar como en la siguiente figura 5.3

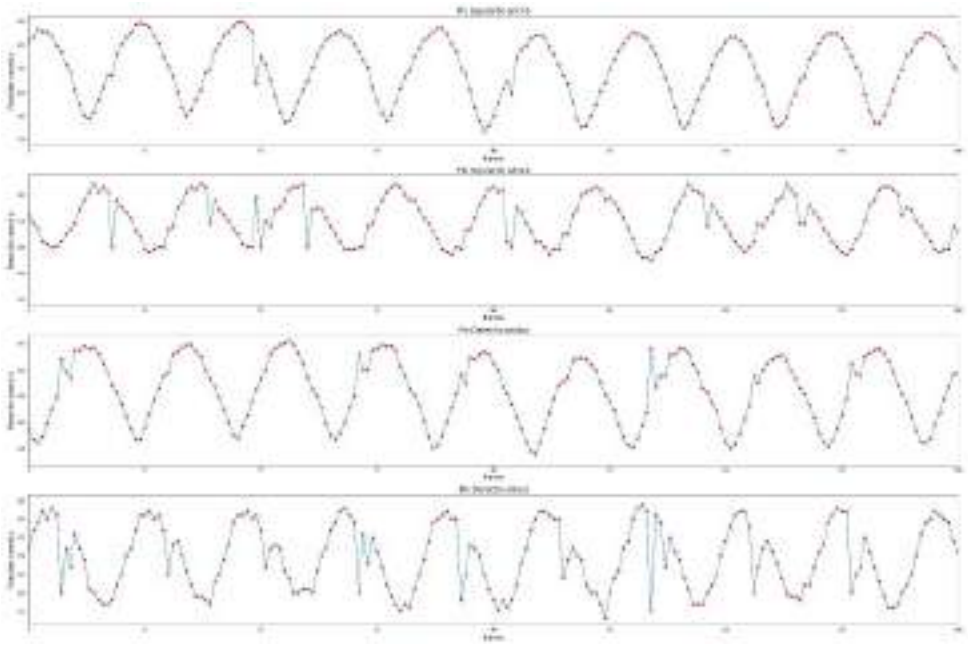

Figura 5.3: Evolución serie temporal keypoints pie izquierdo y derecho

Donde para cada cade frame en 5.3, tendríamos la posición de x o y del keypoint. Finalmente, es con esta información de serie temporal con la que extraeremos las métricas, mediante operaciones basadas en procesado de series temporales.

5.2. Detección de Pasos

Si observamos los videos grabados podemos ver que, dada la posición de la cámara, se observa mejor el lado izquierdo del sujeto, por lo que para aplicar un algoritmo de detección de pasos vamos a trabajar sobre la serie temporal l_y, la serie temporal del keypoint del tobillo izquierdo en el eje y por lo siguiente 5.4:

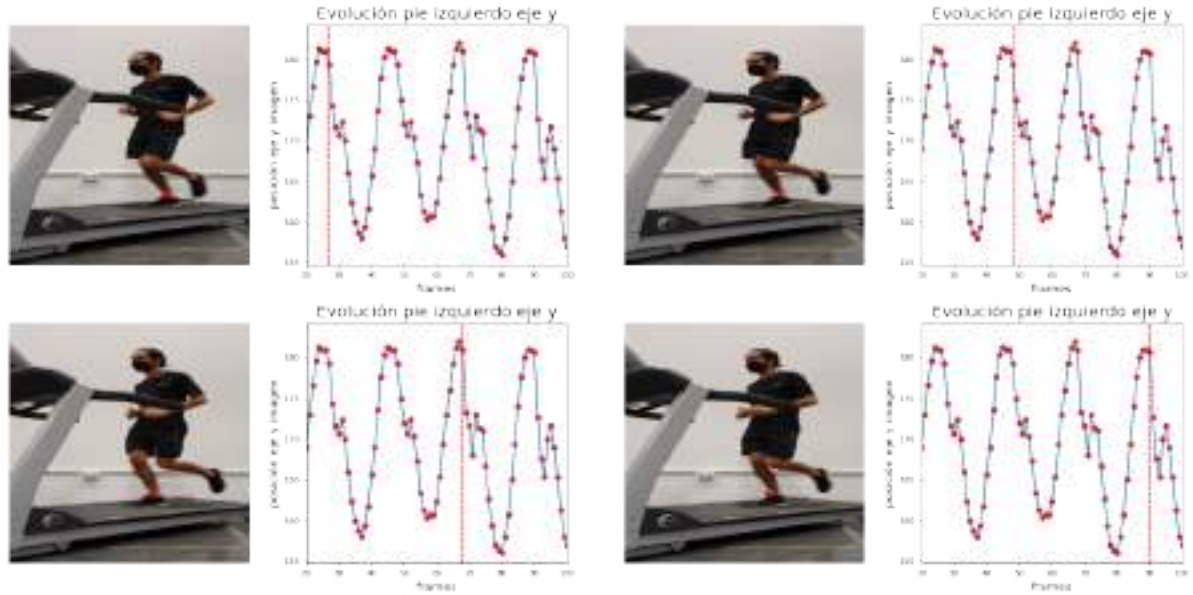

Figura 5.4: Gráfica detección pasos

Si observamos la gráfica 5.4, podemos deducir que el individuo realiza un paso en el momento en el que la serie temporal alcanza un máximo local. Es por ello, por lo que tenemos que aplicar un algoritmo de detección de picos lo suficientemente eficaz como para detectar los pasos correctamente para todos los videos. Para abordar el problema, se han aplicado dos métodos distintos: máximo móvil y método basado en diferencias temporales

5.2.1. Preprocesado serie temporal de pie izquierdo en altura

Tal y como hemos comentado antes, nuestra serie temporal de interés es la evolución en altura del keypoint del pie izquierdo

Antes de aplicar cualquier tipo de preprocesado, hemos observado que la serie temporal es bastante ruidosa con picos anormales, además, en algunos videos presenta instantes en los que **ly(t)** no es observado (tenemos valor ausente en esas situaciones), es por ello que si primero aplicamos un suavizado y una interpolación, cualquier métrica de corredores que queramos extraer será mucho más exacta.

El suavizado de la señal lo conseguimos aplicando el filtro de la media móvil sobre **ly(t)**:

$$l_y(k)_{smooth} = \frac{1}{3}\Big(ly(n) + ly(n-1) + ly(n-2)\Big)$$

Para realizar la interpolación de la serie se ha aplicado un método que combina la media de las dos posiciones más próximas con un auto arima que se aplica cuando hay más de dos valores ausentes consecutivos:

$$\begin{cases} ly(n) = \frac{1}{2}\Big(ly(n-1) + ly(n+1)\Big) \text{si } n+1 \neq na \\ ly(n) = \frac{1}{2}\Big(ly(n-1) + ly(n+2)\Big) \text{si } n+1 = na, n+2 \neq na \\ ly(n) = forecast(auto\_arima(\{ly(0), \cdots, ly(n-1\}), size = 1) \text{ si otro} \end{cases}$$

El primer caso lo aplicamos cuando en el siguiente instante no es un valor ausente, entonces el valor en el instante n lo interpolamos con la media del instante anterior y el siguiente. El segundo caso ocurre cuando desconocemos el valor en n+1 pero en el siguiente instante si lo conocemos, entonces interpolaríamos con la media del valor en n-1 y n+2. El tercer caso, nos lo encontramos cuando los dos instantes siguientes al actual son valores ausentes, entonces lo que hacemos es ajustar un modelo auto arima con toda la serie temporal de 0 a n-1 y predecir el valor en instante n.

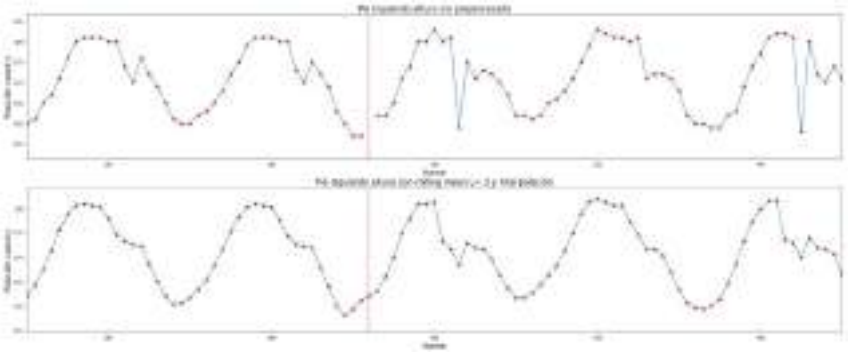

Figura 5.5: Serie temporal ly antes y después del preprocesado

Este motivo por el cual interpolamos con la media o con un modelo autoarima es debido a que la media es eficaz con datos cercanos, pero pierde veracidad cuando los datos son más lejanos. Por otro lado, la interpolación de muchos datos consecutivos tiene más veracidad con un modelo auto arima (que tiene en cuenta las observaciones previas, el error en instantes previos y la estacionalidad), pero tiene mucho más coste computacional y podría retardar la ejecución considerablemente

5.2.2. Método basado en máximo móvil

La detección de pasos por el método del máximo móvil consiste en considerar como pico aquel punto que coincida con el máximo en una ventana temporal de $\pm Lparan$.

El método para la extracción de pasos sería el siguiente:

$$t_{step} = \begin{cases} n \Big| ly(n) = max\Big\{ly(n-kv : n+kv)\Big\} \text{ si } n \geq L \\ n \Big| ly(n) = max\Big\{ly\big(0 : n+\frac{3}{2}L\big)\Big\} \text{ si } n < L \end{cases}$$

Si lo desplegamos:

$$t_{step} = \begin{cases} n \Big| ly(n) = max\Big\{ly(n-kv), ly(n-kv+1), \cdots ly(n), ly(n+1), \cdots ly(n+kv)\Big\} \text{ si } n \geq L \\ n \Big| ly(n) = max\Bigg\{ly(0), ly(1), \cdots, ly(n), ly(n+1), \cdots, ly\Big(\lfloor n+\frac{3}{2}L \rfloor\Big)\Bigg\} \text{ si } n < L \end{cases}$$

Podemos comprobar visualmente como funciona el algoritmo:

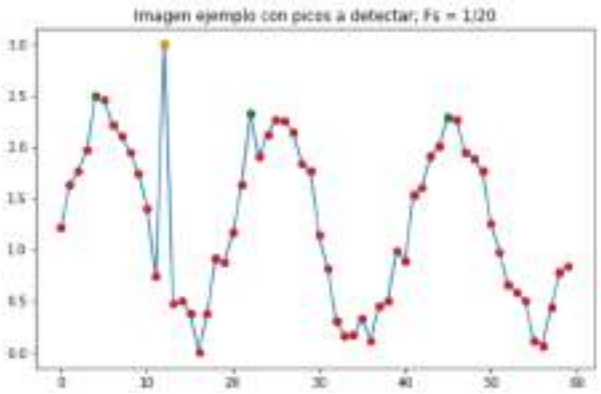

Figura 5.6: Serie temporal de ejemplo

Para realizar esta figura simulamos una sinusoidal de Frecuencia $\frac{1}{20}$ con cierta adición de ruido. Podemos observar que los puntos rojos no corresponden con los máximos, los puntos verdes son nuestros máximos de interés y el punto amarillo es ruido que se sitúa por encima de los máximos que corresponden con los pasos.

Con una ventana temporal de L=5 observamos lo siguiente 5.7:

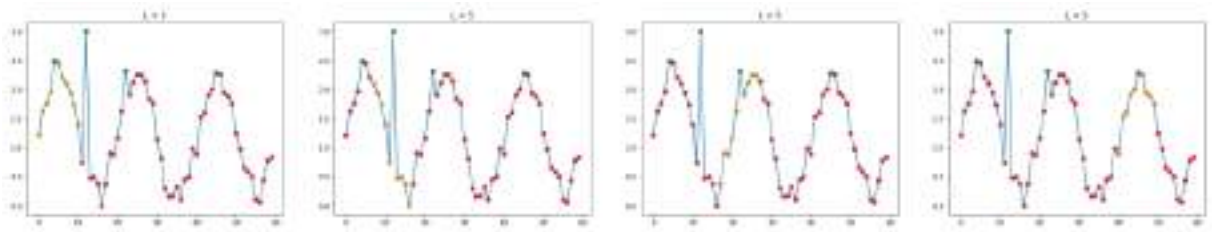

Figura 5.7: Pasos detectados con L=5. Verde: Puntos detectados como pasos, Amarillo: Puntos dentro de ventana temporal, Rojo: puntos restantes. 4 pasos detectados

Para entender la figura 5.7, tenemos que mirarla de izquierda a derecha. Hay una gráfica por cada paso detectado, de modo que en cada una de ellas, aparece el punto verde como el paso, y puntos amarillos en los L puntos anteriores y posteriores. Los puntos restantes aparecen de color rojo. Dicho esto, podemos ver que en principio, con una ventana temporal de 5 logra detectar todos los pasos más el ruido. Visualicemos que ocurriría con una ventana temporal de L=2 y con otra de L=9.

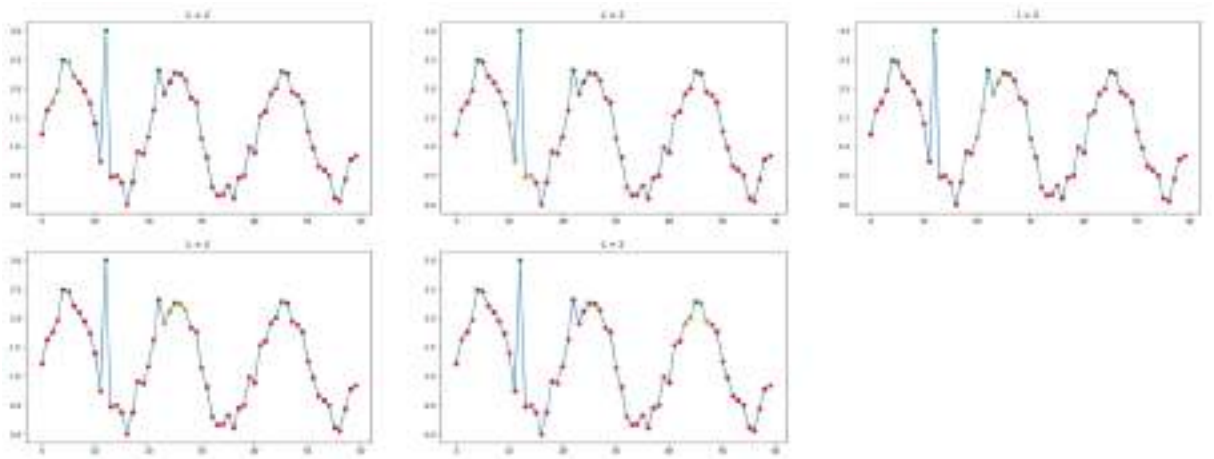

Figura 5.8: Pasos detectados con L=2. Verde: Puntos detectados como pasos, Amarillo: Puntos dentro de ventana temporal, Rojo: puntos restantes. 5 pasos detectados

Con la misma explicación de la gráfica aportada en la figura 5.7, podemos observar en 5.8 que al reducir el tamaño de la ventana temporal (puntos amarillos pintados en cada gráfica), nos aseguramos de que todos los pasos sean detectados, sin embargo, terminamos detectando falsos pasos debidos a los pequeños picos que surgen por tener una señal ruidosa, por lo que incrementamos el ruido.

Si probamos con L=9 5.9:

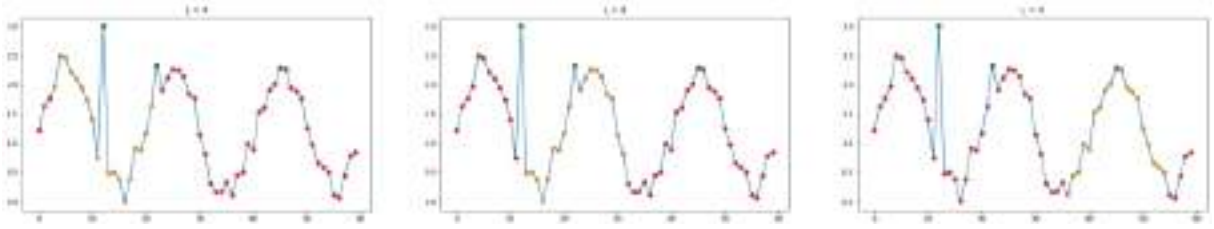

Figura 5.9: Pasos detectados con L=9. Verde: Puntos detectados como pasos, Amarillo: Puntos dentro de ventana temporal, Rojo: puntos restantes. 3 pasos detectados.

Si visualizamos la figura 5.9 podemos ver que el número de puntos amarillos por paso detectado es mucho más grande que en 5.8 y que en 5.7, no obstante se detectan menos pasos. Esto es debido a que, cuando consideramos una ventana temporal grande, nos aseguramos de que los máximos detectados, realmente lo sean, eliminando el ruido de falsos pasos (sin poder eliminar el ruido superior al nivel de los máximos correspondidos con los pasos) a costa de detectar menos pasos.

En resumen, cuanto mayor sea el tamaño de la ventana, menos pasos detectaremos pero eliminaremos más ruido, y cuanto menor sea, mayor el número de pasos pero mayor será el ruido. Este método puede funcionar siempre y cuando tuvieramos una serie sin tendencia, estacionaria, con una frecuencia constante (o con poca variabilidad) y sin picos de ruido de mayor altura al nivel de la altura de los pasos promedio. Respecto al tamaño de la ventana temporal, lo aconsejable sería escoger un tamaño de ventana inferior o igual a un tercio del periodo (inversa de la frecuencia) dominante de la serie . Sin embargo, este no es un método viable para nuestro caso de estudio, ya que no tiene capacidad de generalización, debido a que en videos con distintas velocidades, el tamaño de ventana temporal ideal sería distinto al tener diferentes frecuencias.

5.2.3. Método basado en diferencias temporales

Este método es el que mejores resultados ha proporcionado por video, es debido a que un método para encontrar máximos que no depende de la frecuencia de la señal, por lo que es más generalizable en lo que se refiere a la velocidad del corredor.

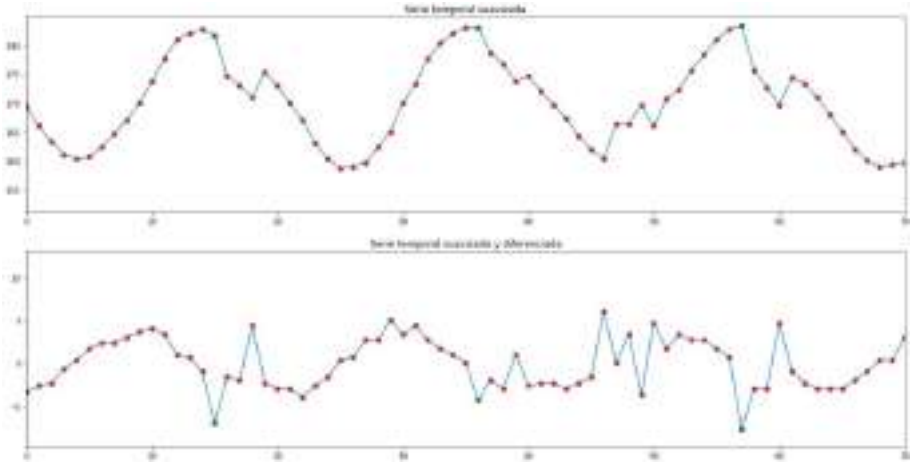

Figura 5.10: Ejemplo serie temporal diferenciada. Verde: pasos detectados, Rojo: puntos restantes

Para aplicar el algoritmo de detección de pasos, en primer lugar, se aplica una diferencia temporal 5.10: $\nabla_1 X_t = X_t - X_{t-1}$. Después, marcamos como puntos candidatos aquellos cuyo valor en la serie diferenciada se sitúe por debajo de un threshold negativo α

$$t_{candidato} = \left\{ n \middle| \nabla_1 ly_{smooth}(n) < \alpha \right\}$$

Con $\alpha = -2{,}5$ se encuentran los siguientes candidatos 5.11:

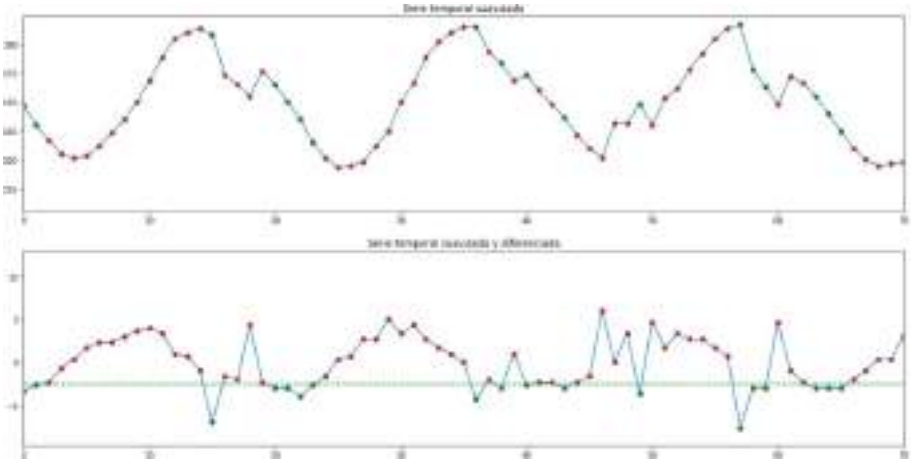

Figura 5.11: candidatos con $\alpha = -2{,}5$

Filtramos $t_{candidatos}$ eliminando puntos consecutivos, en nuestro algoritmo hemos fijado un número de 3 valores consecutivos para el filtrado 5.12, lo que quiere decir que si un pico es detectado a menos de tres frames de distancia de otro pico, se descarta de los candidatos.

Como al calcular la diferencia temporal nos devuelve la distancia entre valores, esta vez diferenciamos $t_{candidatos}$

$$t_{candidatos'} = \left\{ n \middle| \nabla t_{candidatos}(n) > 3 \right\}$$

Una explicación gráfica e iterativa de lo que hace este algoritmo es lo siguiente 5.12:

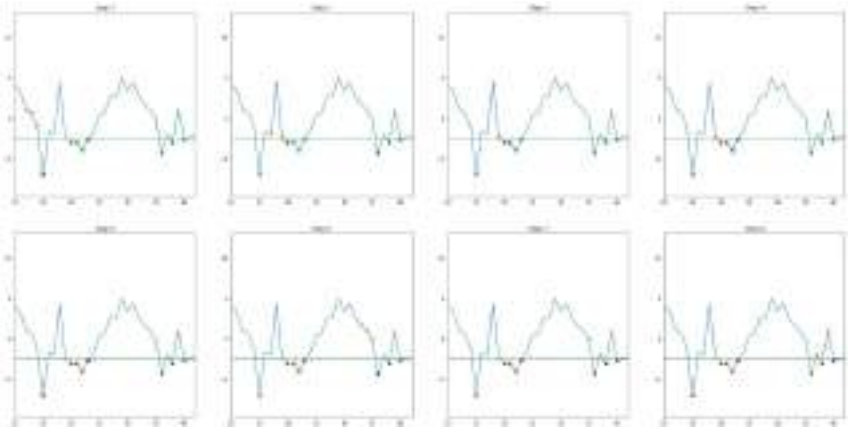

Figura 5.12: Filtrado de candidatos. (a) Azul: punto actual (b) Verde: Puntos candidatos. (c) Rojo: Punto evaluado que ya es candidato (d) Amarillo: puntos que se evalúan

En la gráfica 5.12 vemos que en cada punto actual, cuando evalúa los 3 puntos anteriores, si al menos uno de ellos ya es candidato, el punto actual se descarta (se queda de color rojo).

Aplicamos este método y los candidatos filtrados se nos quedan así 5.13:

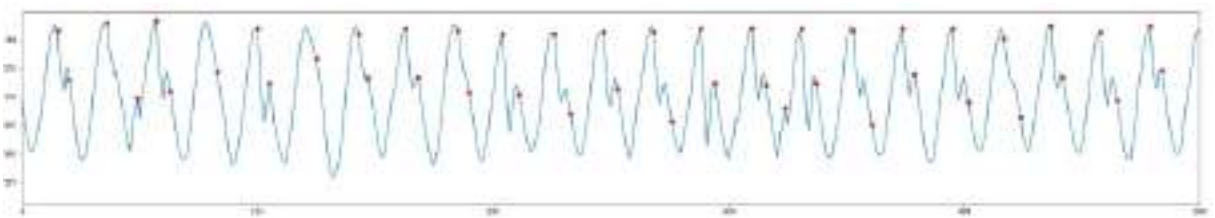

Figura 5.13: Resultado aplicar filtrado

Podemos observar que las series temporales tienen una cierto componente estacional que consiste en un pico elevado (el que corresponde con el verdadero paso), seguido de un semi pico que corresponde con otro máximo local, el cual no está correspondido con un paso.

Para eliminar estos picos aplicamos el siguiente filtro a los candidatos:

$$\texttt{lower\_limit} = max(ly_{smooth}) - \gamma Var(ly_{smooth})$$
$$t_{pasos} = \left\{ t_{candidatos'} \middle| ly_{smooth}(t_{candidatos'}) < \text{lower\_limit} \right\}$$

Donde γ es otro parámetro a indicar que determina cual será el límite inferior para ser considerado como paso. Cuanto más cercano sea a 0, será más estricto, mientras que cuanto mayor sea, admitirá más ruido.

Si probamos los resultados para $\gamma = 0{,}1$ vemos podemos visualizar gráficamente 5.14:

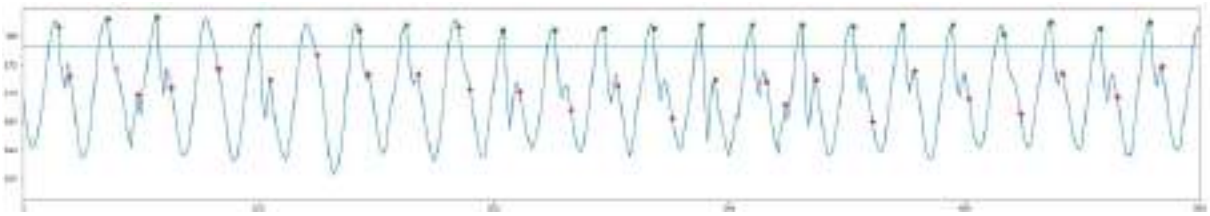

Figura 5.14: Pasos detectados filtrados con $\gamma = 0{,}1$. (a) Verde: Pasos detectados, Rojo: Pasos rechazados

5.2.4. Resultados métodos detección de pasos

Se ha podido evaluar cada una de las metodologías descritas previamente para todos los videos y se han obtenido los siguientes errores.

Para el método del máximo móvil, el mejor error que se obtuvo para todos los videos fue utilizando una longitud de ventana de tamaño 17, tal y como podemos observar en la figura 5.15:

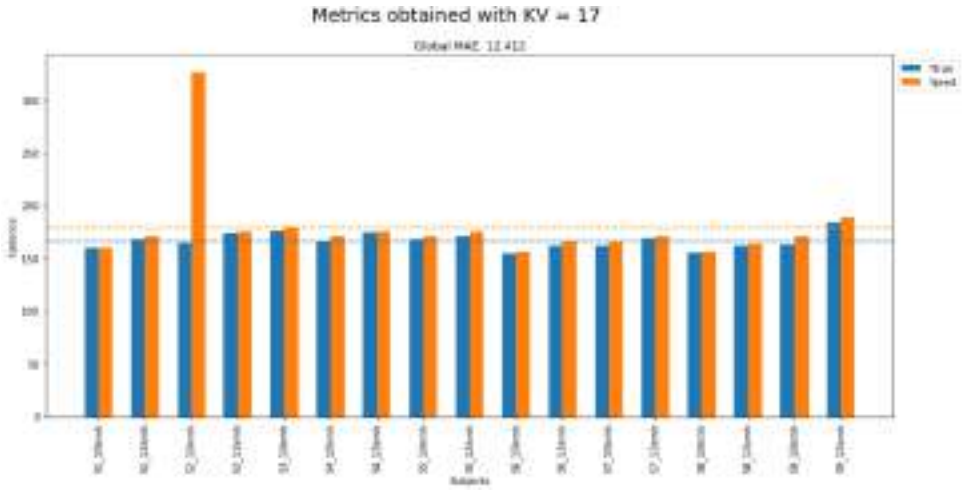

Figura 5.15: Resultados métricas método máximo móvil con L=17

Con el método de de diferencias de series temporales 5.16, el algoritmo que ha proporcionado mejores resultados es empleando los parámetros $\alpha = -2{,}5$ y $\gamma = \frac{1}{15}$

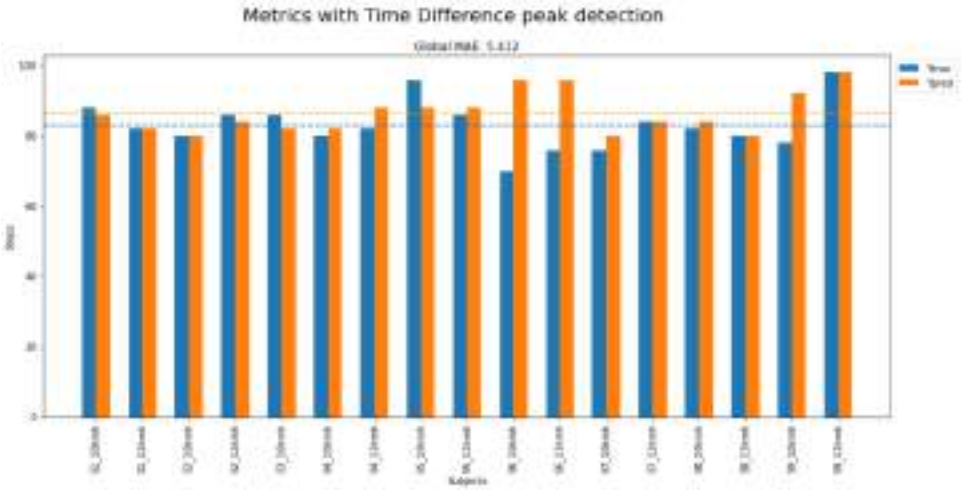

Figura 5.16: Resultados métricas método de diferencias de series temporales con $\alpha = -2{,}5$ y $\gamma = \frac{1}{15}$

El método de diferencias de series temporales ha demostrado para este caso de estudio adaptarse mejor a la detección de pasos 5.16, dado que se adapta mejor a las diferentes velocidades de los sujetos. No obstante, depende de más parámetros para utilizarse.

5.3. Estimación de Cadencia

La cadencia son el número de pasos por minuto de un corredor de cinta. Es una técnica bastante empleada por los fisioterapeutas para estimar el rendimiento de la marcha de un sujeto. Por lo general, si la evaluáramos en diferentes sujetos para la misma velocidad, observaríamos que las personas con menor estatura tienden a tener mayor cadencia respecto a las de mayor estatura.

Los pasos para obtener la cadencia son bastante sencillos:

1. Extracción keypoints pie izquierdo y obtención de la serie temporal.
2. Preprocesado señal.
3. Calcular los pasos: **St**.
4. Calcular el periodo de pasos: T_{frames}.
5. Convertir T_{frames} $\rightarrow$T$_s$.
6. Calcular la frecuencia de pasos en segundos: $\nu[s^{-1}] = \frac{1}{T_s}$.
7. Cadencia = 2 x 60 seg x ν.

En primer lugar se haría la extracción de keypoints y la obtención de la serie temporal del pie izquierdo utilizando los métodos ya explicados previamente. En segundo lugar, interpolaríamos los valores de la señal por el método de la media y arima combinados ya descrito y suavizaríamos la señal, para reducir los picos ruidosos, utilizando el filtro de la media móvil de ventana 3. En tercer lugar, ya con la señal preprocesada, aplicaríamos el algoritmo de detección de pasos basado en diferencias temporales, para obtener la serie temporal de los pasos **St**, que nos indica todos los instantes temporales (en frames) en los que se ha podido detectar un paso, para ello hemos utilizado los parámetros $\alpha = -2{,}5$ y $\gamma = \frac{1}{15}$, que han sido los que mejor resultado han proporcionado. En cuarto lugar, tenemos que obtener el periodo.

Para obtenerlo, se ha aplicado el siguiente método:

$$\nabla_1 St(k) = St(k) - St(k-1)$$

$$T_{frames}(k') = median\left\{\left[\nabla_1 St(k),\ \nabla_1 St(k-1),\ \nabla_1 St(k-2),\ nabla_1 \nabla_1 St(k-3)\right]\right\}$$

Dado que **St(0)**, **St(1)** son los frames donde se dan el primer y el segundo paso, la diferencia de los frames ∇_1 **St(t)**, nos va a obtener una serie temporal de **periodos en frames**. No obstante, hay algunas pasos que no se detectan, o pasos falsos que se detectan, es por ello que si calculamos el periodo como la diferencia, obtendríamos valores muy ruidosos, es por ello que aplicamos una **mediana móvil** (para reducir el ruido de pasos outliers) de tamaño 4. De este modo, tendríamos un periodo temporal en frames más estable.

El quinto lugar sería la conversión de T_{frames} a $T_{segundos}$, para ello es necesario saber los fps (frames per second) del dispositivo de la cámara. En caso de estar utilizando **OpenCV**, este parámetro lo conocemos con la función **(CAP_PROP_FPS)**, que en el caso de las cámaras utilizadas, los videos están grabados a 30 fps.

Una vez conocemos los fps, convertir de $T_{segundos}$ a T_{frames} es tan sencillo como

$$T_{segundos} = \frac{T_{frames}}{fps}$$

En sexto lugar, tendríamos que convertir de periodo $T_{segundos}$ a frecuencia $\nu_{segundos^{-1}}$

$$\nu_{segundos^{-1}} = \frac{1}{T_{segundos}}$$

Finalmente, obtenemos la cadencia:

$$Cadencia(k) = 2 \text{ x } 60\text{s x } \nu_{segundos^{-1}(k)}$$
$$Cadencia = median\Big\{Cadencia(k)\Big\}$$

Utilizamos la mediana también por el mismo motivo, que en el cálculo de T_{frames}, debido a los valores outliers obtenidos. Estos son los resultados obtenidos al aplicar los pasos anteriores 5.17:

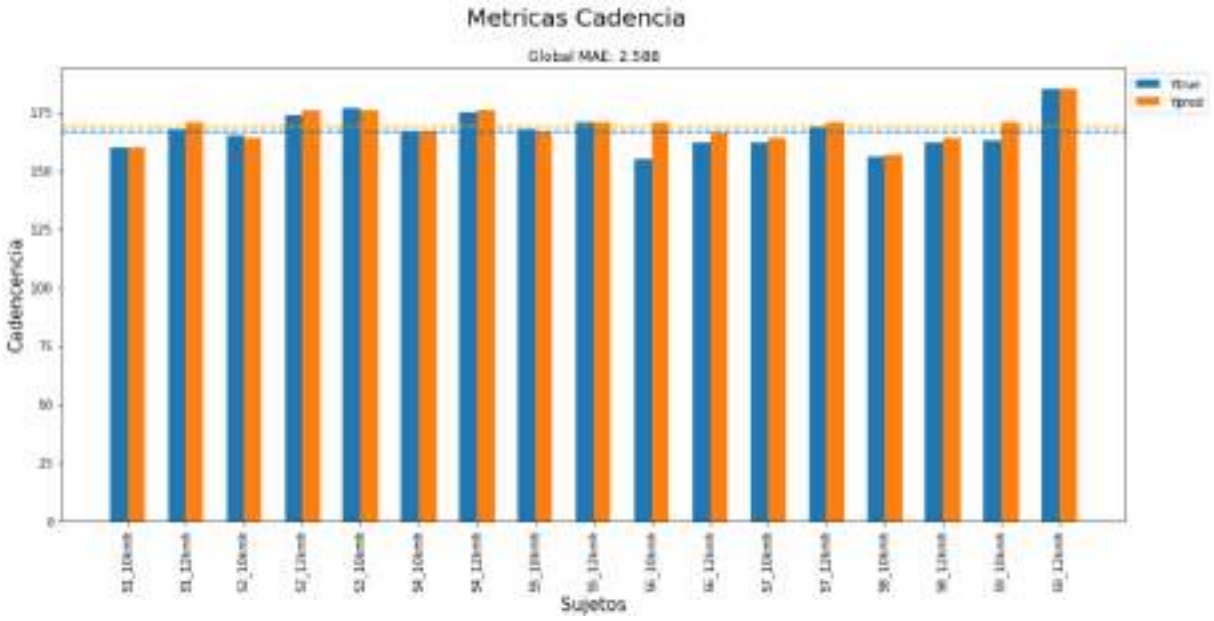

Figura 5.17: Métricas de la cadencia. Línea discontínua amarilla: media cadencias predecidas; Línea discontínua azul: media cadencias reales

5.4. Estimación del máximo ángulo de separación de piernas

Otra métrica interesante a considerar es el ángulo de máxima separación de piernas 5.18. Esta es una medida que ha sido calculada sin la etiqueta proporcionada por el propio sensor.

Figura 5.18: Ejemplo de dibujo y estimación del ángulo

Para realizar el cálculo se ha aplicado el siguiente esquema 5.19:

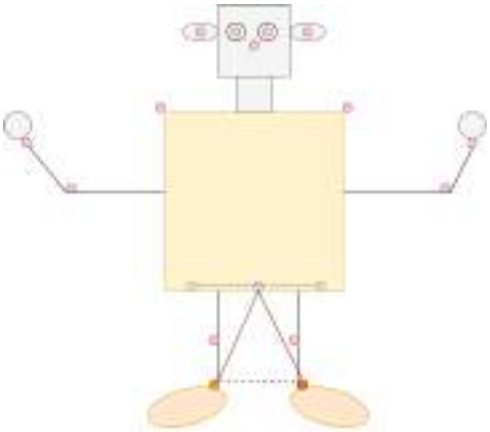

Figura 5.19: Esquema del ángulo de interés y de disposición de keypoints

En primer lugar, si nos fijamos en los dos puntos verdes de la figura 5.19, referidas a la cintura izquierda y derecha, tenemos que calcular la pelvis como la distancia media de ambos keypoints, tal y como se indica al principio del capítulo:

$$pelvis = \left(\frac{L_x + R_x}{2}, \frac{L_y + R_y}{2} \right)$$

En segundo lugar, tenemos que calcular los vectores de las dos líneas rojas 5.19

$$\vec{PL} = \Big(L_x - P_x, L_y - P_y\Big)$$
$$\vec{PR} = \Big(R_x - P_x, R_y - P_y\Big)$$

Después calculamos el módulo de cada uno de los dos vectores:

$$\big|\vec{PL}\big| = \sqrt{PL_x^2 + PL_y^2}$$
$$\big|\vec{PR}\big| = \sqrt{PR_x^2 + PR_y^2}$$

Seguidamente, calculamos el ángulo de separación de las piernas del siguiente modo:

$$\theta_{rad} = aCos\left(\frac{\vec{PL}\cdot\vec{PR}}{\big|\vec{PL}\big|\big|\vec{PR}\big|}\right)$$
$$\theta_{sex} = \theta_{rad}\frac{180}{\pi}$$

Si aplicamos estos pasos en cada frame del video, obtendremos la siguiente serie temporal

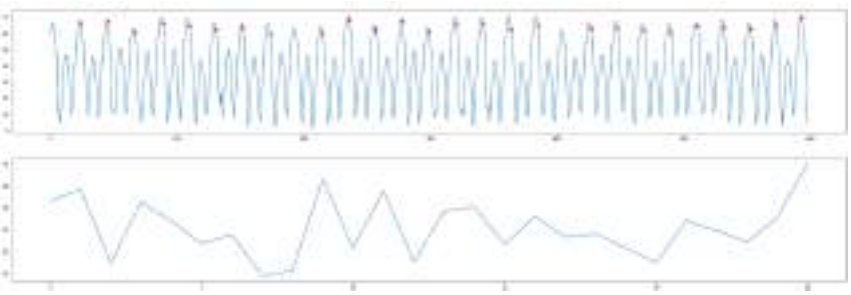

Figura 5.20: Gráfica ángulos separación. La parte de arriba representa el ángulo de separación en cada frame, con los ángulos máximos detectados. La parte de abajo, representa la evolución temporal del máximo ángulo de separación

Cuando obtenemos la serie temporal de los ángulos de separación, tenemos que aplicar otro suavizado e interpolación de la señal, aunque en este caso, el suavizado lo hemos realizado con la media movil de tamaño L=2. Para la obtención de la señal de los máximos ángulos de separación, utilizando la señal obtenida con el suavizado tal y como observamos en 5.20, aplicamos el mismo algoritmo que utilizamos en la detección de pasos, el basado en diferencias de series temporales, con la diferencia de que, en este caso, utilizamos los parámetros $\alpha = -5$ y $\gamma = \frac{1}{30}$.

Finalmente, el máximo angulo de separación lo obtendremos con la mediana de toda la señal, con el objetivo de reducir el ruido. Por último, el máximo ángulo de separación estimado para cada uno de los sujetos, siguiendo los pasos descritos, es el siguiente 5.21:

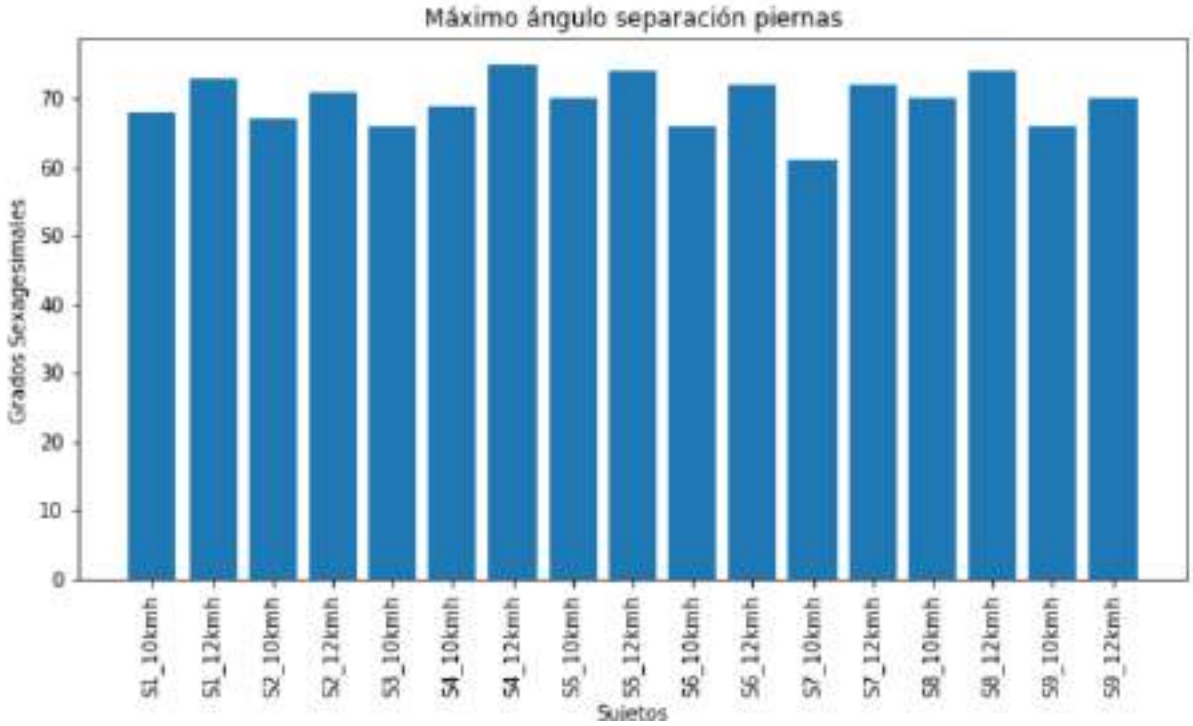

Figura 5.21: Métricas máximo ángulo de separación

5.5. Tipos de Enfoque del problema

El programa está pensado para poderse realizar, tanto de manera Off-line cómo On-line, de manera que se puedan obtener las métricas de los corredores de cinta. De manera que, por un lado se puede indicar la dirección de la carpeta que contenga los videos de los corredores y, por otro lado, se puede realizar grabando a tiempo real con una cámara

5.5.1. Enfoque Offline

Se introduce como parámetro la dirección en la que se encuentra la carpeta con los videos y la dirección en la que se van a escribir los videos. Funciona de forma que el video se lee y se escribe con **OpenCV**, y utiliza un lapso de 300 frames (que equivaldrían a 10 segundos de video) para mostrar las métricas en el video por pantalla. Este lapso del tiempo es debido a que la estimación de las métricas se realiza en micro-batches temporales de 300 frames. De modo que, por un lado, a partir de los 10 segundos muestra por pantalla los cambios en corto plazo de las estimaciones del sujeto y, por otro lado, ahorra coste computacional de tener que estimar las métricas con todos los instantes anteriores. Como salida, el programa escribe en la ruta indicada los videos con el esqueleto obtenido por la estimación de pose dibujado, y con las métricas mostradas en el mismo 5.23. Además, genera un informe en un pdf 5.22 donde en la primera página aparece una tabla resumen de las métricas y en las restantes, aparece la gráfica de la evolución de cada una de las métricas, para cada sujeto.

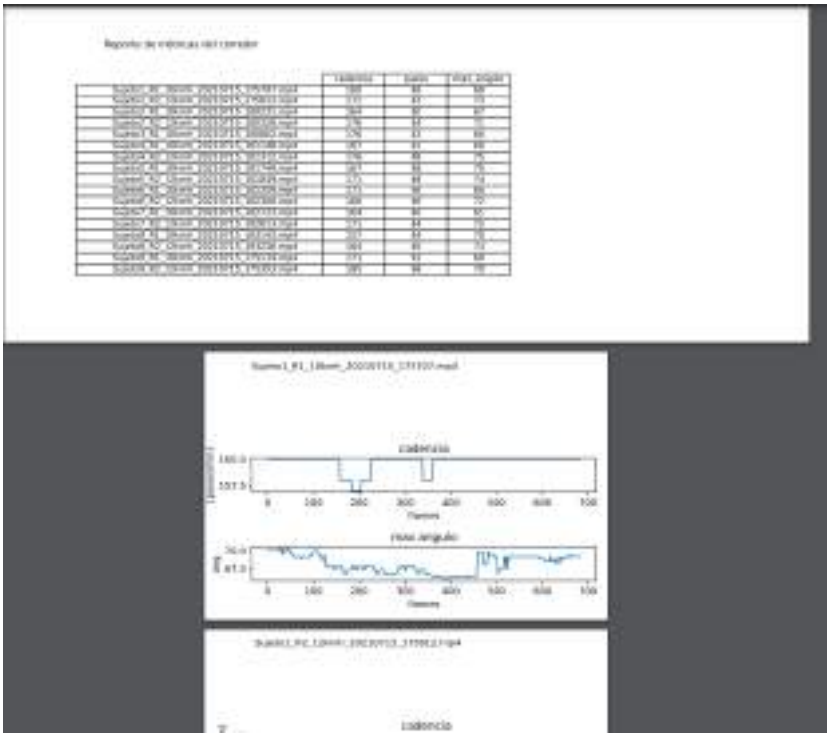

Figura 5.22: Demo del informe en pdf. La primera página tiene la tabla con el resumen de resultados. Las páginas restantes tienen la evolución de las métricas de cada sujeto

Y así se vería la carpeta del output una vez ejecutado el programa 5.23:

Figura 5.23: Carpeta con resultados obtenidos con el programa en modo Off-Line. Tiene el reporte en pdf y los videos con el esqueleto y las métricas

Link al drive compartido con la carpeta de los resultados:
https://drive.google.com/drive/folders/1y5lPY$_{8Z}$*bsUeczz*22*xXcVFNg*6*O*3*OWYl?usp* = *sharing*

5.5.2. Enfoque Online

Mientras que el enfoque Off-Line no muestra ningún problema en ejecutarse con cualquier tecnología Hardware (aunque en el caso de la CPU la ejecución sería bastante más lenta), el enfoque online ha podido ser posible gracias al uso de la **Edge TPU** [15], que ha demostrado, tal y como mostramos en la figura 4.9, que puede ser hasta 10 veces más rápido que la GPU del propio ordenador (Nvidia GeForce GTX 1080) con el modelo de MoveNet de **lightning**, aunque la diferencia de tiempos de ejecución en **thunder** entre GPU y TPU no es significativa, por lo que también podría utilizarse la GPU.

Es por ello, que hemos podido implementarlo con thunder que, en principio, era un modelo más lento que lightning 4.9 aunque más eficaz. El programa tiene asegurado su funcionamiento si se está utilizando el dispositivo **Edge TPU**, aunque también podría funcionar con GPU, por mostrar un tiempo de ejecución similar. No obstante, su uso quedaría descartado si quisieramos ejecutar el programa con CPU, debido a que tarda unos 8 segundos por frame 4.9, por lo que probablemente colapse.

El programa funciona de modo que se puede grabar utilizando, tanto la cámara del ordenador, como una webcam conectada al mismo. Además, se puede aplicar un retardo de ejecución al programa de 5 o más segundos para que al sujeto le de tiempo a ejecutar y ponerse a correr en la cinta.

Para utilizarse correctamente, es bastante necesario que se cumplan los siguientes requisitos:

- la distancia de la cámara ha de ser relativamente cerca.
- la cámara ha de estar posicionada en una perspectiva en la que se pueda detectar todas las partes del cuerpo.

Las métricas se van estimando, al igual que el enfoque Off-Line, por mini-batches, aunque en este caso los mini-batches temporales son de 16.6 segundos (500 frames). Las métricas las va mostrando por pantalla a medida que el sujeto corre a tiempo real. El programa deja de ejecutarse una vez transcurre 1 minuto o cuando el usuario pulse la tecla del ordenador **s**, en cuyo caso el programa se detendrá de inmediato. Una vez termine de ejecutarse, se generará un informe pdf 5.22 con el mismo contenido que con el modo Off-Line, con el resumen de las métricas obtenidas y con una gráfica de la evolución de las mismas.

Aquí dejamos un link a la demo del programa online: https://youtu.be/F31Nnxhl-ag

Screenshots de ejemplos en el video de la demo 5.24 5.25 5.26 5.27

Figura 5.24: Sujeto corriendo en la cinta

Figura 5.25: Comprobación funcionamiento de la TPU. LED intermitente del dispositivo (no apreciable en una captura) indica que se encuentra operativa

Figura 5.26: Tiempo de espera para realizar el cálculo

Figura 5.27: Cálculo métricas tiempo real

Capítulo 6

Conclusiones Finales

A lo largo del trabajo fin de grado, hemos realizado una breve introducción acerca de en que consiste la visión por computador, donde hemos introducido la detección de objetos, junto a modelos importantes que se utilizan en ese campo de estudio, seguido de la descripción de los tipos de tecnologías de hardware y como estas importan en el campo del machine learning a la hora de realizar entrenamiento o inferencia. Asimismo, hemos descrito el dispositivo de Google Coral Edge TPU.

Una vez descritos los conceptos más importantes anteriores, pasamos a detallar en profundidad los modelos de estimación de pose.

Al comparar los tipos de modelos de estimación de pose, llegamos a la conclusión que para nuestro caso de estudio de los corredores de cinta, el modelo o librería que más nos sirve entre OpenPose, Detectron2 y MoveNet, es el tercero, más en concreto, el modelo thunder que utiliza Edge TPU. Esto es debido a su alta velocidad de ejecución, lo que nos permitirá aplicar la estimación a tiempo real de los corredores.

Finalmente pasamos al caso de estudio de los corredores de cinta. En este punto, se disponían como datos 17 videos proporcionados por 9 sujetos voluntarios, pero dada la poca cantidad de videos, el uso de redes neuronales profundas para estimar las métricas quedó descartada.

Es por ello, que se han tenido que recurrir a métodos más tradicionales basados en el procesado de series univariantes, cuyas series vienen dadas por las posiciones de los keypoints a lo largo de todo el video.

Se logró estimar con gran precisión algunas métricas como el número total de pasos y la cadencia, además de incorportar otra métrica como el máximo ángulo de separación de las piernas.

Sin embargo, a pesar de haber conseguido afrontar este problema mediante estas estrategias basadas en procesado de señales, estos métodos han sido realizados sobre el mismo conjunto de videos bajo las mismas condiciones, por lo que no se puede afirmar con certeza que el algoritmo logre generalizar la extracción de estas métricas. Al fin y al cabo, solamente se han analizado videos que han sido grabados en 2 únicas velocidades (10km/h y 12km/h), a una distancia fija de la cinta y con la misma perspectiva. Además de que con estos métodos puede darse el caso en el que una determinada posición de la cámara haga que el algoritmo no funcione, sobretodo aquellas posiciones en las que una sola parte del cuerpo no sea captada.

Es por ello, que harían falta más videos realizados con más cámaras en distintos ángulos y posiciones, de manera que pudieramos sustituir estos métodos basados en procesamiento de la señal poco extrapolables, por algoritmos basados en redes neuronales profundas y generalizables.

No obstante, estos métodos sirven para demostrar, como prueba de concepto, que se puede abordar este problema e incluso llegar a implementarse a tiempo real, lo cual ha sido logrado empleando los métodos expuestos en el trabajo.

Bibliografía

[1] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2013.

[2] T. Gevers A.W.M.Smeulders J.R.R. Uijlings, K.E.A. van de Sande. Selective search for object recognition. *International Journal of Computer Vision*, pages 154–171, 2013.

[3] Rohith Gandhi. R-cnn, fast-rcnn, faster-rcnn, yolo - object detection algorithms.

[4] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[5] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[6] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2017.

[7] Peng Tang, Xinggang Wang, Angtian Wang, Yongluan Yan, Wenyu Liu, Junzhou Huang, and Alan Yuille. Weakly supervised region proposal network and object detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[8] Architecture of the central processing unit (cpu).

[9] Cornell University. Understanding gpu architecture: Gpu characteristics.

[10] Norm Jouppi. Google supercharges machine learning tasks with tpu custom chip.

[11] Google Cloud. System architecture tpu vm.

[12] Antonio J. Rivera, Francisco David Charte Luque, Macarena Espinilla, and María D. Pérez-Godoy. Nuevas arquitecturas hardware de procesamiento de alto rendimiento para aprendizaje profundo, 10 2018.

[13] Coral A.I. Datasheet coral usb accelerator.

[14] Coral A.I. Get started with the usb accelerator.

[15] Coral A.I. Tensorflow models on the edge tpu.

[16] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields, 2018.

[17] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[18] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.

[19] Tensorflow HUB. Movenet: Ultra fast and accurate pose detection model.

[20] Tensorflow HUB. movenet/singlepose/lightning.

[21] Tensorflow HUB. movenet/singlepose/thunder.

[22] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[23] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6568–6577, 2019.

[24] Chi-Feng Wang. A basic introduction to separable convolutions.

[25] RunScribe.

[26] Yang B. Hicks J.L. et. al. Kidziński, Ł. Deep neural networks enable quantitative movement analysis using single-camera videos. *Nature Communications*, (11):4054–, 08 2020.