# 项目7 预测汽车油耗效率：TensorFlow.js应用开发

项目描述

TensorFlow.js 是一个用于使用 JavaScript 进行机器学习开发的库，用于在浏览器和 Node.js 训练和部署机器学习模型。TensorFlow.js支持使用JavaScript在浏览器端部署，也与微信小程序有很好的集成。TensorFlow.js支持所有Python可以加载的模型，在Node.js环境中直接通过调研API即可，在浏览器环境中需要转换为浏览器支持的JSON格式。TensorFlow.js提供了一系列预训练好的模型，包括图像识别、语音识别、人体姿态识别、物体识别、文字分类等。

本项目将通过预测汽车油耗效率、手写数字识别两个项目介绍使用TensorFlow.js进行 TensorFlow 模型的开发、训练和部署，体验TensorFlow.js让用户直接在浏览器加载TensorFlow，通过本地CPU或者GPU资源进行所需的机器学习运算，灵活的进行各种AI应用的开发。

项目目标

知识目标

* 了解TensorFlow.js的优点
* 了解TensorFlow.js的相关概念
* 掌握TensorFlow.js环境配置

技能目标

* 能通过Layers API创建模型
* 能通过Core API创建模型
* 能在浏览器中使用TensorFlow.js
* 能在Node.js中使用TensorFlow.js
* 能熟练使用Node.js
* 能熟练的TensorFlow.js模型部署，开发相应AI应用

## 7.1 认识TensorFlow.js

TensorFlow.js 是 TensorFlow 的 JavaScript 版本，支持 GPU 硬件加速，可以运行在 Node.js 或浏览器环境中。它不但支持完全基于 JavaScript 从头开发、训练和部署模型，也可以用来运行已有的 Python 版 TensorFlow 模型，或者基于现有的模型进行继续训练。

### 7.1.1 TensorFlow.js优点

Google针对浏览器、移动端、IOT 设备及大型生产环境均提供了相应的扩展解决方案，TensorFlow.js 就是 JavaScript 语言版本的扩展，在它的支持下，前端开发者就可以直接在浏览器环境中来实现深度学习的功能。

在2017年，一个叫做DeepLearning.js的工程诞生了，这是一款基于 WebGL 加速的开放源代码 JavaScript 机器学习库，该库可以直接在浏览器中运行，而无需进行安装，也无需借助后端运行。

DeepLearning.js不仅通过利用 WebGL 在 GPU 上执行计算大幅提高了速度，同时还能够执行完整全面的反向传播。在2018年3月，DeepLearn.js团队与TensorFlow团队合并，重命名为TensorFlow.js。

浏览器环境在构建交互型应用方面有着天然优势，而端侧机器学习不仅可以分担部分云端的计算压力，也具有更好的隐私性，同时还可以借助 Node.js 在服务端继续使用 JavaScript 进行开发，这对于前端开发者而言非常友好。除了提供统一风格的术语和 API，TensorFlow 的不同扩展版本之间还可以通过迁移学习来实现模型的复用，或者在预训练模型的基础上来定制自己的深度神经网络。

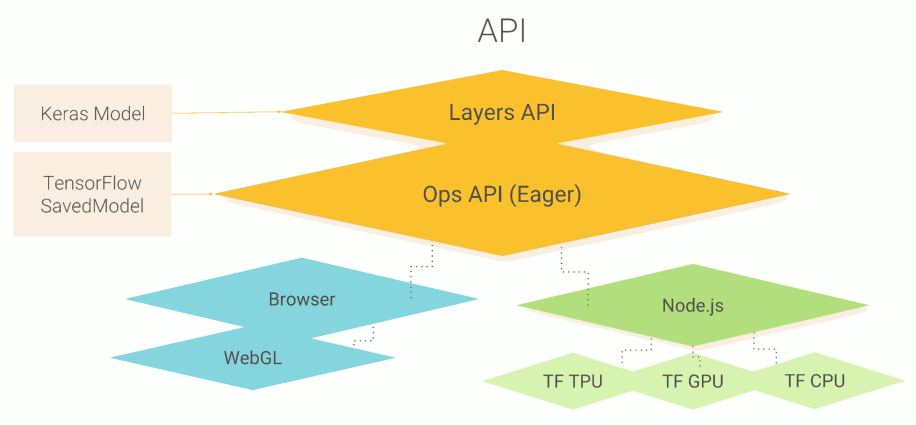


图7-1 TensorFlow.js架构

TensorFlow.js架构如图7-1，在 TensorFlow.js 中可以使用底层Core API或最高级的Layers API在浏览器上开发模型，也能基于浏览器运行已训练的模型。例如在网页端训练一个模型来识别图片或语音，训练一个模型以新颖的方式玩游戏或构建一个能创造钢琴音乐的神经网络等。

TensorFlow.js 支持 GPU 硬件加速。在 Node.js 环境中，如果有 CUDA 环境支持，或者在浏览器环境中有WebGL环境支持，那么TensorFlow.js可以使用硬件进行加速。

Tensorflow.js 还有如下优势：

1. Tensorflow.js 是开箱即用的开发库，开发者无需花精力去编写基础复杂的数学问题。
2. 由于可运行于浏览器，减少服务器的运算，提高服务器资源利用，增强客户端响应运算结果的速度。
3. 使用语言就是Javascript，前端工程师不需要学习其他后端语言，降低入门门槛。
4. 由于浏览器的 WebGL 可调用 GPU，所以 Tensorflow.js 会使用 GPU 加速模型的运算，提高运算效率。
5. Node 和 Python 一样都是使用 C++编写的环境，所以在 Node 环境进行运算的速度目前与 Python 速度不相上下。
6. Tensorflow.js 的模型可以跟 Python 等其他语言模型进行互转。
7. 浏览器可以很好可视化机器训练过程，同时浏览器可调用设备的摄像头、麦克风等增加机器学习的应用场景，让机器学习跟接近用户。

### 7.1.2 TensorFlow.js 的核心概念

TensorFlow.js 不仅可以提供低级的机器学习构建模块，还可以提供高级的类似 Keras 的 API 来构建神经网络。

在 TensorFlow.js 中可以通过两种方式创建机器学习模型：

1. 使用 Layers API（使用层构建模型）
2. 使用 Core API（借助低级运算，例如 tf.matMul()、tf.add() 等）

TensorFlow.js为机器学习提供低级构建模块，以及构建神经网络的高级Keras Layers API。 下面简要介绍一些核心组件。

1. 张量(Tensor)

TensorFlow.js 中的中心数据单元是张量（tensor）：一维或多维数组。一个 Tensor 实例的 shape 属性定义了其数组形状。

Tensor 主要构造函数是 tf.tensor 函数：

// 2x3 Tensor

const shape = [2, 3]; // 2 rows, 3 columns

const a = tf.tensor([1.0, 2.0, 3.0, 10.0, 20.0, 30.0], shape);

a.print(); // print Tensor values

// Output: [[1 , 2 , 3 ],

// [10, 20, 30]]

// The shape can also be inferred:

const b = tf.tensor([[1.0, 2.0, 3.0], [10.0, 20.0, 30.0]]);

b.print();

// Output: [[1 , 2 , 3 ],

// [10, 20, 30]]

1. 变量（Variable）

变量用张量的值进行初始化。 然而，与张量不同的是，它们的值是可变的。 您可以使用assign方法为现有变量分配一个新的张量：

const initialValues = tf.zeros([5]);

const biases = tf.variable(initialValues); // initialize biases

biases.print(); // output: [0, 0, 0, 0, 0]

const updatedValues = tf.tensor1d([0, 1, 0, 1, 0]);

biases.assign(updatedValues); // update values of biases

biases.print(); // output: [0, 1, 0, 1, 0]

1. 操作（Ops）

Tensor 可以用于保存数据，而操作则可用于操作数据。TensorFlow.js 提供了多种适用于张量的线性代数和机器学习运算的操作。由于 Tensor 是不可改变的，这些操作不会改变它们的值，而会返回新的 Tensor。这些运算不仅包含 add、sub 和 mul 等二元运算，同时还包括 square 等一元运算：

const e = tf.tensor2d([[1.0, 2.0], [3.0, 4.0]]);

const f = tf.tensor2d([[5.0, 6.0], [7.0, 7.0]]);

const e\_plus\_f = e.add(f);

e\_plus\_f.print();

// Output: [[6 , 8 ],

// [10, 12]]

const d = tf.tensor2d([[1.0, 2.0], [3.0, 4.0]]);

const d\_squared = d.square();

d\_squared.print();

// Output: [[1, 4 ],

// [9, 16]]

1. 模型和层

在 Tensorflow.js 有两种创建模型的方式：可以用高层API：Layers API来建立模型，也用Core API来搭建相同的模型。

Layers API有两种方式创建模型：第一种是创建 sequential 模型，第二种是创建 functional 模型。

Sequential 模型将网络的每一层简单的叠在一起。您可以将需要的层按顺序写在一个列表里，然后将列表作为 sequential() 函数的输入：

const model = tf.sequential({

layers: [

tf.layers.dense({inputShape: [784], units: 32, activation: 'relu'}),

tf.layers.dense({units: 10, activation: 'softmax'}),

]

});

也可以通过 tf.model() 来创建 LayersModel。可以用 tf.model() 来创建任何非闭环的计算图。以下是使用tf.model() API 建立和上文相同模型的列子：

// 用apply()方法创建任意计算图

const input = tf.input({shape: [784]});

const dense1 = tf.layers.dense({units: 32, activation: 'relu'}).apply(input);

const dense2 = tf.layers.dense({units: 10, activation: 'softmax'}).apply(dense1);

const model = tf.model({inputs: input, outputs: dense2});

在每一层用 apply() 将上一层的输出作为本层的输入。

不同于 sequential model 使用 inputShape 来定义第一层的输入，用 tf.input() 创建的 SymbolicTensor 作为第一层的输入。

Layers API 提供了大量方便的工具，例如权重初始化，模型序列化，训练监测，可迁移性和安全检查。当遇到如下情况时，可能会需要使用 Core API：

1. 需要更多灵活性和控制
2. 不需要序列化或可以创造自己的序列化方法

用 Core API 写的模型包含了一系列的函数。这些函数以一个或多个张量作为输入，并输出另一个张量。可以用 Core API 来重写之前定义的模型：

// The weights and biases for the two dense layers.

const w1 = tf.variable(tf.randomNormal([784, 32]));

const b1 = tf.variable(tf.randomNormal([32]));

const w2 = tf.variable(tf.randomNormal([32, 10]));

const b2 = tf.variable(tf.randomNormal([10]));

function model(x) {

return x.matMul(w1).add(b1).relu().matMul(w2).add(b2).softmax();

}

1. 内存管理

因为TensorFlow.js使用了GPU来加速数学运算，因此当tensorflow处理张量和变量时就有必要来管理GPU内存。在TensorFlow.js中，可以通过dispose 和 tf.tidy这两种方法来管理内存。

可以在张量或变量上调用dispose来清除它并释放其GPU内存：

const x = tf.tensor2d([[0.0, 2.0], [4.0, 6.0]]);

const x\_squared = x.square();

x.dispose();

x\_squared.dispose();

进行大量的张量操作时使用dispose可能会很麻烦。TensorFlow.js提供了另一个函数tf.tidy，它对JavaScript中的常规范围起到类似的作用，不同的是它针对GPU支持的张量。

tf.tidy执行一个函数并清除所有创建的中间张量，释放它们的GPU内存。 它不清除内部函数的返回值。

const average = tf.tidy(() => {

const y = tf.tensor1d([1.0, 2.0, 3.0, 4.0]);

const z = tf.ones([4]);

return y.sub(z).square().mean();

});

average.print()

使用tf.tidy将有助于防止应用程序中的内存泄漏。它也可以用来更谨慎地控制内存何时回收。

### 7.1.2 TensorFlow.js 环境配置

JavaScript项目中有两种主要的方式来获取TensorFlow.js：通过脚本标签（script tags）或从yarn（或者NPM）安装并使用Parcel，WebPack或Rollup等工具构建工程。如果您是 Web 开发新手，或者从未听说过诸如 Webpack 或 Parcel 的工具，建议您使用脚本代码。如果您比较有经验或想编写更大的程序，则可能值得使用构建工具进行探索。

1. 使用Script Tag

在浏览器中加载 TensorFlow.js ，最方便的办法是在 HTML 中直接引用 TensorFlow.js 发布的 NPM 包中已经打包安装好的 JavaScript 代码。

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@2.0.0/dist/tf.min.js"></script>

将下面的代码添加到HTML文件中，在浏览器中打开该HTML文件。

<html>

<head>

<!-- Load TensorFlow.js -->

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@2.0.0/dist/tf.min.js"></script>

<!-- Place your code in the script tag below. You can also use an external .js file -->

<script>

// Notice there is no 'import' statement. 'tf' is available on the index-page

// because of the script tag above.

console.log(tf.version.tfjs);

const a = tf.tensor([[1, 2], [3, 4]]);

console.log('shape:', a.shape);

a.print();

</script>

</head>

<body>

</body>

</html>

摁下F12键打开开发人员工具，可以方便调试自己的代码。可随时在任何网页上使用 F12 工具，从而快速调试 JavaScript、HTML 和级联样式表 (CSS)，还可以跟踪并查明网页或网络的性能问题。网页运行结果如图7-2所示。

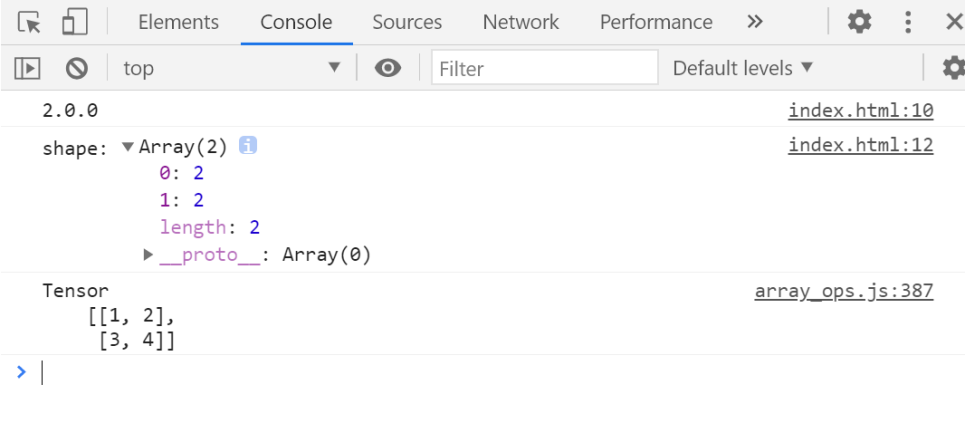


图7-2 运行结果

1. 通过parcel打包执行

服务器端使用 JavaScript ，首先需要按照 [NodeJS.org](https://nodejs.org/) 官网的说明，完成安装最新版本的 Node.js

1. 建立 TensorFlow.js 项目目录
2. 初始化项目管理文件 package.json

**$ yarn init**

yarn init v1.22.5

question name (test):

question version (1.0.0):

question description: TensorFlow.js test

question entry point (index.js):

question repository url: index.html

question author:

question license (MIT):

question private:

success Saved package.json

Done in 47.23s.

1. 在目录下创建两个文件，index.html,index.js

在index.html中通过script标签引入index.js就可以了，在index.js中写一段简单的测试代码。

index.html代码如下：

<html>

<body>

<h4>TFJS example<hr/></h4>

<div id="micro-out-div">TensorFlow.js Test</div>

<script src="./index.js"> </script>

</body>

</html>

index.js代码如下：

import \* as tf from '@tensorflow/tfjs'

console.log(tf.version.tfjs)

const shape = [2, 3]; // 2 rows, 3 columns

const a = tf.tensor([1.0, 2.0, 3.0, 10.0, 20.0, 30.0], shape);

a.print(); // print Tensor values

1. 修改package.json配置文件

如果使用 JavaScript、或者曾经与 JavaScript 项目、Node.js 或前端项目进行过交互，则肯定会遇到过 package.json 文件。package.json 文件是项目的清单，它可以做很多完全互不相关的事情。它是用于工具的配置中心，也是 yarn 存储所有已安装软件包的名称和版本的地方。开发依赖是仅用于开发的程序包，在生产环境中并不需要。例如测试的软件包、webpack 或 Babel。

{

"name": "test",

"version": "1.0.0",

"description": "TensorFlow.js test",

"main": "index.js",

"repository": "index.html",

"license": "MIT",

"engines": {

"node": ">=7.9.0"

},

"dependencies": {

"@tensorflow/tfjs": "2.0.0"

},

"scripts": {

"watch": "cross-env NODE\_ENV=development parcel index.html --no-hmr --open",

"build": "cross-env NODE\_ENV=production parcel build index.html --no-minify --public-url ./",

"link-local": "yalc link"

},

"devDependencies": {

"@babel/core": "^7.0.0-0",

"@babel/plugin-transform-runtime": "^7.1.0",

"babel-preset-env": "~1.6.1",

"clang-format": "~1.2.2",

"cross-env": "^5.1.6",

"parcel-bundler": "~1.12.5",

"yalc": "~1.0.0-pre.22"

}

}

1. 安装依赖

运行yarn安装依赖。

$  **yarn**

yarn install v1.22.5

info No lockfile found.

[1/5] Validating package.json...

[2/5] Resolving packages...

warning babel-preset-env > browserslist@2.11.3: Browserslist 2 could fail on reading Browserslist >3.0 config used in other tools.

[3/5] Fetching packages...

info fsevents@1.2.13: The platform "win32" is incompatible with this module.

info "fsevents@1.2.13" is an optional dependency and failed compatibility check. Excluding it from installation.

[4/5] Linking dependencies...

warning "@tensorflow/tfjs > @tensorflow/tfjs-data@2.0.0" has unmet peer dependency "seedrandom@~2.4.3".

[5/5] Building fresh packages...

success Saved lockfile.

Done in 194.90s.

1. 运行查看结果

然后执行yarn watch来启动开发服务器。

$  **yarn watch**

yarn run v1.22.5

$ cross-env NODE\_ENV=development parcel index.html --no-hmr --open

Server running at http://localhost:1234

√ Built in 15.67s.

接着在浏览器中打开该网址，查看结果。

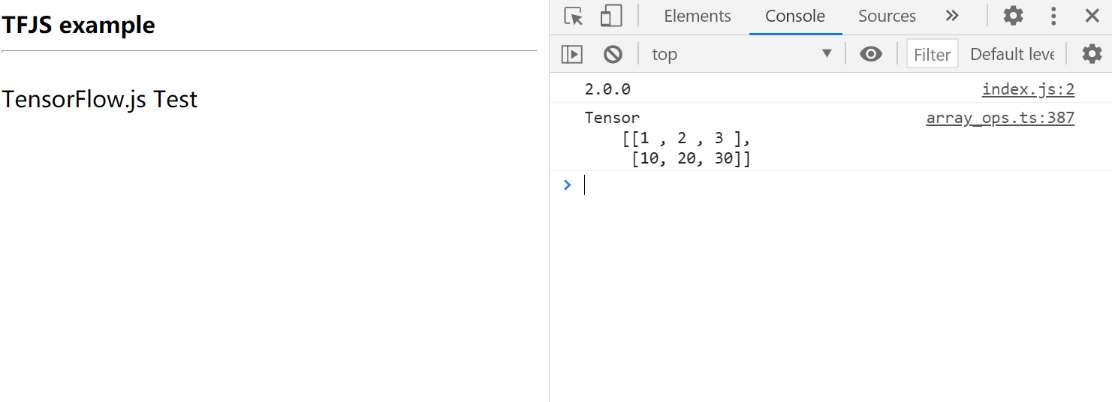


图7-3 运行结果

## 7.2 任务1：预测汽车油耗效率

这个项目是简单的线性回归的实验，用来预测汽车的油耗效率MPG，将使用一个小数据集和一个简单的模型，帮助大家熟悉使用TensorFlow.js进行训练模型的基本流程与概念和语法。

首先创建一个使用TensorFlow.js在浏览器中训练模型的网页，然后给定汽车的功率（Horsepower），使用模型预测汽车油耗（MPG），具体流程如下：

* 加载数据并准备进行训练；
* 定义模型结构；
* 训练模型，并监视其性能；
* 评估模型。

### 7.2.1 创建主页并加载数据

建立一个 HTML 文件，在头信息中，通过将 NPM 模块转换为在线可以引用的免费服务 cdn.jsdelivr.net，来加载 @tensorflow/tfjs 和 @tensorflow/tfjs-vis 两个 TFJS 模块。tfjs-vis是TensorFlow.js进行浏览器可视化的一组实用工具库。HTML文件名为index.html，代码如下：

<!DOCTYPE html>

<html>

<head>

<title>TensorFlow.js Tutorial</title>

<!-- Import TensorFlow.js -->

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@2.0.0/dist/tf.min.js"></script>

<!-- Import tfjs-vis -->

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@1.0.2/dist/tfjs-vis.umd.min.js"></script>

<!-- Import the main script file -->

<script src="index.js"></script>

</head>

<body>

</body>

</html>

在与上面的HTML文件相同的文件夹中，创建一个名为index.js的文件，并将以下代码放入其中。

async function getData() {

const carsDataResponse = await fetch('https://storage.googleapis.com/tfjs-tutorials/carsData.json');

const carsData = await carsDataResponse.json();

const cleaned = carsData.map(car => ({

mpg: car.Miles\_per\_Gallon,

horsepower: car.Horsepower,

}))

.filter(car => (car.mpg != null && car.horsepower != null));

return cleaned;

}

async function run() {

// Load and plot the original input data that we are going to train on.

const data = await getData();

const values = data.map(d => ({

x: d.horsepower,

y: d.mpg,

}));

tfvis.render.scatterplot(

{name: 'Horsepower v MPG'},

{values},

{

xLabel: 'Horsepower',

yLabel: 'MPG',

height: 300

}

);

// More code will be added below

}

document.addEventListener('DOMContentLoaded', run);

首先需要加载数据、并对数据格式化（进行预处理）和可视化要用于训练模型的数据。从服务端获取 JSON 文件中加载数据集。数据集中包含了关于每辆给定汽车的许多特性，例如MPG(油耗)、Cylinders(气缸数量)、Displacement(排气量)、Weight(车重)。然后提取有关Horsepower和MPG的数据作为训练数据。

{

"Name": "chevrolet chevelle malibu",

"Miles\_per\_Gallon": 18,

"Cylinders": 8,

"Displacement": 307,

"Horsepower": 130,

"Weight\_in\_lbs": 3504,

"Acceleration": 12,

"Year": "1970-01-01",

"Origin": "USA"

},

run()是项目的主函数，后面的功能将陆续添加在这里。通过 map 来获得将要进行训练的特性，通过 tfvis 将数据绘制成散点图。

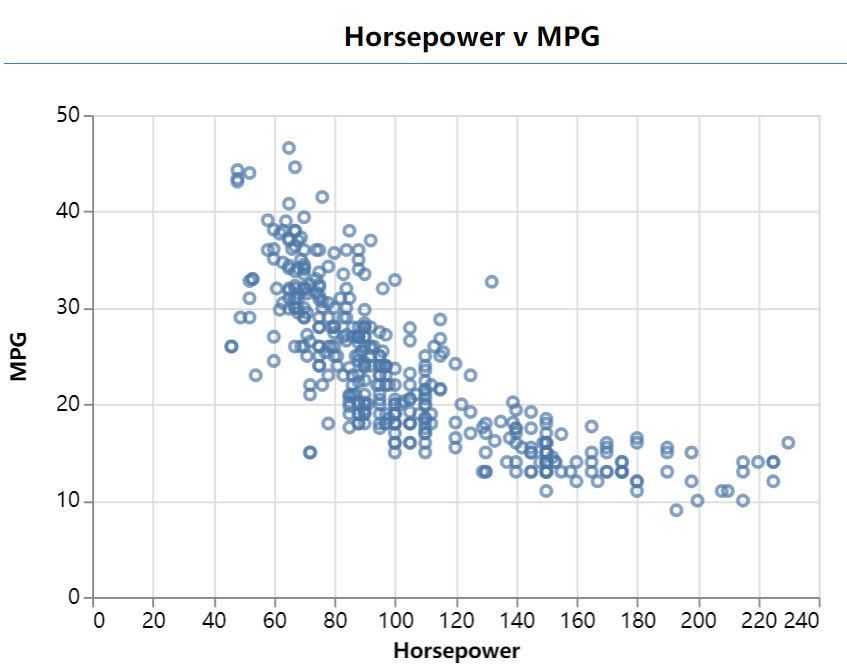


图7-4 tfvis绘制散点图

为了避免阻塞整个程序的执行，可能耗时的函数应当尽量使用异步方式，也就是function getData()关键字之前的async。

### 7.2.2 定义模型结构

TensorFlow.js有两种创建模型的方式，一种是用的tf.sequential()，另外一种是tf.model()，两者的差别是tf.sequential()是一个线性堆叠layers的模型，而tf.model()定义的神经元网络层与层之间的关系较为随意。TensorFlow.js完整模仿了Keras的模型定义方式，下面代码是模型定义：

function createModel() {

// Create a sequential model

const model = tf.sequential();

// Add a single input layer

model.add(tf.layers.dense({inputShape: [1], units: 1, useBias: true}));

// Add an output layer

model.add(tf.layers.dense({units: 1, useBias: true}));

return model;

}

因为数据比较简单，所以使用了两层全连接网络。只有功耗这一个输入值，所以输入的张量形状是[1]，输出的神经元数量也为1（油耗）。useBias是神经元权重计算中的偏置量，可以不用显式设为true。

首先实例化一个tf. sequential对象，然后调用add方法为网络添加一个输入层，该输入层将自动连接到具有一个隐藏单元的dense层。inputShape形状为[1]，因为有1个数字作为输入（即给定汽车的功率）。最后再添加一个输出dense层，units为1（即输出为一个数字，油耗）。

模型定义完成后，需要在run()函数中添加调用，并调用可视化工具提供的modelSummary方法，将模型显示在浏览器中。

// Create the model

const model = createModel();

tfvis.show.modelSummary({name: 'Model Summary'}, model);

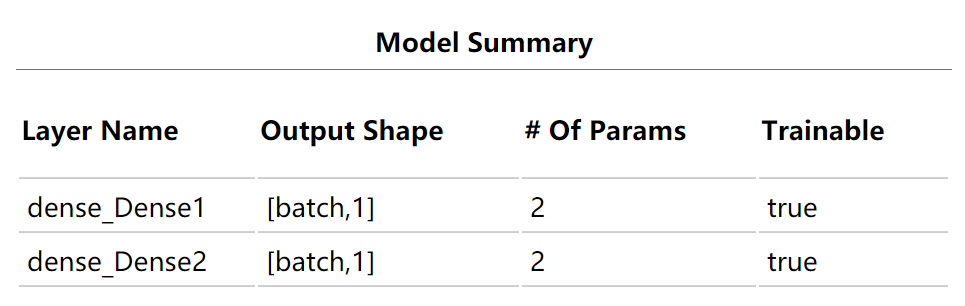


图7-5 模型各层的参数状况

### 7.2.3 数据预处理

在数据载入的时候需要进行预处理的工作，需要做数据规范化，并把转换为TensorFlow处理起来更高效的张量类型。

JavaScript语言在大规模数据的处理上不如Python的高效，其中最突出的问题是内存的回收。用户对于浏览器的内存占用本身也是非常敏感的。TensorFlow.js为了解决这个问题，专门提供了tf.tidy()函数。使用方法是把大规模的内存操作，放置在这个函数的回调中执行。函数调用完成后，tf.tidy()得到控制权，进行内存的清理工作，防止内存泄露。

function convertToTensor(data) {

// Wrapping these calculations in a tidy will dispose any

// intermediate tensors.

return tf.tidy(() => {

// Step 1. Shuffle the data

tf.util.shuffle(data);

// Step 2. Convert data to Tensor

const inputs = data.map(d => d.horsepower)

const labels = data.map(d => d.mpg);

const inputTensor = tf.tensor2d(inputs, [inputs.length, 1]);

const labelTensor = tf.tensor2d(labels, [labels.length, 1]);

//Step 3. Normalize the data to the range 0 - 1 using min-max scaling

const inputMax = inputTensor.max();

const inputMin = inputTensor.min();

const labelMax = labelTensor.max();

const labelMin = labelTensor.min();

const normalizedInputs = inputTensor.sub(inputMin).div(inputMax.sub(inputMin));

const normalizedLabels = labelTensor.sub(labelMin).div(labelMax.sub(labelMin));

return {

inputs: normalizedInputs,

labels: normalizedLabels,

// Return the min/max bounds so we can use them later.

inputMax,

inputMin,

labelMax,

labelMin,

}

});

}

convertToTensor函数首先tf.util.shuffle方法打乱数据集中数据顺序，创建特征向量inputTensor与标签向量labelTensor，将原始数据转变为tensorflow可读的张量格式。最后对输入和输出的数据做归一化操作（让输入输出映射到0-1之间），保证后期更有效地训练。

### 7.2.4 训练与测试模型

训练和测试的代码与TensorFlow非常类似，一样使用model.fit()做训练，以及model.predict()做预测。模型训练的过程和结果，可以使用TensorFLow-vis图表工具可视化出来，显示在浏览器中。其中训练部分是使用回调函数，目的是能够动态的显示训练的过程。

async function trainModel(model, inputs, labels) {

// Prepare the model for training.

model.compile({

optimizer: tf.train.adam(),

loss: tf.losses.meanSquaredError,

metrics: ['mse'],

});

const batchSize = 32;

const epochs = 50;

return await model.fit(inputs, labels, {

batchSize,

epochs,

shuffle: true,

callbacks: tfvis.show.fitCallbacks(

{ name: 'Training Performance' },

['loss', 'mse'],

{ height: 200, callbacks: ['onEpochEnd'] }

)

});

}

模型优化算法使用adam，使用均方差作为判断训练结果的参数。训练模型采用分批采样训练，一次采样32条（batchSize）训练数据，遍历所有样本50次（epochs），shuffle设置为true，表示打乱数据集。最后设置了callback，可以在每一个训练周期显示训练情况。

在run()函数添加训练代码，运行后结果如图：

// Convert the data to a form we can use for training.

const tensorData = convertToTensor(data);

const {inputs, labels} = tensorData;

// Train the model

await trainModel(model, inputs, labels);

console.log('Done Training');

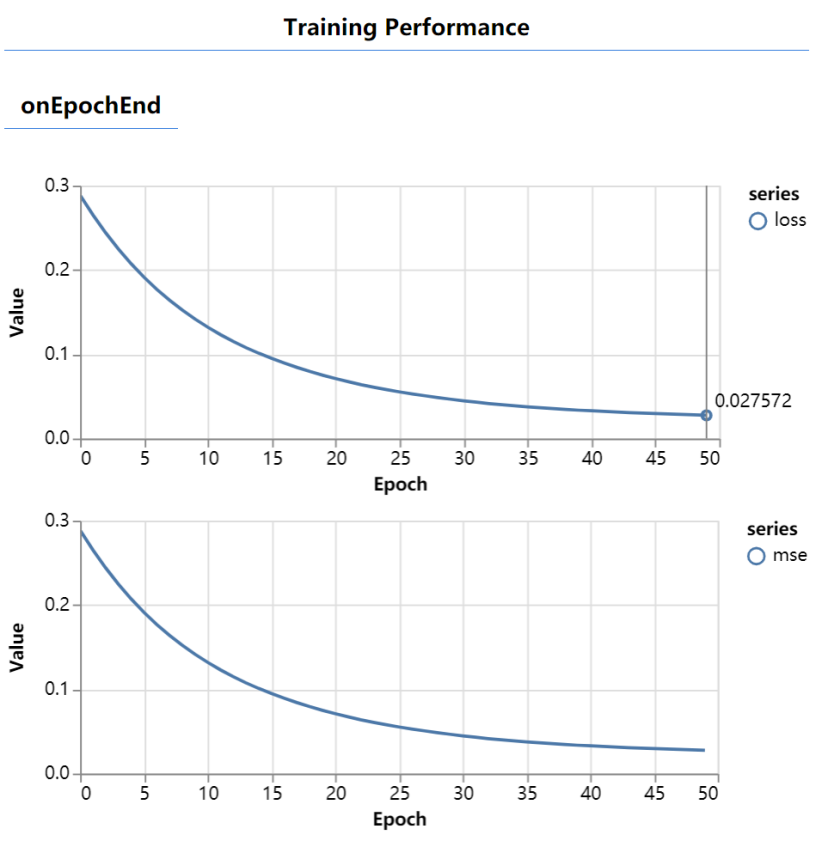


图7-6 每一个训练周期显示训练情况

训练完成后，调用testModel()函数来预测油耗，代码如下:

function testModel(model, inputData, normalizationData) {

const {inputMax, inputMin, labelMin, labelMax} = normalizationData;

// Generate predictions for a uniform range of numbers between 0 and 1;

// We un-normalize the data by doing the inverse of the min-max scaling

// that we did earlier.

const [xs, preds] = tf.tidy(() => {

const xs = tf.linspace(0, 1, 100);

const preds = model.predict(xs.reshape([100, 1]));

const unNormXs = xs

.mul(inputMax.sub(inputMin))

.add(inputMin);

const unNormPreds = preds

.mul(labelMax.sub(labelMin))

.add(labelMin);

// Un-normalize the data

return [unNormXs.dataSync(), unNormPreds.dataSync()];

});

const predictedPoints = Array.from(xs).map((val, i) => {

return {x: val, y: preds[i]}

});

const originalPoints = inputData.map(d => ({

x: d.horsepower, y: d.mpg,

}));

tfvis.render.scatterplot(

{name: 'Model Predictions vs Original Data'},

{values: [originalPoints, predictedPoints], series: ['original', 'predicted']},

{

xLabel: 'Horsepower',

yLabel: 'MPG',

height: 300

}

);

}

调用tf.linspace()方法创建0~1之间平均分配的100个值，然后调用predict()方法预测。

在run()函数添加调用代码，运行后结果如图7-7：

testModel(model, data, tensorData);

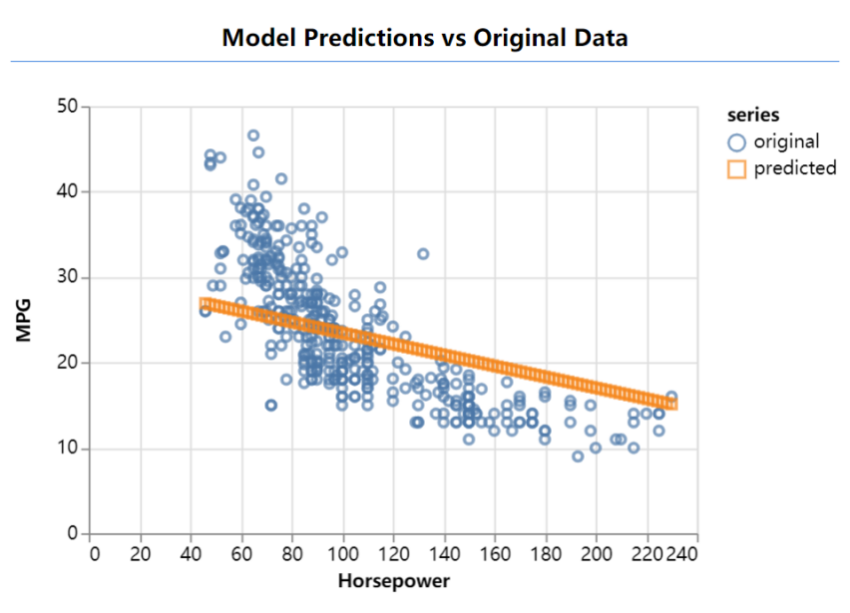


图7-7 预测结果

## 7.3 任务2：手写数字识别

下面将使用 CNN 构建一个 Tensorflow.js 模型来识别手写数字。首先训练分类器，让它查看数千个图像以及其标签，然后将使用模型从未见过的测试数据来评估分类器的准确性。

### 7.3.1 从GitHub获取源码并运行

Tensorflow.js在其示例官网https://github.com/tensorflow/tfjs-examples中已经公开了诸多的例子，mnist项目目录如图7-8：

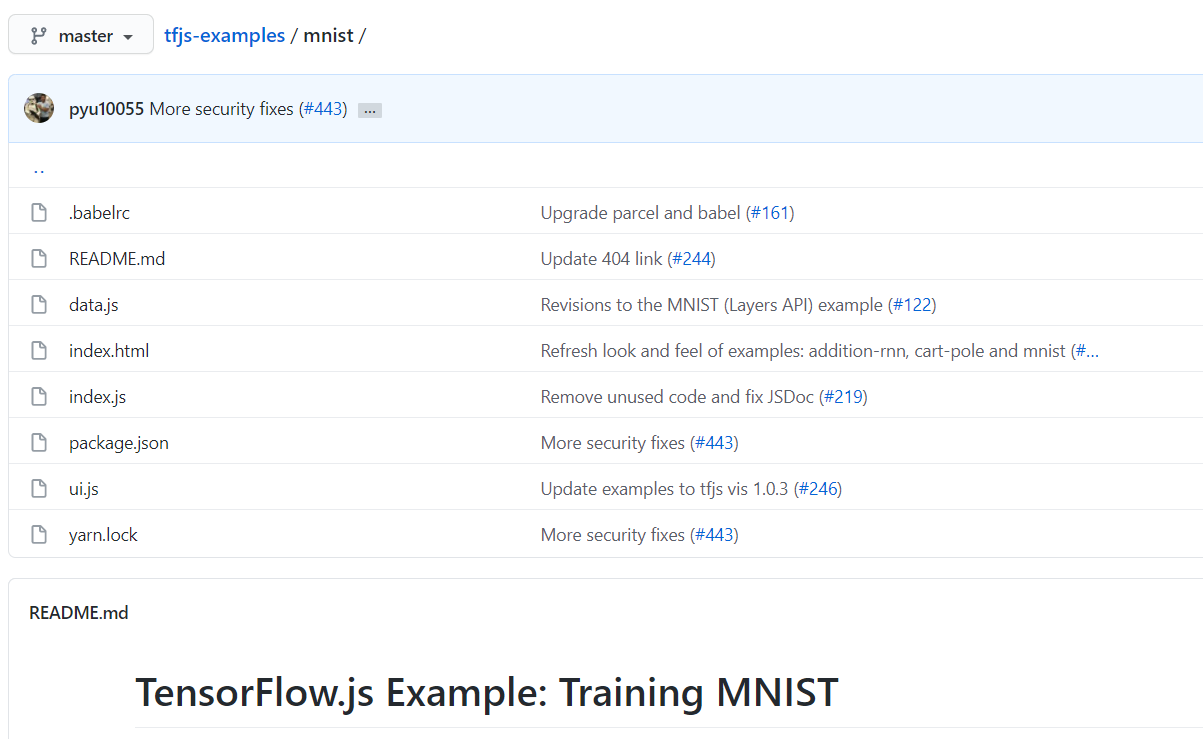


图7-8 mnist项目目录

从GitHub克隆项目代码，以获取项目所需的HTML，JS文件和配置文件的副本。

$ git clone https://github.com/tensorflow/tfjs-examples.git

$ cd tfjs-examples/ mnist

如图7-8项目包含三类文件。第一是HTML文件，文件主要包含页面的基本结构，命名为index.html，它包含一些div标签，一些UI元素以及一个源标签，以JavaScript代码插入例如index.js 。

JS代码通常分为几个文件，以提高良好的可读性。用于更新可视化元素的代码位于ui.js中，而用于下载数据的代码位于data.js中。

第三个重要文件类型是软件包配置文件package.json，这是npm软件包管理器。如果以前从未使用过npm 或yarn ，建议您在https://docs.npmjs.com/getting-started/what-is-npm上浏览一下npm“入门”文档，并逐渐熟悉以便能够构建并运行示例代码。下面将使用yarn作为包管理器。

在mnist代码目录中包含一下文件:

1. index.html ：HTML根文件，提供DOM根并调用JS脚本。
2. index.js ：根文件，用于加载数据，定义模型，训练循环并指定UI元素。
3. data.js ：实现下载和访问mnist数据集。
4. ui.js ：用于更新可视化元素。
5. package.json ：软件包配置文件，描述了构建和运行此示例所需的依赖项。

使用yarn命令构建，运行mnist代码：

$ yarn

$ yarn watch

如果您是Web 开发新手，或者从未听说过诸如 Webpack 或 Parcel 的工具，建议使用脚本代码。如果您比较有经验或想编写更大的程序，则可能值得使用构建工具进行探索。本项目将使用脚本代码实现相关功能。

### 7.3.2 创建相关文件

在同一目录下创建index.html文件，index.js文件，拷贝tfjs-examples/ mnist目录下data.js文件。index.html文件代码如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>TensorFlow.js Tutorial</title>

<!-- Import TensorFlow.js -->

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@1.0.0/dist/tf.min.js"></script>

<!-- Import tfjs-vis -->

<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@1.0.2/dist/tfjs-vis.umd.min.js"></script>

<!-- Import the data file -->

<script src="data.js" type="module"></script>

<!-- Import the main script file -->

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

</head>

<body>

</body>

</html>

data.js实现了预处理数据，其功能与之前讲解的Python代码类似。其中包含了MnistData类，从MNIST数据集中随机批量提取MNIST图像。

MnistData将整个数据集分为训练数据和测试数据。当训练模型时，分类器将使用训练集进行训练。在评估模型时，使用测试集中的数据，以检查模型对新数据的泛化情况。

MnistData有两个public方法：

nextTrainBatch（batchSize）：从训练集中返回一批随机图像及其标签。

nextTestBatch（batchSize）：从测试集中返回一批图像及其标签。

在训练MNIST分类器时，为了模型的预测将不受图像的顺序的影响，随机打乱数据集是非常重要的。例如，如果首先将所有1位数字提供给模型，那么在此阶段的训练中，模型可能会学会简单地预测1。如果我们只给模型提供2，它可能会简单地转换到仅预测2，并且从不预测1。

在index.js文件中添加一下代码加载数据集并显示20张图片，如图7-9，代码如下：

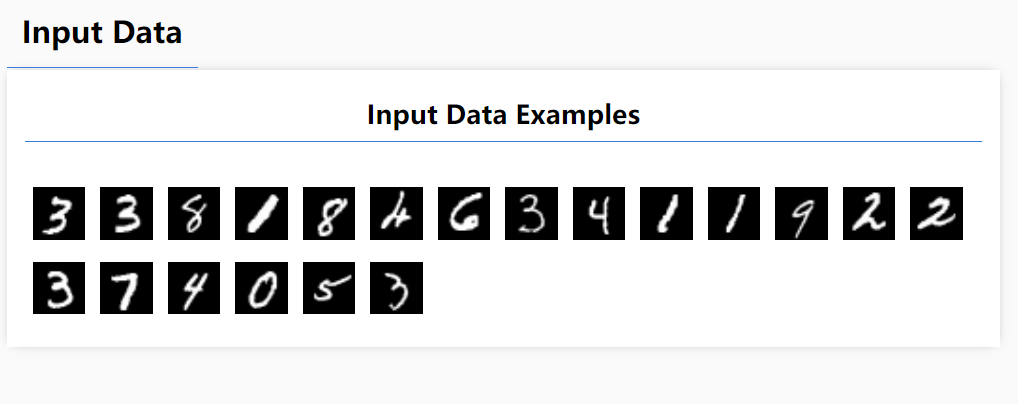


图7-9 显示20张图片

import {MnistData} from './data.js';

async function showExamples(data) {

// Create a container in the visor

const surface =

tfvis.visor().surface({ name: 'Input Data Examples', tab: 'Input Data'});

// Get the examples

const examples = data.nextTestBatch(20);

const numExamples = examples.xs.shape[0];

// Create a canvas element to render each example

for (let i = 0; i < numExamples; i++) {

const imageTensor = tf.tidy(() => {

// Reshape the image to 28x28 px

return examples.xs

.slice([i, 0], [1, examples.xs.shape[1]])

.reshape([28, 28, 1]);

});

const canvas = document.createElement('canvas');

canvas.width = 28;

canvas.height = 28;

canvas.style = 'margin: 4px;';

await tf.browser.toPixels(imageTensor, canvas);

surface.drawArea.appendChild(canvas);

imageTensor.dispose();

}

}

async function run() {

const data = new MnistData();

await data.load();

await showExamples(data);

}

document.addEventListener('DOMContentLoaded', run);

通过搭建本地一个服务器去进行资源的问题来解决跨域问题，例如使用Web Server for Chrome，如图7-10。否则会报如下错误：

Access to script at 'file:///E:/mnist/data.js' from origin 'null' has been blocked by CORS policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, chrome-untrusted, https.

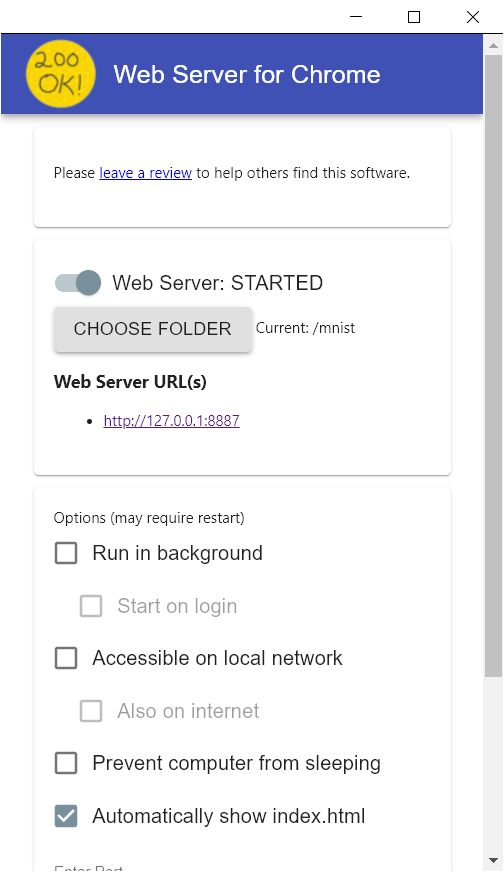


图7-10 Web Server for Chrome

### 7.3.3 定义模型结构

我们已经知道MNIST数据集的神经网络采用什么样的输入，以及它应该生成什么样的输出。神经网络的输入张量形状为[null，28，28，1]，输出张量形状为[null，10]，其中第二维度对应于十个可能的数字。下面将定义一个卷积图像分类模型，将使用Sequential模型，其中张量将连续地从一层传递到下一层。

在index.js文件中添加以下代码：

function getModel() {

const model = tf.sequential();

const IMAGE\_WIDTH = 28;

const IMAGE\_HEIGHT = 28;

const IMAGE\_CHANNELS = 1;

// In the first layer of our convolutional neural network we have

// to specify the input shape. Then we specify some parameters for

// the convolution operation that takes place in this layer.

model.add(tf.layers.conv2d({

inputShape: [IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS],

kernelSize: 5,

filters: 8,

strides: 1,

activation: 'relu',

kernelInitializer: 'varianceScaling'

}));

// The MaxPooling layer acts as a sort of downsampling using max values

// in a region instead of averaging.

model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2, 2]}));

// Repeat another conv2d + maxPooling stack.

// Note that we have more filters in the convolution.

model.add(tf.layers.conv2d({

kernelSize: 5,

filters: 16,

strides: 1,

activation: 'relu',

kernelInitializer: 'varianceScaling'

}));

model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2, 2]}));

// Now we flatten the output from the 2D filters into a 1D vector to prepare

// it for input into our last layer. This is common practice when feeding

// higher dimensional data to a final classification output layer.

model.add(tf.layers.flatten());

// Our last layer is a dense layer which has 10 output units, one for each

// output class (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

const NUM\_OUTPUT\_CLASSES = 10;

model.add(tf.layers.dense({

units: NUM\_OUTPUT\_CLASSES,

kernelInitializer: 'varianceScaling',

activation: 'softmax'

}));

// Choose an optimizer, loss function and accuracy metric,

// then compile and return the model

const optimizer = tf.train.adam();

model.compile({

optimizer: optimizer,

loss: 'categoricalCrossentropy',

metrics: ['accuracy'],

});

return model;

}

首先用tf.sequential实例化Sequential模型，然后为它添加层。

1. 添加第一层

第一层是一个二维卷积层。 卷积在图像上滑动滤波器窗口以学习空间不变的变换（即，图像不同部分的图案或目标将以相同方式处理）。

使用tf.layers.conv2d来创建二维卷积层，它接受一个定义层结构的配置对象作为输入：

model.add(tf.layers.conv2d({

inputShape: [28, 28, 1],

kernelSize: 5,

filters: 8,

strides: 1,

activation: 'relu',

kernelInitializer: 'VarianceScaling'

}));

配置对象中的参数说明如下：

inputShape：将流入模型第一层的数据的形状。MNIST样本是28x28像素的黑白图像。图像数据的规范格式是[row，column，depth]，所以形状是[28,28,1]。

kernelSize：应用于输入数据的滑动卷积滤波器窗口的大小。设置kernelSize为5，表示一个5x5的正方形卷积窗口。

Filters：应用于输入数据，大小为kernelSize的滤波器窗口的数量。

Strides：滑动窗口的步长，即每次在图像上移动时，滤波器将移动多少个像素。指定步幅为1，这意味着过滤器将以1像素为单位滑过图像。

Activation：卷积完成后应用于数据的激活函数，设置为ReLU函数。

kernelInitializer：用于随机初始化模型权重的方法。

1. 添加第二层

为模型添加第二层：最大池化层，使用 tf.layers.maxPooling2d创建它，该层将通过计算每个滑动窗口的最大值来缩减卷积结果的大小：

model.add(tf.layers.maxPooling2d({

poolSize: [2, 2],

strides: [2, 2]

}));

参数说明如下：

poolSize：应用于输入数据的滑动窗口大小。设置poolSize为[2,2]，池化层将对输入数据应用2x2窗口。

Stride：滑动窗口的步长。

由于poolSize和strides都是2×2，所以池窗口将完全不重叠。这意味着池化层会将前一层的激活图的大小减半。

1. 添加剩余层

重复使用层结构是神经网络中的常见模式。添加第二个卷积层到模型，并在其后添加池化层。在第二个卷积层中，将滤波器数量从8增加到16。没有指定inputShape，因为它可以从前一层的输出形状中推断出来。

接下来添加一个 flatten层，将前一层的输出平铺到一个向量中。

最后添加一个 dense层，它将执行最终的分类。 在dense层前先对卷积+池化层的输出执行flatten也是神经网络中的另一种常见模式。

1. 定义优化器

将使用自适应矩估计（Adam）优化器，Adam 算法是一种对随机目标函数执行一阶梯度优化的算法，该算法基于适应性低阶矩估计。

1. 编译模型

编译模型时需要传入一个由优化器，损失函数和一系列评估指标组成的配置对象。损失函数使用常用于优化分类任务的交叉熵（ categoricalCrossentropy）。 categoricalCrossentropy 度量模型的最后一层产生的概率分布与标签给出的概率分布之间的误差，这个分布在正确的类标签中为1（100％）。

对于评估指标，将使用准确度，该准确度衡量所有预测中正确预测的百分比。

模型各层的参数状况如图7-11：

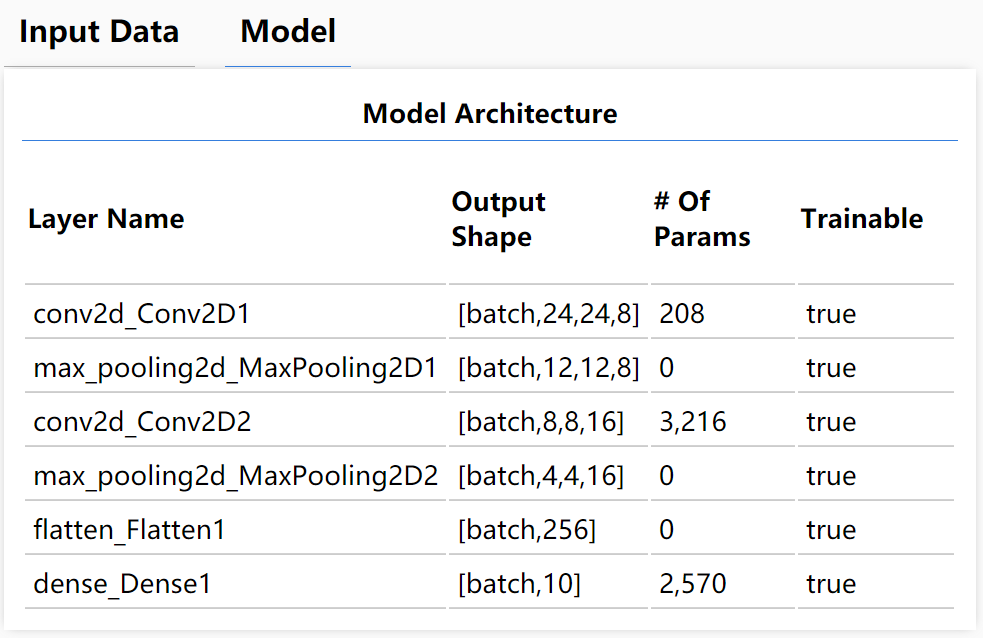


图7-11 模型各层的参数

### 7.3.4 训练模型

现在已经成功地定义了模型的拓扑结构，下一步就是训练并评估训练的结果。在index.js文件中添加以下代码：

async function train(model, data) {

const metrics = ['loss', 'val\_loss', 'acc', 'val\_acc'];

const container = {

name: 'Model Training', tab: 'Model', styles: { height: '1000px' }

};

const fitCallbacks = tfvis.show.fitCallbacks(container, metrics);

const BATCH\_SIZE = 512;

const TRAIN\_DATA\_SIZE = 5500;

const TEST\_DATA\_SIZE = 1000;

const [trainXs, trainYs] = tf.tidy(() => {

const d = data.nextTrainBatch(TRAIN\_DATA\_SIZE);

return [

d.xs.reshape([TRAIN\_DATA\_SIZE, 28, 28, 1]),

d.labels

];

});

const [testXs, testYs] = tf.tidy(() => {

const d = data.nextTestBatch(TEST\_DATA\_SIZE);

return [

d.xs.reshape([TEST\_DATA\_SIZE, 28, 28, 1]),

d.labels

];

});

return model.fit(trainXs, trainYs, {

batchSize: BATCH\_SIZE,

validationData: [testXs, testYs],

epochs: 10,

shuffle: true,

callbacks: fitCallbacks

});

}

开始训练前先定义需要监视的指标，['loss', 'val\_loss', 'acc', 'val\_acc']分别表示训练集的准确度和损失值、以及验证集的准确度和损失值。 验证集的最大作用是方便我们了解模型效率、调试超参数。

trainXs是训练集，将用这个训练集训练模型，testXs是验证集，在每个时期结束时对模型进行测试，在训练过程中，验证集中的数据永远不能用于训练。

数据集需要调整为模型期望的形状，调整后的形状为[num\_examples，image\_width，image\_height，channels]，然后再将它们输入模型。 对于每个数据集，都有输入（Xs）和标签（Ys）。

model.fit调用指定BATCH\_SIZE 设置为512，每次批量处理512个图像，MNIST 数据集中单个图像的维度为[28,28,1]，意味数据的实际形状是[512,28,28,1]。

一般而言，使用较大的批次与较小的批次相比好处是，它对模型的权重产生了更一致且变化较小的渐变更新。在优化过程中，只能在对多个样本中的梯度进行平均后更新内部参数。这有助于避免因错误的样本（例如错误标记的数字）而改向错误的方向。但批次越大，训练期间就需要更多的内存。在给定相同数量的训练数据的情况下，较大的批次大小会导致每个时期的梯度更新数量较少。如果使用较大的批次，请确保相应地增加epochs，以免在训练过程中无意中减少了权重更新的次数。

model.fit 设置验证集validationData为[testXs, testYs]。在训练期间需要验证损失和准确性，了解模型是否以及何时过度拟合。

model.fit是异步函数，因此如果后续操作依赖于fit调用的完成，则需要对其使用await。需要在模式训练后使用测试数据集对模型执行评估。

在index.js文件run函数中添加以下代码：

const model = getModel();

tfvis.show.modelSummary({name: 'Model Architecture', tab: 'Model'}, model);

await train(model, data);

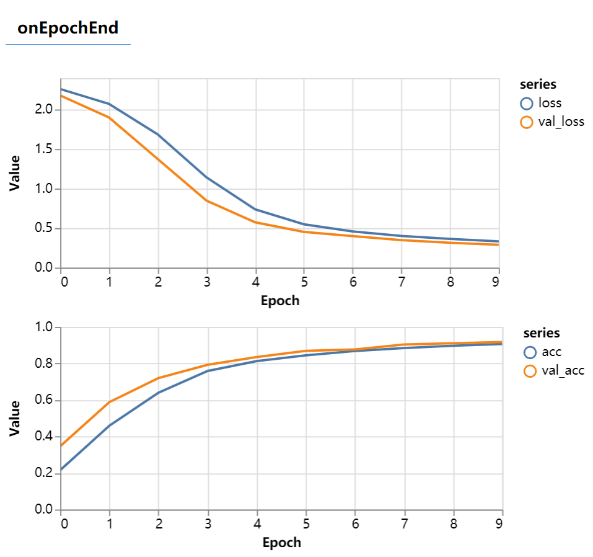


图7-12 训练曲线

图7-12 MNIST模型训练曲线，执行10个周期，每个周期由大约110批次组成。训练集和验证集的值由不同的颜色符号显示。经过10个阶段的训练，最终得到的评价准确率为95.0%。

### 7.3.4 使用模型进行评估与预测

现在已经有一个训练有素的模型。如何评估它的性能以及使用它来对手写数字的图像进行真正的分类？

评估性能代码如下, 请在index.js文件中添加以下代码:

const classNames = ['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine'];

function doPrediction(model, data, testDataSize = 500) {

const IMAGE\_WIDTH = 28;

const IMAGE\_HEIGHT = 28;

const testData = data.nextTestBatch(testDataSize);

const testxs = testData.xs.reshape([testDataSize, IMAGE\_WIDTH, IMAGE\_HEIGHT, 1]);

const labels = testData.labels.argMax(-1);

const preds = model.predict(testxs).argMax(-1);

testxs.dispose();

return [preds, labels];

}

async function showAccuracy(model, data) {

const [preds, labels] = doPrediction(model, data);

const classAccuracy = await tfvis.metrics.perClassAccuracy(labels, preds);

const container = {name: 'Accuracy', tab: 'Evaluation'};

tfvis.show.perClassAccuracy(container, classAccuracy, classNames);

labels.dispose();

}

async function showConfusion(model, data) {

const [preds, labels] = doPrediction(model, data);

const confusionMatrix = await tfvis.metrics.confusionMatrix(labels, preds);

const container = {name: 'Confusion Matrix', tab: 'Evaluation'};

tfvis.render.confusionMatrix(container, {values: confusionMatrix, tickLabels: classNames});

labels.dispose();

}

准备500张图片作为测试集，调用model.predict预测结果，可以稍后增加测试集以在更大的测试集上进行测试。

模型为每个类输出一个概率，argMax函数提供了最高概率类的索引，找出最大的概率，并指定使用它作为预测。

run函数中添加以下代码开始预测。

await showAccuracy(model, data);

await showConfusion(model, data);

通过预测结果和标签，可以计算每个类的准确度，结果如图。

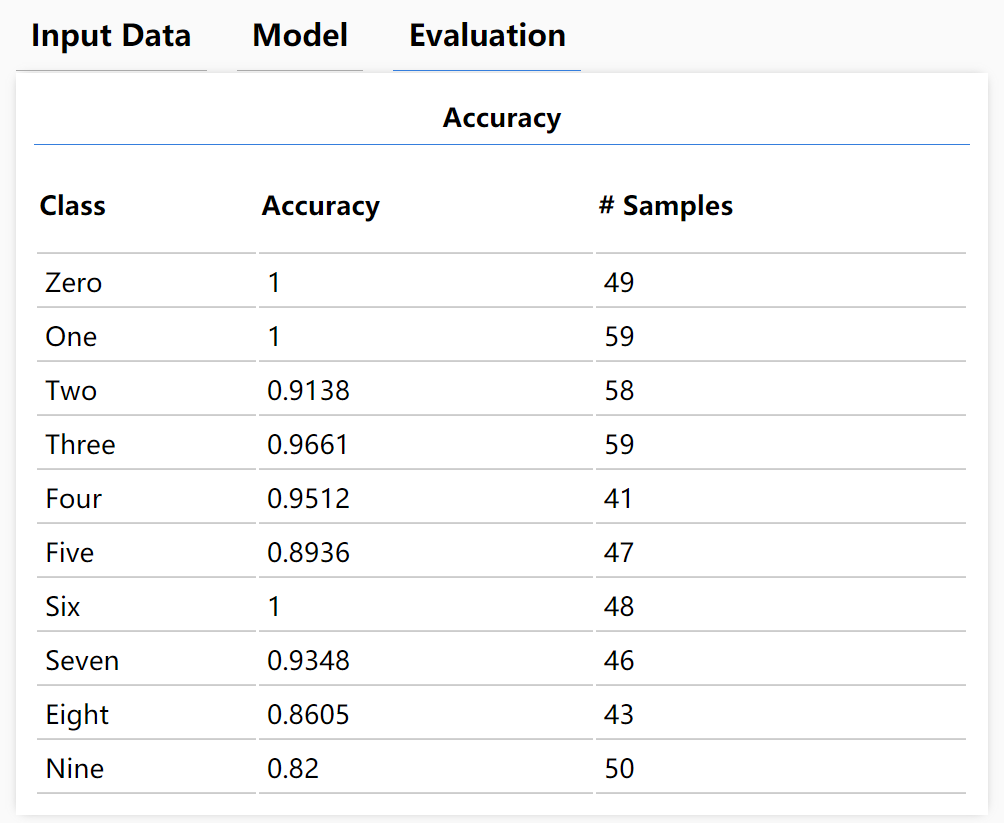


图7-12 每个类别的准确度

使用tfvis.metrics.confusionMatrix绘制混淆矩阵如图7-13所示，混淆矩阵又称为可能性表格或是错误矩阵。它是一种特定的矩阵用来呈现算法性能的可视化效果，通常用于监督学习，非监督学习通常用匹配矩阵（matching matrix）。其每一列代表预测值，每一行代表的是实际的类别。这个名字来源于它可以非常容易的表明多个类别是否有混淆。

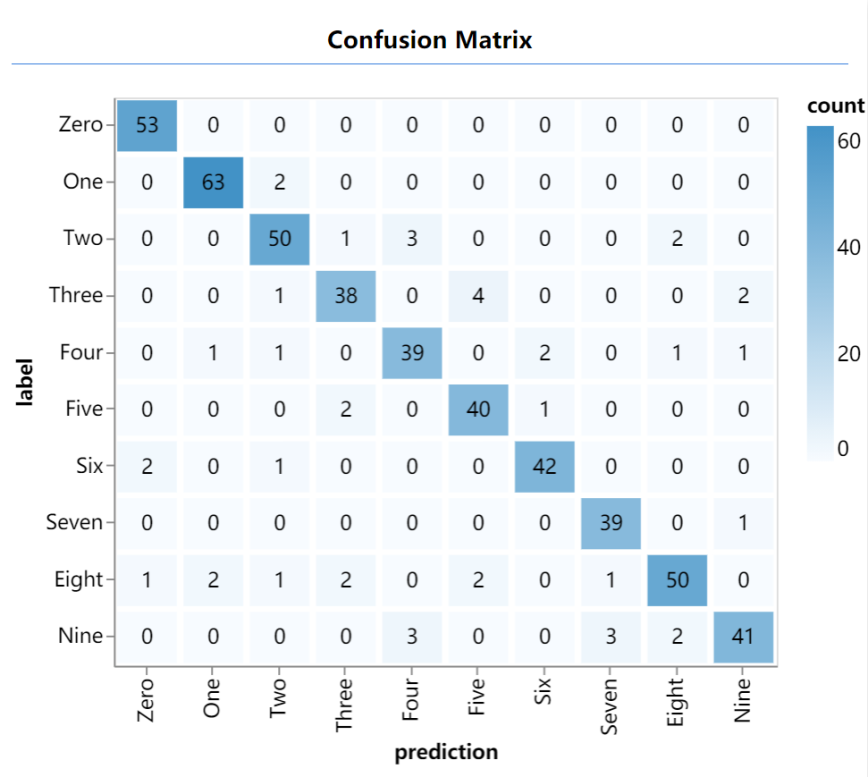


图7-13 混淆矩阵

## 习题与练习

TensorFlow官网提供了很多在您项目中开箱即用的 TensorFlow.js 预训练模型，例如图像分类、对象检测、姿势估计、文本恶意检测等。同时提供了很多演示项目，例如会学习的机器、Node.js高音预测等，请参见 <https://tensorflow.google.cn/js/demos?hl=zh_cn>。

“剪刀石头布”是大家小时候经常玩的游戏，日常生活中做一些纠结的决策，有时候也常常使用这种规则得出最后的选择，人能很轻松地认知这些手势，“石头”呈握拳状，“布”掌心摊开，“剪刀”食指和中指分叉，如何让机器识别这些手势呢？

本项目任务要求使用TensorFlow.js实现根据摄像头采集的手势图像来确定它代表剪刀、石头、布中的哪一个。

Laurence Moroney提供了大量的优秀数据，其中也包括剪刀、石头、布手势的图像。数据集链接地址：

http://www.laurencemoroney.com/rock-paper-scissors-dataset/

这是图像分类任务，所需工作任务包括数据图像的采集、模型的训练、参数的调整，最终得到模型文件（如：VGG、ResNet等），并在网页端部署，最后使用网络摄像头检查自己做出的代表石头剪刀布的手势图像。