Tutorial 3

Detección de articulaciones del cuerpo con movenet/singlepose/lightning

Introducción

En este tutorial, cargaremos y usaremos el modelo bruto movenet/singlepose/lighting que detecta las articulaciones del cuerpo de un cuerpo humano en una imagen o fotograma. Las aplicaciones del modelo incluyen entrenamiento físico, atención médica y realidad aumentada, para nombrar algunas ideas.

movenet/singlepose/lighting es un modelo de red neuronal convolucional (CNN) que se ejecuta en imágenes RGB y predice la ubicación de las articulaciones humanas de una sola persona. Puede ejecutarse >50FPS en la mayoría de los portátiles modernos
y es más adecuado para detectar la pose de una sola persona que está a una distancia de entre 1 y 2 metros. Si hay más de una persona en la imágen, el modelo detecta la pose de la persona que está más cerca del centro de la imagen. El modelo proporciona una predicción de 17 puntos clave del cuerpo, incluso cuando están ocluidos.

El modelo opera con imágenes RGB de forma [192, 192, 3] y requiere un tensor de entrada [1, 192, 192, 3]. Todos los valores deben ser del tipo 'int32'. La primera dimensión se usa para lotes de imágenes. La segunda, tercera y última para los píxeles de la imagen: la imagen debe ser un cuadrado de dimensiones 192x192 píxeles. La segunda dimensión de la forma representa la altura, la tercera el ancho y la última los canales de color (rojo, verde y azul) de cada pixel. Todos los valores almacenados en una altura, anchura y canal determinados deben estar entre 0 y 255.

El modelo devuelve un tensor de salida 4D de forma [1, 1, 17, 3]. Todos los valores son del tipo 'float32'. El último valor en la forma de salida es 3, es decir, es una array de 3 números:

- La coordenada **y** del punto clave encontrado en la imagen de entrada
- La coordenada **x** del punto clave,
- **Valor de confianza** del punto clave.

Estos valores están normalizados entre 0 y 1.

Hay 17 de estos puntos clave devueltos según lo especificado por el valor del penúltimo eje. El orden de los puntos clave encontrados es: nose, left eye, right eye, left ear, right ear, left shoulder, right shoulder, left elbow, right elbow, left wrist, right wrist, left hip, right hip, left knee, right knee, left ankle, right ankle.

Para más información sobre el modelo MoveNet, y sus diferentes versiones, puedes consultar la ficha del modelo.

Objetivos

En este tutorial aprenderás a:

- Cargar un modelo bruto, el modelo movenet/singlepose/lighting
- Normalizar los datos de entradas de acuerdo a las necesidades del modelo
- Ejecutar el modelo bruto movenet/singlepose/lighting para realizar estimaciones sobre fotogramas de la cámara web e imágenes estáticas
- Visualizar la salida de Movenet (17 puntos clave del cuerpo)

Requisitos

- Una versión reciente de Chrome o de otro navegador actualizado.
- Tu editor favorito que se **ejecute localmente en tu máquina**, por ejemplo Visual Studio Code con la extensión Live Server, **o en la Web** con servicios como codesandbox.io o glitch.com.
- Familiaridad con JavaScript.
- Conocimientos básicos de Node.js.
- Familiaridad con las herramientas para desarrolladores de Chrome (DevTools) o de otro navegador.

Código

- Tutorial para cámara web
- Tutorial para imágenes estáticas

Tutorial para cámara web

En este primer tutorial aprenderás a realizar estimaciones con movenet/singlepose/lighting utilizando como entrada al modelo los fotogramas de la cámara web.

Paso 1. Esqueleto HTML

1.1. Cargar TensorFlow.js

Empieza creando el esqueleto de una página Web. Crea un archivo y nómbralo **index.html**. El archivo contendrá el siguiente código:

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <title>Tutorial movenet/singlepose/lighting</title>
   <meta charset="utf-8" />
   <meta http-equiv="X-UA-Compatible" content="IE=edge" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />

   <link rel="stylesheet" href="style.css" />
   <script
```

```
    src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"
      type="text/javascript"
    ></script>
</head>
<body>
  <h1>Tutorial movenet/singlepose/lighting</h1>
  <div id="outputMessage"></div>


  <script src="index.js"></script>
</body>
</html>
```

Observa que, a diferencia de los modelos prefabricados, no se incluye ningún fichero JavaScript adicional, como por ejemplo hiciste en el tutorial anterior para el modelo prefabricado COCO-SSD.

Crea un archivo **index.js** que inicialmente estará vacío.

1.2. Rellenar el esqueleto de la página HTML

En la etiqueta `body`, introduce también el código a continuación, justo antes de las etiquetas `script`:

```
  <section>
    <div>
      <canvas id="canvas" width="252" height="252"> </canvas>
      <video
        playsinline
        muted
        id="webcam"
        width="252px"
        height="252px"
      ></video>
    </div>
  </section>
```

1.3. Entender el código

Observa que en la etiqueta **section,** el espacio principal de la aplicación, hay **div** que contiene:

- El elemento `<canvas>` que utilizarás para visualizar la salida de MoveNet.
- En el elemento `<video>` se transmitirá la entrada de la cámara web, al igual que hiciste en el tutorial anterior. El tamaño del video es de 252x252 pixeles, pero podría ser de otros tamaños. Se ha elegido un tamaño cuadrado porque la entrada de

movenet/singlepose/lighting es una imagen cuadrada de 192x192 pixeles. De esta manera podemos visualizar más fácilmente la salida de la cámara y de MoveNet.

Para el elemento <video> añadimos en el fichero **style.css** una directiva para que transforme la imagen, rotándola de 180 grados en el eje y para así obtener un efecto de espejo.

```
video {
 transform: scaleX(-1);
}
```

Paso 2. Código JavaScript

El código en este paso se muestra en orden, pero se divide en partes para una explicación más detallada. Si copias y pegas cada segmento al final del archivo JavaScript cada vez, o reemplaza una función vacía definida del paso anterior como se recomienda, todo debería funcionar bien.

2.1. Referenciar los elementos del DOM

Primero, asegúrate de poder acceder a las partes clave de la página que necesitarás manipular o acceder más adelante, como el elemento video con la salida de la cámara web o el elemento canvas. Para esto, crearás las variables globales al principio del fichero **index.js**:

```
const webcamEl = document.querySelector("#webcam");
const canvas = document.querySelector("#canvas");
```

2.2. Declarar la URL del modelo

También crearás otra variable global que contiene la URL del modelo movenet/singlepose/lighting: esta información se puede recuperar en [la página del modelo en TFHub] (ahora alojado en Kaggle):

```
const MODEL_URL =
"https://tfhub.dev/google/tfjs-model/movenet/singlepose/lightning/4";
```

2.3. Definir la función principal de la aplicación

Como hemos estado trabajando hasta ahora, el código que utilizaremos contendrá llamadas a funciones asíncronas. Para esto, creamos una función principal asíncrona app(), que contendrá nuestra aplicación:

```
async function app() { }
```

Escribiremos el código fuente principalmente en esta función.

Paso 3. Cargar el modelo de aprendizaje automático

Ahora estás listo para cargar el modelo movenet/singlepose/lighting.

3.1. Cargar movenet/singlepose/lighting
Para cargar el modelo, tendrás que añadir el siguiente código al principio de la función app(), en el fichero **index.js**.

```
async function app() {
 const model = await tf.loadGraphModel(MODEL_URL, { fromTFHub: true });
 …
}
```

Observa que estamos utilizando el método tf.loadGraphModel() y que el segundo parámetro identifica que el modelo ha sido cargado de TFHub. Este segundo parámetro es esencial para cargar correctamente modelos que están alojados en TFHub (Kaggle). Sin el atributo fromTFHub: true, obtendríamos un error en las políticas CORS al intentar cargar el modelo.

Paso 4. Capturar fotogramas con la camara Web

Ahora pasamos a capturar los fotogramas de la cámara Web, transformándolos en tensores para así poder utilizarlos como entrada para nuestro modelo de detección de pose. Recordad que movenet/singlepose/lighting acepta como entrada un tensor [1,192,192,3].

4.1. Instanciar el objeto para generar tensores
Primero, es necesario instanciar el objeto para generar tensores a partir de la cámara web. Como hemos visto en el tutorial anterior, para esto es necesario incluir el código continuación en la función app, utilizando tf.data.webcam():

```
async function app() {
 const model = await tf.loadGraphModel(MODEL_URL, { fromTFHub: true });
 const webcam = await tf.data.webcam(webcamEl, {
   resizeWidth: 192,
   resizeHeight: 192
 });
 …
}
```

El segundo parámetro del método tf.data.webcam es un objeto que define la configuración de la cámara web. Con los atributos resizeWidth y resizeHeight estamos pidiendo que cada fotograma capturado por la cámara web, independientemente de su resolución, se escale a una imagen de 192x192 píxeles. Esta es la configuración más rápida para nuestro caso. Podría no ser la más eficaz: por ejemplo podría darnos mejores resultados recortar la imágen desde el centro, utilizando el atributo centerCrop, y valores para resizeWidth y resizeHeight iguales al menor entre altura y anchura de la resolución de la cámara web, y

escalar la imagen a 192x192 después, utilizando por ejemplo uno de los métodos que nos proporciona el API de TensorFlow.js, como tf.image.resizeBilinear().

A continuación, podemos empezar a generar los tensores para cada fotograma con el siguiente código, siempre en la función app():

```
while (true) {
   const img = await webcam.capture();

   img.dispose();
   await tf.nextFrame();
 }
```

Como vimos en el primer tutorial, el método capture() (es un método asíncrono) retorna una promesa que se resuelve en el siguiente fotograma, un tensor que representa la imágen. El método tf.nextFrame() espera a que esté disponible el siguiente fotograma para que el bucle continúe. Recordad que es necesario eliminar los tensores de la memoria manualmente si no queremos generar problemas de memoria, especialmente si estamos escribiendo nuestro código en un bucle infinito. Hubiésemos podido utilizar también la función tf.tidy().

Paso 5. Realizar estimaciones

Podemos ahora empezar a realizar las estimaciones de las articulaciones del cuerpo utilizando MoveNet, y mostrar el resultado por encima de cada fotograma.

5.1. Detectar pose en cada fotograma
Añade el código a continuación para que el modelo pueda detectar la pose en cada fotograma:

```
while (true) {
    const img = await webcam.capture();

   // Prediction
   const prediction = await model.predict(img.toInt().expandDims());
   const arrayOut = await prediction.array();
   const points = arrayOut[0][0];

   img.dispose();
   await tf.nextFrame();
 }
```

5.2. Análisis del código
La llamada realmente importante en este nuevo código es model.predict(). Todos los modelos pre-entrenados cargados a través del API tf.loadGraphModel() tienen una función

como esta para realizar la inferencia de aprendizaje automático. El modelo necesita como entrada un tensor [1, 192, 192, 3], pero el tensor devuelto por el método capture(), que guardamos en la variable img, es un tensor [192, 192, 3]. Si pasamos este tensor al modelo obtendríamos un error, porque el tensor no es del mismo tamaño de entrada que se espera el modelo.

```
▸Uncaught (in promise) Error: The shape of dict['input'] util_base.js:153
provided in model.execute(dict) must be [1,192,192,3], but was [192,192,3]
    at Hv (util_base.js:153:11)
    at graph_executor.js:629:9
    at Array.forEach (<anonymous>)
    at t.checkInputShapeAndType (graph_executor.js:620:25)
    at t.execute (graph_executor.js:198:10)
    at t.execute (graph_model.js:438:34)
    at t.predict (graph_model.js:320:32)
    at app (index.js:27:36)
```

Además, necesitamos hacer un casting del tensor al tipo de dato “int32”, a través del método toInt(), porque el tipo de dato del tensor retornado por el método capture es “float32”, mientras que la entrada de movenet/singlepose/lighting tiene que ser “int32”. De no hacer el casting, obtendríamos el siguiente error:

```
▸Uncaught (in promise) Error: The dtype of dict['input'] util_base.js:153
provided in model.execute(dict) must be int32, but was float32
    at Hv (util_base.js:153:11)
    at graph_executor.js:636:9
    at Array.forEach (<anonymous>)
    at t.checkInputShapeAndType (graph_executor.js:620:25)
    at t.execute (graph_executor.js:198:10)
    at t.execute (graph_model.js:438:34)
    at t.predict (graph_model.js:320:32)
    at app (index.js:27:36)
```

La variable prediction es un tensor [1,1,17,3] que contiene el resultado de la detección de la pose en el fotograma. Como se ha adelantado, cada predicción consiste en un valor **y**, un valor **x** y un **valor de confianza** para cada uno de los 17 puntos clave del cuerpo. Las coordenadas **yx** están normalizadas entre 0 y 1. Podemos obtener estos valores para cada punto como un array de 17 elementos, donde cada elemento es un array de tres elementos(y, x, confianza). Para esto utilizamos el método de tensores arraySync() que retorna el tensor como un array anidado. Recordad que la salida de movenet/singlepose/lighting es un tensor [1, 1, 17, 3], por lo que nuestro array resultado será el elemento en posición [0][0].

Paso 6. Mostrar la salida de MoveNet

Por último, necesitamos mostrar la salida del modelo, es decir los 17 puntos del cuerpo detectados en los fotogramas.

6.1. Dibujar círculos para los puntos clave del cuerpo

Primero, para poder dibujar los puntos clave en los fotogramas, necesitamos mover el canvas a la misma posición del elemento **video** en la página HTML. Ponemos el siguiente código justo después de haber creado el objeto webcam en la función app():

```
const camerabbox = webcamEl.getBoundingClientRect();
canvas.style.top = camerabbox.y + "px";
canvas.style.left = camerabbox.x + "px";
```

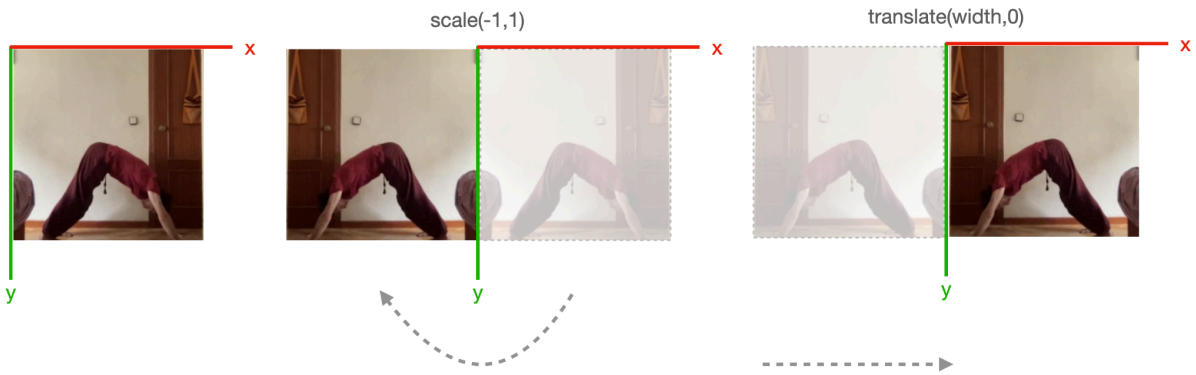
A continuación, recuperamos el contexto del canvas para poder dibujar en ella:

```
const context = canvas.getContext("2d");
```

Y por último, recordad que la imagen de la cámara web está rotada de 180 grados en el eje y para obtener una imagen espejo. Por esta razón, tengo que realizar también un espejo del canvas, incluyendo el siguiente código:

```
context.translate(webcamEl.width, 0);
context.scale(-1, 1);
```

Un valor de -1 en el primer parámetro del método **scale** corresponde a una rotación de 180 en el eje y.



Creamos una función para dibujar círculos del tamaño y del color que nos guste:

```
function drawCircle(context, cx, cy, radius, color) {
 context.beginPath();
 context.arc(cx, cy, radius, 0, 2 * Math.PI, false);
 context.fillStyle = "red";
 context.fill();
 context.lineWidth = 1;
 context.strokeStyle = color;
 context.stroke();
}
```

Y luego añadimos el siguiente código en el bucle de recuperación y detección, dentro de nuestra función app:

```
…
const points = arrayOut[0][0];
context.clearRect(0, 0, canvas.width, canvas.height);
for (let i = 0; i < points.length; i++) {
  const point = points[i];

  if (point[2] > 0.4) {
    drawCircle(context, point[1] * 252, point[0] * 252, 5, "#003300");
  }
}

img.dispose();
…
```

6.2. Análisis del código

El método clearRect() del objeto con el contexto del canvas sirve para limpiar el canvas. Necesitamos llamarlo en cada iteración del bucle: este método borra los píxeles en un área rectangular, configurándolos en negro transparente. Luego, para cada uno de los 17 puntos clave del cuerpo, cuya información está en el array points, dibujamos un círculo en el canvas utilizando la función drawCircle(). Recordad que los valores yx están normalizados, por lo que tendremos que multiplicarlos por el tamaño del canvas (en este caso 252px) si queremos que se dibujen en la posición correcta. Dibujaremos el círculo correspondiente al punto clave únicamente si el valor de confianza es mayor que un umbral establecido (por ejemplo 0.4). El modelo retorna siempre un valor yx para los 17 puntos, independientemente de si están o no en la imagen. Sin embargo, si no están, su valor de confianza será más próximo al 0.

Ejercicios

1. Identificar y resaltar articulaciones específicas. Modifica el código para que solo las articulaciones de los hombros, codos y muñecas sean resaltadas en el canvas, utilizando un color diferente para cada articulación. (Pista: Mira la ficha del modelo para conocer el orden de cada punto de articulación en los resultados)
2. Calcular y mostrar el ángulo del codo. Calcula el ángulo entre el hombro, el codo y la muñeca para ambos brazos y muestra los ángulos en el canvas. (Pista: Puedes utilizar el producto punto entre los vectores hombro-codo y codo-muñeca para calcular el ángulo)