TensorFlow.js with WebWorker

Introduction

Personal Information

Full Name	Wenhe Li
Institute	New York University Shanghai
Email	wl1508@nyu.edu
Phone	+86-18605355676 +1-6462888624
Blog	https://blog.steins.live
Github	https://github.com/WenheLi
Portfolio	http://portfolio.steins.live

About Me

I am Wenhe Eric Li, a junior from NYU Shanghai double majors in Computer Science & Interactive Media Arts. I have experience in multi-programming languages like c++, Java, JavaScript, Python, Kotlin, Dart and Go. I like trying new stuff and making

fantastic projects. Also, I got inspired by Prof. Daniel Shiffman to step into the open source world to make some contributions.

During the last semester, I have made a gif library for p5.js which is a drawing library based on p5.js. Also, I have made some attempts to contribute to the TensorFlow.js and ml5 (a library on top of tfjs). It's like from this moment I really get into the open source world and become a part of them!

As for my background, I have taken the Software Engineering, Open Source Studio and some basic Machine Learning stuff like cs231n on myself and instructed by Prof. Zhen Zhang. I have a lot of experience with native app development, machine learning basics, React and React Native as well as the TypeScript/JavaScript thing.

Why GSoC with TensorFlow.js?

Personally speaking, I can see the potential of making Machine Learning happens on the web. By library like TensorFlow.js, we can build up new Web Apps that utilize the benefits of fancy deep learning stuff.

The above is from the emotional part, and the other side, by now the Web can have a competitive Computation power compared with the native app in some aspects.

Synopsis

The TensorFlow.js will be highly computational consuming while we are doing prediction or classification. And if such a computation happens at the UI threads, it will block all the things until the computation is done. The above will result in a bad experience while using the web to render the results and so on. And it is natural to come up with the idea that is doing a multi-threads model that let the main thread(ui thread) run for rendering UI and listen to the results from other threads and let another thread opens when the computation is needed. This WebWorker is the base for the multi-thread model over the browser. We can use the WebWorker to achieve the paradigm that separates the heavy computation apart from UI rendering.

Solution Discussion

To solve the above problem, we need to use WebWorker with TensorFlow.js. Below is a code snippet that how will it look like:

```
// At the ui thread index.js
//...
const worker = new Worker('./worker.js'); //create new thread when it is
needed
worker.postMessage({data}); // here post the img/video data to the backend
thread for computation

// register onMessgae listener for any incoming results and print it out in
console.
worker.onmessage = (e) => {
      console.log(e.data);
}
//...

// At the computation thread worker.js
importScripts('https://cdn.jsdelivr.net/npm/setimmediate@1.0.5/setImmediate
.min.js');
importScripts('https://cdn.jsdelivr.net/npm/@tensorflow/tfjs');
importScripts('https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.
0.0');

// Above import necessary tfjs scripts

// Register listener when there is an incoming data
self.onmessage = (event) => {
      mobilenet.load(2, .5).then(model => {
      const canvas = document.createElement('offscreenCanvas');
      canvas.getContext('2d').drawImage(event.data, 0, 0); // put incoming
data to canvas
      let res = model.classify(canvas);
      res.then((it) => {
            postMessage({data}); // return results to the main thread
        })
      });
}
```

Current Problem

The above code looks solid enough, while one thing needs to be mentioned that the environment is quite different between the **index.js** and the **worker.js** that will result

in failure if we want to run the above code. The major difference here is that the secondary thread will not have anything related with **window** and **documents** which are essential for tfjs running prediction. While the good news is that we can forge the document and window to achieve our goal by adding the following lines:

```
    self.document = {
        createElement: () => new OffscreenCanvas(640, 480);
      };
    self.window = self;
    self.HTMLVideoElement = function() {};
    self.HTMLImageElement = function() {};
    self.HTMLCanvasElement = OffscreenCanvas;
```

So basically, we just need to follow the above code add up things that are missing in the WebWorker thread and it will be safe and sound.

While I am thinking if we can move forward, either by wrapping the WebWorker with the pre-set document or by letting the TensorFlow.js identities the WebWorker environment and calls different APIs.

Pre-set Document

For the pre-set document, it is quite straightforward from the big picture that we just need to write a set of documents to describe what users need to put to ensure a perfect working condition for WebWorker. While we still need to notice, the tfjs is about a cross-platform framework that will work on mobile, different browsers and even Raspberry PI. This means a lot of extra work on experimenting on compatibility and documents as well as detailed tutorials.

TFjs WebWorker API

This is about to build a class inside TensorFlow.js to create a customized WebWorker with all the essential objects inside. In my design, the API should look like:

```
const preset = {
      document: document
}; //preset is for registering external APIs

const path = './work.js' //path to the WebWorker script

const worker = tf.WebWorker(path, preset); // You can also put a callback
function at last, it will be called when the Webworker script starts
```

```
//post data
worker.post();

//register listener for onmessage
worker.addEventListener('message', () => {
      ...
});
```

This solution requires a lot of extra work on how to hack the WebWorker mechanism. Also, one thing that Daniel points out that, a new set of API will make it harder to get used to and decline the stability of tf-core.

Project Goals

Objectives

The overall objectives will be the following:

- Develop detailed documents and examples for how to use `WebWorker` in TF.js for different use cases.
- A blog post for the `WebWorker` from a big view.
- Setup a Unit test guideline for Multi-process programming in TF.js.
- A WebGL supports for TF.js.

One more additional work would be if we want to develop a custom **WebWorker** class for TF.js.

Tasks

1. **Documents, examples and Blog post**

   This part will be comprehensive documents on how to use WebWorker with TF.js in different use cases like Browsers, mobile, and node environment. Besides, I will also include a tutorial on how to use pure **WebWorker** to achieve the multi-thread thing. Since I have a background in **`Software Engineering`**, it won't be a big problem.

2. WebGL supports

Since the **WebWorker** does not support the WebGL environment, we need to write some support code to make it possible. And without WebGL, the TFjs using the CPU backend which makes the predicting process extremely slow. That means WebGL support is necessary but it also requires some extra work on building up **WebGL** code base for **WebWorker**. As I have a Computer Graphics background, playing with **WebGL** won't be a major problem.

Here is a reference post for the WebGL implementation with **WebWorker**.

One more task is about the solution I have proposed in the previous section, which is about the **TFjs WebWorker API:**

By having such a `Custom WebWorker` class, we can make the code clear and well-organized and the below is the set of API that I proposed for TF.js.

```
class tfWoker {
    constructor(src, preset) {
        this.worker = new Worker(src);
        const obj = {
            document: document,
            HTMLVideoElement: function(){},
            HTMLImageElement: function(){},
            HTMLCanvasElement: OffscreenCanvas,
        };
        this.obj = Object.assign(obj, preset);
        this.worker.postMessage({obj: this.obj});
        this.worker.addEventListener('message', (event) => {
            if(event.data.init) this.worker.removeEventListener('message');
        });
    }

    addEventListener(type, cb) {
        this.worker.addEventListener(type, cb);
    }

    remveEventListener(type, cb) {
        this.worker.remveEventListener(type, cb);
    }
    postMessage(msg) {
        this.worker.postMessage(msg);
    }
}
```

```
// In WebWorker

self.onmessage = (event) => {
    // init the environment for WebWorker
    if (! self.window) {
        const obj = event.data.obj;
        self.document = obj.document;
        self.window = self;

        self = Object.assign(self, obj);
        importScripts(JSON.stringify(this.obj)); //import incoming scripts into
environment
        self.postMessage({init: true});
    } else {
        //normal code like what happens in browser
     ...
    }
}
```

The code is still a prototype and some design is not perfect, while I think they will demonstrate a pattern of how to call the WebWorker in my mind. A more detailed discussion should happen internally for future development.

The shortage here is about the WebWorker code that we still need to hardcode a bit, if we can dynamically generate it then the code will be much cleaner and such a thing can be done easily. And this shortage can be solved by calling `importScripts('pure js scripts')`.

Timeline

- Before Start
    - I will work on the API designing and WebGL environment preparation for WebWorker.
- Week 1 && Week 2

    - Based on the API design and discussion within the community, I will start to build up the `tf.utils.WebWorker` and will have a daily update with the code and share the class with the community to do some user testing.
    - Or I will work on various examples using **WebWorker** under different use cases and write general guidelines on the philosophy of making TF.js multi-threading.
- Week 3
    - A time period for any unexpected delay. And for the Phase 1 evaluation.
- Week 4 && Week 5
    - From this week, I should successfully develop the class/examples. I will start work on the documentation and unit tests. And start to merge the class to the official repo to get a public test for bugs and API design.
- Week 6
    - A time period for any unexpected delay and debugging. And for the Phase 2 evaluation.
- Week 7 && Week 8
    - From this week, I will work on the WebGL thing. While this is a tremendous work that may go beyond the scope for GSoC, I think I will start on building a WebGL codebase at least that allows the WebGL is operatable in the **WebWorker environment**. Moreover, if we still have time, we can continue to use the codebase to build up a connection between the WebGL and TensorFlow operations.

Deliverables

- Fully functioned tf.WebWorker class for CPU backend. (Optional)
- Various examples under different use cases.
- Well written documentation and tutorials.
- At least a WebGL codebase for WebWorker. If we can conduct it with TensorFlow WebGL backend it will be perfect.

Future Goals

The future work is more about WebGL supporting since it is really important!