Tutorial 6

Generación de texto con Transformers.js

Introducción

En este tutorial, aprenderás cómo usar Transformers.js para integrar modelos generativos de lenguaje en aplicaciones web. Aprenderás a generar prompts contextualizados para interactuar con el modelo y recibir respuestas útiles.

Objetivos

En este tutorial aprenderás a:

- Al completar este tutorial, sabrás cómo:
    - Cargar y utilizar un modelo de lenguaje como Qwen2.5-0.5B desde Transformers.js.
    - Gestionar la generación progresiva de texto en tiempo real.
    - Utilizar hilos de trabajo (workers) para procesamiento asincrónico.
    - Diseñar prompts efectivos para obtener respuestas contextuales.

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.
- Familiaridad con las herramientas para desarrolladores de Chrome (DevTools) o de otro navegador.

Código

- Código tutorial

Tutorial

Transformers.js es una librería que permite utilizar modelos de lenguaje basados en transformers directamente en el navegador o en entornos de JavaScript. Estos modelos son capaces de realizar tareas avanzadas de procesamiento de lenguaje natural (NLP), como generación de texto, clasificación, traducción, entre otras. Transformers.js aprovecha la flexibilidad y potencia de WebAssembly para ejecutar modelos sin necesidad de backend, brindando una experiencia ágil y segura.

En este tutorial aprenderás a utilizar Transformers.js para implementar un sistema de generación de texto basado en prompts personalizados. Usaremos un modelo preentrenado para responder de manera progresiva a entradas del usuario, lo que permite adaptarse a diversos contextos, como tutorías interactivas, asistentes virtuales o herramientas creativas de escritura. Además, aprenderás a utilizar el API Streamers, una herramienta poderosa que permite procesar y mostrar tokens de texto generados en tiempo real, proporcionando respuestas rápidas e interactivas. Este enfoque es particularmente útil en aplicaciones donde la velocidad de respuesta y la experiencia de usuario son esenciales.

La idea principal es la siguiente: proporcionaremos un prompt inicial al modelo que defina el contexto y las instrucciones de la tarea. A medida que el modelo genera la respuesta, los streamers manejarán los tokens generados y los enviarán progresivamente al navegador para mostrarlos al usuario en tiempo real. Esto crea una experiencia fluida y dinámica que mejora la interacción con el sistema.

En este tutorial, también utilizaremos el API Web Workers para ejecutar la generación de texto en un hilo separado. Esto asegura que la interfaz de usuario se mantenga receptiva incluso durante procesos intensivos de generación de texto.

Con este enfoque, no solo aprenderás los conceptos básicos de Transformers.js, sino que también comprenderás cómo integrarlo eficientemente en una aplicación web moderna para crear experiencias únicas y personalizadas.

Paso 1. Esqueleto HTML

1.1. Rellenar el esqueleto de la página HTML

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>
   <meta charset="UTF-8" />
   <link rel="icon" type="image/png" href="/logo.png" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Transformers.js</title>
```

```
 </head>
 <body>
   <div id="root">
     <p id="outputMessage"> </p>
   </div>

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

Este archivo define la estructura básica de la interfaz web, proporcionando un espacio donde se mostrará el texto generado en una etiqueta <p>. Este archivo HTML es el punto de partida para la integración del modelo de lenguaje en la aplicación web. Asegúrate de que esté correctamente configurado antes de avanzar.

Paso 2. Código JavaScript

Podemos empezar a crear nuestra aplicación, escribiendo el código en el fichero **index.js**. Este archivo contiene la lógica principal de la aplicación y la comunicación con un procesador en segundo plano (worker).

```
const outputMessageEl = document.getElementById('outputMessage');

const worker = new Worker('worker.js', { type: "module" }); // Path to your
worker file
worker.postMessage({ type: 'load' });

worker.onmessage = async (e) => {
 switch(e.data.type) {
   case 'token':
     console.log(e.data.token);
     outputMessageEl.textContent += e.data.token;
     break;
 }
};
```

El hilo principal crea un worker (new Worker) y establece un canal de comunicación bidireccional mediante **postMessage()** y **onmessage()**. Se crea un **new Worker** indicando el fichero del procesador en segundo plano (**worker.js**) para manejar tareas de generación de texto de forma asincrónica. El trabajador envía los tokens generados de vuelta al hilo principal, que se añaden al contenido del elemento con id **outputMessage** (el elemento con etiqueta **<p>**).

Se utiliza un procesador en segundo plano para ejecutar las operaciones de generación de texto de manera asincrónica y mantener la interfaz de usuario receptiva. Esto se logra mediante la comunicación entre el hilo principal (archivo index.js) y el worker (archivo **worker.js**) usando los métodos **postMessage()** y **onmessage()**.

El método **postMessage()** se utiliza para enviar un mensaje al worker. En este caso, el mensaje tiene un tipo load para indicarle al worker que inicie la carga del modelo.

El método **onmessage()** captura los mensajes enviados desde el worker hacia el hilo principal. Contiene los datos enviados por el worker dentro de la propiedad **e.data**. En este ejemplo se utiliza una estructura switch para procesar diferentes tipos de mensajes (**e.data.type**). Si el tipo es **“token”**, significa que el worker ha generado un token y lo envía al hilo principal para ser mostrado o registrado. En el código, se añade el token generado al contenido del elemento HTML identificado por el id **outputMessage**. Esto permite mostrar de manera progresiva los resultados generados por el modelo en la interfaz del usuario.

Ahora, incluye un nuevo caso para manejar un mensaje de tipo ”**ready**”. Este caso se utiliza para indicar que el worker ha completado la inicialización (como cargar el modelo) y está listo para procesar nuevas solicitudes.

```
worker.onmessage = async (e) => {
 switch(e.data.type) {
   case 'token':
     console.log(e.data.token);
     outputMessageEl.textContent += e.data.token;
     break;

   case 'ready':
     console.log('Ready');
     worker.postMessage({ type: 'generate' });
     break;
 }
};
```

Paso 3. Lógica del procesador en segundo plano

El archivo **worker.js** actúa como un procesador en segundo plano que se comunica con el hilo principal para realizar tareas de generación de texto utilizando Transformers.js. Este diseño utiliza el enfoque de Web Workers para mantener la interfaz de usuario receptiva y manejar procesos asincrónicos intensivos sin bloquear el hilo principal.

```
self.onmessage = async (e) => {
   console.log(e);
   switch (e.data.type) {
       case 'load':
           await load();
           break;
```

```
        case 'generate':
            await generate();
            break;
    }

};
```

El worker recibe mensajes desde el hilo principal mediante el evento **self.onmessage()** y responde usando **self.postMessage()**. La lógica del worker está dividida en dos tareas principales: carga del modelo (load) y generación de texto (generate).

Mediante un manejador **onmessage**, el worker escucha mensajes del hilo principal. El mensaje debe contener un campo **type** que indica qué tarea debe realizar el worker: **"load"** inicializa el modelo de generación y configura el streamer, "**generate"** genera texto basado en un prompt predefinido y lo procesa mediante el streamer.

3.1. Cargar el modelo

Configuramos el modelo de generación de texto y preparamos el código para la generación progresiva a través del API Streamers.

```
import { TextStreamer, pipeline } from
"https://cdn.jsdelivr.net/npm/@huggingface/transformers@3.1.0";

const TASK_NAME = "text-generation";
const MODEL_NAME = "onnx-community/Qwen2.5-0.5B-Instruct";

let generator = null;
let streamer = null;
```

Importamos las funciones y clases **TextStreamer** y **pipeline** desde la librería Transformers.js, alojada en la red de distribución de contenidos. **pipeline** es la función principal del API de Transformers.js, y facilita el acceso a modelos de lenguaje preentrenados para tareas específicas, como generación de texto. Se usa para configurar el modelo de lenguaje asociado con la tarea de generación de texto. **TextStreamer** es una clase que permite procesar y transmitir tokens generados de manera progresiva y en tiempo real. Se utiliza para gestionar la salida progresiva de texto token por token.

Definimos dos variables:

- **TASK_NAME** define el nombre de la tarea que se ejecutará con el modelo. **"text-generation"** indica que el modelo será utilizado para generar texto en respuesta a un prompt. Es un parámetro pasado a **pipeline** para especificar que se ejecutará una tarea de generación de texto.

- **MODEL_NAME** especifica el identificador del modelo que será utilizado. Es un parámetro pasado a **pipeline** para cargar y configurar el modelo correspondiente.

Declaramos otras dos variables que serán utilizadas para almacenar las instancias configuradas de:

- **generator**: El objeto retornado por **pipeline**, que representa el modelo de generación de texto.
- **streamer**: Una instancia de **TextStreamer** para gestionar la salida de texto progresiva.

Inicialmente, ambas variables están declaradas como **null** y se inicializan más adelante en la función **load()**.

```
async function load() {

    generator = await pipeline(
        TASK_NAME,
        MODEL_NAME,
        { dtype: "fp16", device: "wasm", }
    );

    streamer = new TextStreamer(generator.tokenizer, {
        skip_prompt: false,
        callback_function,
    });

    // WARM-UP: Perform a dummy inference
    await generator("Warm up", {
        max_new_tokens: 1
    });

    self.postMessage({ type: "ready" });
}
```

En esta función:

1. Cargamos el modelo: se utiliza la función **pipeline()** de Transformers.js para inicializar un modelo de generación (**Qwen2.5-0.5B-Instruct**). Configuramos parámetros como **dtype: "fp16"** y **device: "wasm"** para optimizar el rendimiento.
2. Configuramos el streamer: se usa **TextStreamer** para procesar y transmitir tokens generados en tiempo real. Se define la función **callback_function** para manejar tokens a medida que se generan.
3. Realizamos un warm-up: se realiza una inferencia mínima para precalentar el modelo y mejorar los tiempos de respuesta posteriores.

4. Notificamos la disponibilidad: se envía un mensaje de tipo **“ready”** al hilo principal para indicar que el modelo está listo para usar.

La función de devolución de llamada **callback_function()** permite enviar tokens generados de forma progresiva al hilo principal. Cada vez que el streamer genera un token, lo procesa esta función, que envía el token al hilo principal con un mensaje de tipo **“token**”.

```
function callback_function(token) {
   self.postMessage({ type: "token", token });
}
```

3.2. Generación de texto

La función **generate()** nos permite generar texto en base a un prompt predefinido.

```
async function generate() {
   const prompt = `You are an English tutor. Respond to students based on
their language level and the specific topic of interest they mention.
Always include examples or explanations directly related to the topic.

Example 1:
Student level: Beginner
Topic: Food
Student: "How do I order food in English?"
Tutor: "You can say: 'Can I have a pizza, please?' or 'I would like a
burger, please.' Practice saying these phrases!"

Example 2:
Student level: Intermediate
Topic: Travel
Student: "What do I say when I need to find my gate at the airport?"
Tutor: "You can ask: 'Excuse me, where is gate 25?' or 'Can you help me
find my gate, please?' These phrases are useful at airports."

Now, respond to this student:
Student level: Beginner
Topic: Food
Student: "How can I start a conversation with locals when traveling
abroad?"
Tutor:
`;

   const output = await generator(prompt, {
```

```
        max_new_tokens: 64,
        //temperature: 0.5,
        //top_p: 0.5,
        do_sample: false,
        early_stopping: true,
        streamer
    });

    console.log(output);
}
```

Proporcionamos un contexto detallado al modelo, a través de la variable **prompt**, para guiar la generación de texto. El prompt incluye ejemplos para ilustrar el formato esperado de las respuestas.
La función llama al modelo con el prompt y parámetros ajustables como:

- **max_new_tokens**: Limita la longitud del texto generado.
- **top_p**: Controla el filtro de probabilidad acumulativa (nucleación). No se usa si **do_sample** es **false**.
- **early_stopping**: Detener la generación de texto de manera anticipada si el modelo detecta que ya ha completado una respuesta coherente antes de alcanzar el límite de tokens
- **do_sample**: Determina si se realiza muestreo estocástico.
- **streamer**: Procesa los tokens a medida que se generan

Podéis encontrar todas las opciones disponibles en la documentación de la librería.
Se muestra también el resultado completo en la consola para depuración.

Paso 4. Estrategias de prompt

En este ejemplo se ha utilizado la estrategia de prompt few-shot learning, que consiste en proporcionar al modelo ejemplos específicos dentro del texto de entrada (prompt) para que comprenda mejor la tarea antes de generar una respuesta. En esta técnica, el prompt incluye uno o más ejemplos de entrada y salida que sirven como guía para que el modelo adapte su comportamiento a la tarea deseada. Por ejemplo, en el caso del código presentado, se le muestra al modelo cómo un tutor responde a estudiantes de diferentes niveles sobre temas específicos.

Ventajas:

1. Flexibilidad: No requiere modificar el modelo ni entrenarlo nuevamente, lo que ahorra tiempo y recursos.
2. Adapatabilidad: Se puede ajustar rápidamente a diferentes tareas o contextos simplemente cambiando los ejemplos en el prompt.
3. Resultados Contextuales: Los ejemplos proporcionan un contexto claro, lo que puede mejorar significativamente la calidad y relevancia de las respuestas generadas.

4. No requiere datos adicionales: Es ideal cuando no se dispone de un gran conjunto de datos etiquetados para la tarea.

Desventajas:

1. Límite de longitud: Los modelos tienen un límite en la longitud del contexto que pueden procesar, por lo que el número de ejemplos incluidos en el prompt es limitado.
2. Inconsistencia: La calidad de las respuestas puede variar según los ejemplos y cómo se estructuren en el prompt.
3. Dependencia del diseño del prompt: Requiere diseñar prompts cuidadosamente, lo que puede ser un desafío, especialmente en tareas complejas.

Otra estrategia podría ser Chain-of-Thought (CoT), que incluye ejemplos donde el modelo es guiado a razonar paso a paso para llegar a una respuesta. Mejora el rendimiento en tareas que requieren razonamiento lógico o matemático, pero puede ser más complicado diseñar este tipo de prompts.

Podéis encontrar detalles sobre las distintas estrategias de prompt aquí.