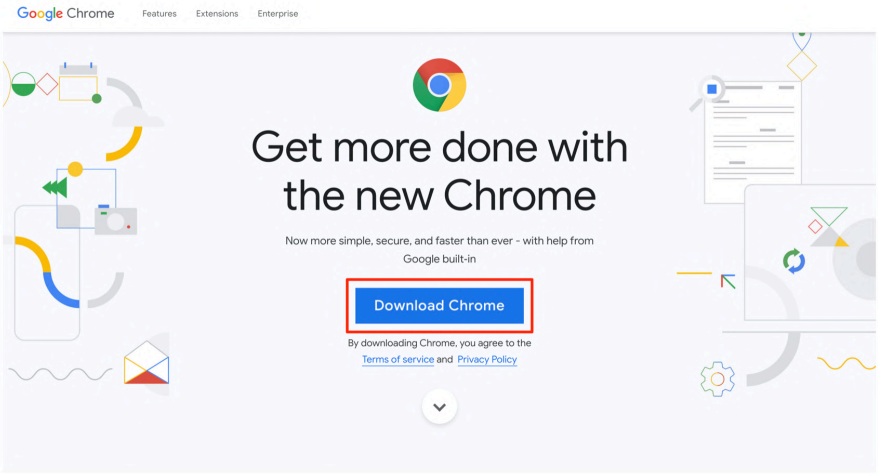
Getting Your System Ready

In this course you will be running all the examples and exercises locally on your machine. Consequently, you will need to have an internet browser, an HTML editor, and a web server installed on your machine. Throughout this course we are going to use Chrome as our internet browser, Brackets as our HTML editor and the Web Server for Chrome App as our web server. All of these are free and are available for various platforms including Windows, Mac OS, and Linux. Below, we will go over the installation process of each of these. Of course, you are welcome to use any software that you like, but we recommend that you install the above so that it is easier for you to follow along.

Chrome

To install the Chrome browser, simply visit the Chrome home page.



Click the blue "Download Chrome" button to download the installation file to your machine.

Install Chrome on Windows

1. If prompted, click **Run** or **Save**.
2. If you chose **Save**, double-click the downloaded file to start installing.
3. Start Chrome.

Install Chrome on Mac

1. Open the file called **googlechrome.dmg**.
2. In the window that opens, find Chrome .
3. Drag Chrome to the Applications folder.
4. Open Chrome.
5. Open Finder.
6. In the sidebar, to the right of Google Chrome, click Eject.

Install Chrome on Linux

Use the same software that installs programs on your computer to install Chrome. You'll be asked to enter the administrator account password.

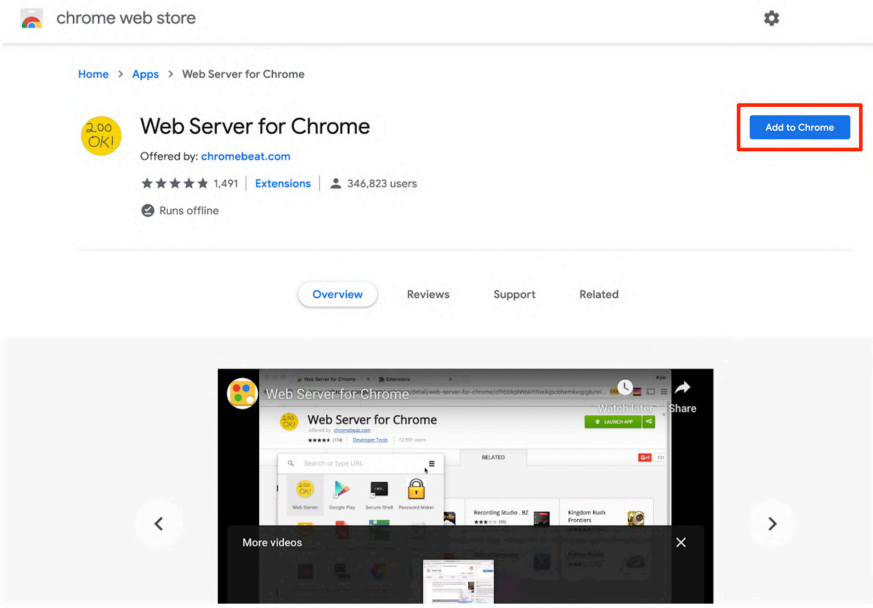
1. To open the package, click **OK**.
2. Click **Install Package**.

Google Chrome will be added to your software manager so it stays up-to-date.

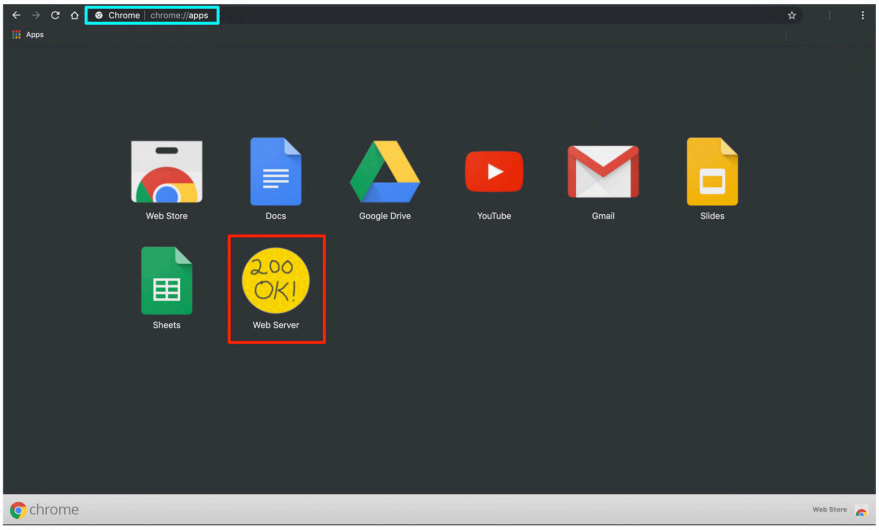
NOTE: It is very important that you use the latest version of Chrome in order for the exercises to run. So make sure you have the latest version of Chrome installed. The current latest version is: 78.0.3904.108.

Web Server for Chrome

To install the Web Server for Chrome App go to the Web Server for Chrome App page in the chrome web store.

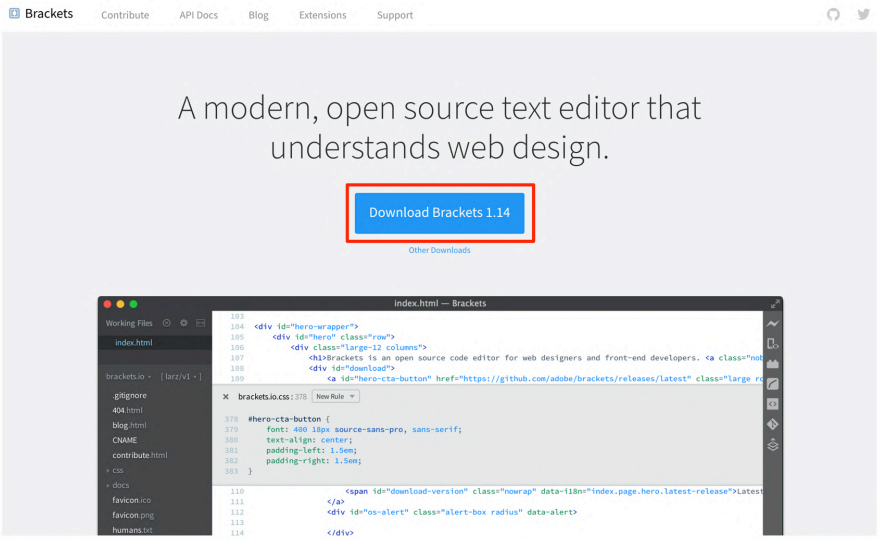


Click on the blue "Add to Chrome" button on the upper right-hand corner. The app will automatically be added to your Chrome Browser. You can find it under the chrome apps (chrome://apps/):



Brackets

To install the Brackets, simply visit the Brackets home page.



Click on the blue "Download Brackets" button to download the installation file to your machine.

Install Brackets on Windows

1. If prompted, click **Run** or **Save**.
2. If you chose **Save**, double-click the downloaded file to start installing.
3. Start Brackets.
4. When you open Brackets for the first time it will open a Windows Security Alter window. Click on "Allow Access" to run Brackets.

Install Brackets on Mac

1. Open the file called **Brackets.Release.1.14.0.dmg**. Note: The numbers at the end of the filename will change depending on the version that you are installing.
2. Open the downloaded file.
3. Drag Brackets to the Applications folder.
4. Open Brackets.
5. Open Finder.
6. In the sidebar, to the right of Brackets, click Eject.

Install Brackets on Linux

Use the same software that installs programs on your computer to install Brackets. You'll be asked to enter the administrator account password.

1. To open the package, click **OK**.
2. Click **Install Package**.

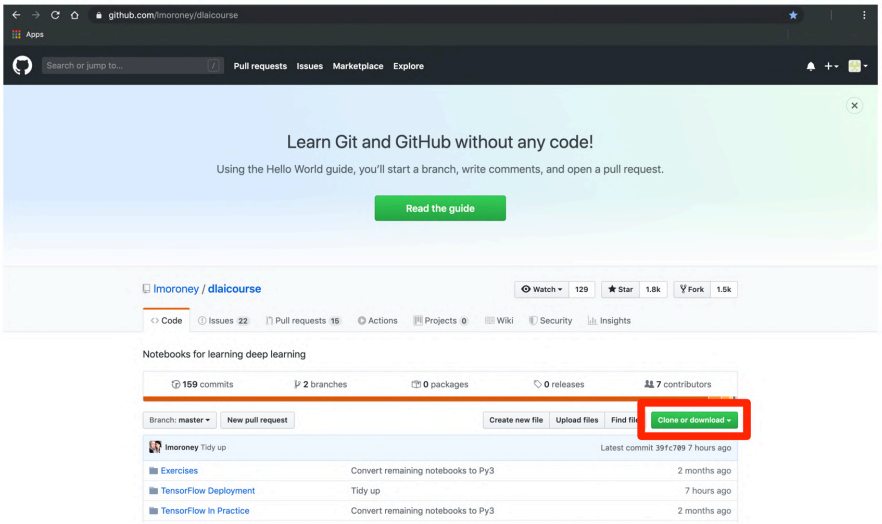
If you run into trouble installing Brackets in your Linux system, make sure to check out the Brackets Linux Guide.

That's it! You should now be all setup to follow along.

Downloading the Coding Examples and Exercises

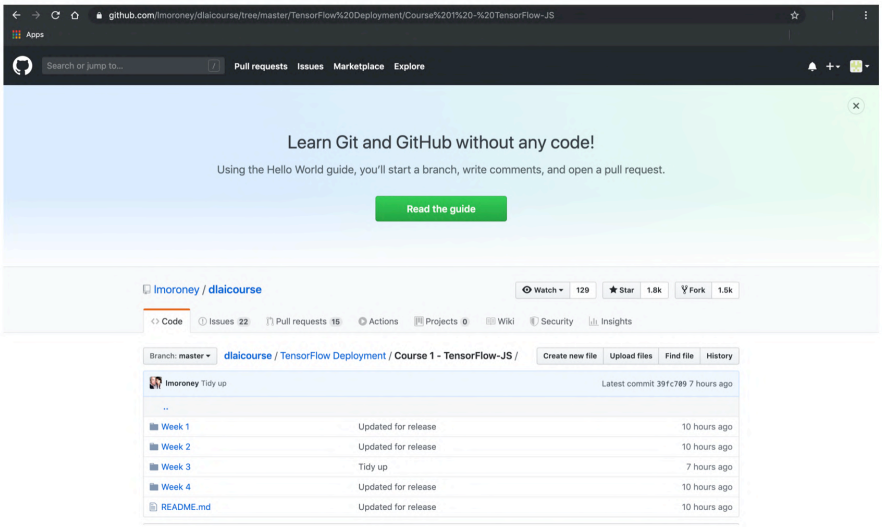
Like we mentioned earlier, in this course you will be running all the coding examples and exercises locally on your machine. We have created this GitHub Repository where you can find all the examples and exercises not only for this course but for the entire TensorFlow for Data and Deployment Specialization .

You can download all the examples and exercises to your computer by cloning or downloading the GitHub Repository.



You can find the corresponding coding examples and exercises for this course in the following folder in the GitHub repository:

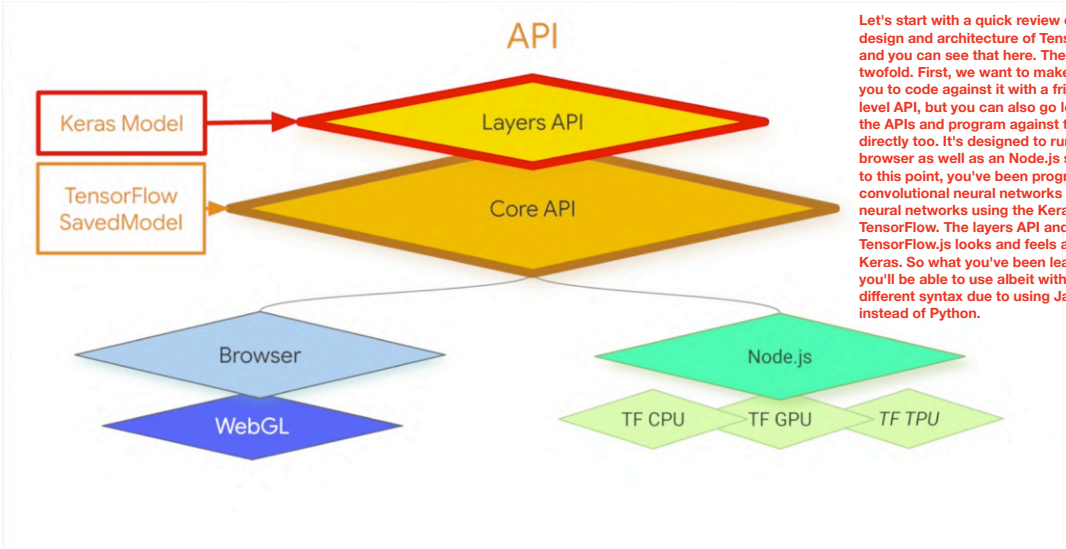
dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/



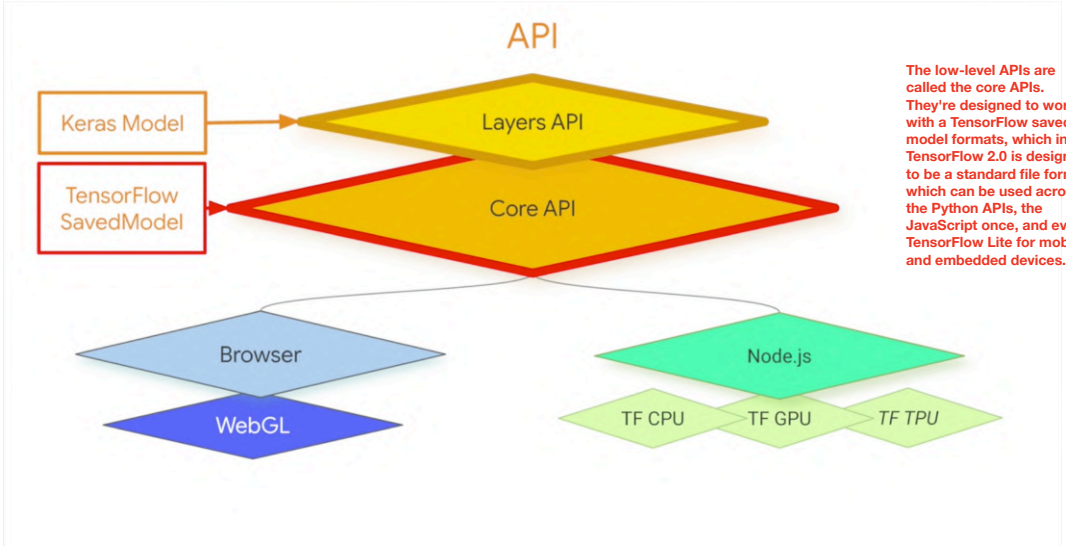
Each folder contains the corresponding examples and exercises for each week of this course on TensorFlow.js.

NOTE: The code in the repository is updated occasionally. Therefore the code in the repository may vary slightly from the one shown in the videos.

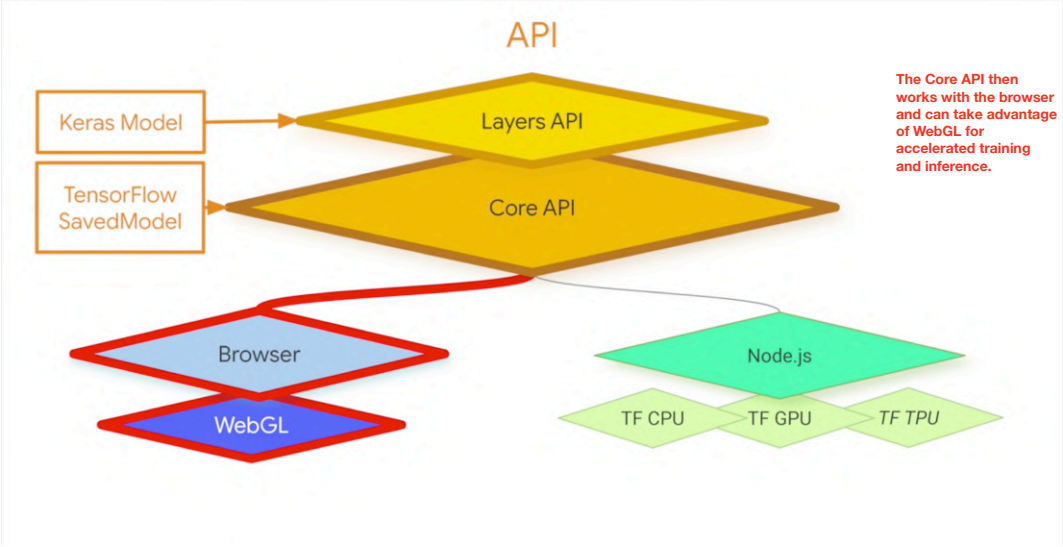
You're going to take a look at training and inference using JavaScripts. This will allow you to take your knowledge of Machine Learning models and use them in the browser as well as on backend servers like Node.js. We're going to start in the browser, and this week you're going to build some basic models using JavaScript, and you'll execute them in a simple web page.



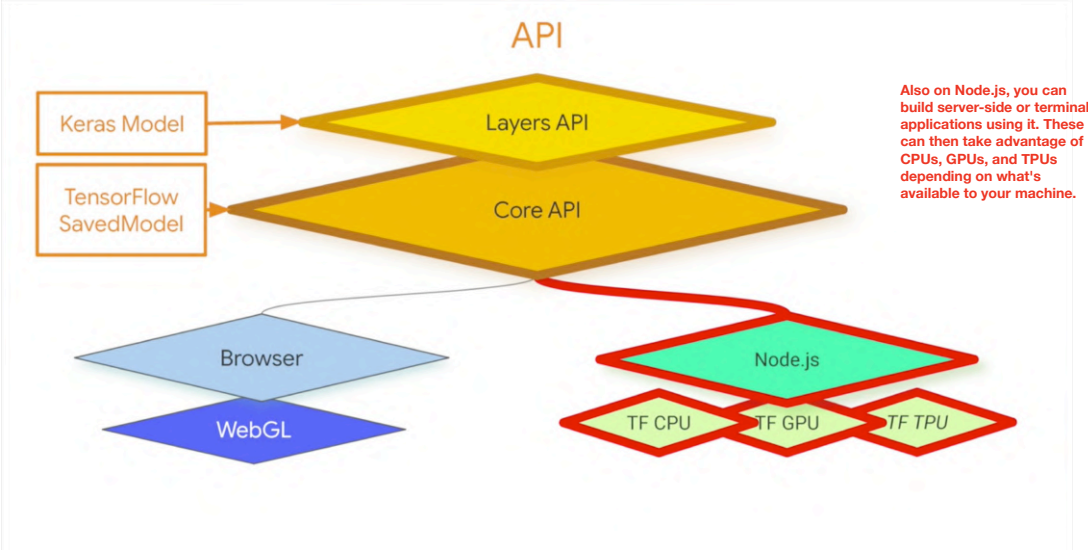
Let's start with a quick review of the design and architecture of TensorFlow.js, and you can see that here. The goal is twofold. First, we want to make it easy for you to code against it with a friendly high-level API, but you can also go lower into the APIs and program against them directly too. It's designed to run in the browser as well as an Node.js server. Up to this point, you've been programming convolutional neural networks and deep neural networks using the Keras API in TensorFlow. The layers API and TensorFlow.js looks and feels a lot like Keras. So what you've been learning, you'll be able to use albeit with a slightly different syntax due to using JavaScript instead of Python.



The low-level APIs are called the core APIs. They're designed to work with a TensorFlow saved model formats, which in TensorFlow 2.0 is designed to be a standard file format which can be used across the Python APIs, the JavaScript once, and even TensorFlow Lite for mobile and embedded devices.



The Core API then works with the browser and can take advantage of WebGL for accelerated training and inference.



Also on Node.js, you can build server-side or terminal applications using it. These can then take advantage of CPUs, GPUs, and TPUs depending on what's available to your machine.

```
<html>
<head></head>
<body>
    <h1>First HTML Page</h1>
</body>
</html>
```

You'll need to do is add a script tag below the head and above the body to load the TensorFlow.js file.

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
```

We're going to build a model that infers the relationship between two numbers where y equals 2x minus 1. So let's do that now in JavaScript.

Make sure this is above the body tag in your HTML page.

```
<script lang="js">
    const model = tf.sequential();
    model.add(tf.layers.dense({units: 1, inputShape: [1]}));
    model.compile({loss:'meanSquaredError',
                   optimizer:'sgd'});
    model.summary();
</script>
```

The simplest possible neural network is one layer with one neuron. So we're only adding one dense layer to our sequence. This dense layer has only one neuron in it, as you can see from the units equals one parameter.

```
<script lang="js">
    const model = tf.sequential();
    model.add(tf.layers.dense({units: 1, inputShape: [1]}));
    model.compile({loss:'meanSquaredError',
                   optimizer:'sgd'});
    model.summary();
</script>
```

We then compiled a neural network with a loss function and an optimizer as before. The loss function is Mean Squared Error, which works really well in a linear relationship like this one. The SGD and the optimizer stands for stochastic gradient descent as before.

```
<script lang="js">
    const model = tf.sequential();
    model.add(tf.layers.dense({units: 1, inputShape: [1]}));
    model.compile({loss:'meanSquaredError',
                   optimizer:'sgd'});
    model.summary();
</script>
```

```
<script lang="js">
    const model = tf.sequential();
    model.add(tf.layers.dense({units: 1, inputShape: [1]}));
    model.compile({loss:'meanSquaredError',
                   optimizer:'sgd'});
    model.summary();
</script>
```

Model.summary just outputs the summary of the model definition for us. You can see this in the console outputs.

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
dense_Dense1 (Dense)         [null,1]                  2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

Next up, before the closing script tag at this code, this is the data that you'll use to train the neural network.

Now, this is likely a little different than what you might be used to using from Python. So let me call that out somewhat. First, you'll notice that we're defining it as a tensor 2D, whereas in Python we were able to use a NumPy array. We don't have NumPy in JavaScript, so we're going a little lower.

```
const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
```

As its name suggests, when using a Tensor 2D, you have a two dimensional array or two one-dimensional arrays. So in this case you'll see that my training values are in one array, and the second array is the shape of those training values. So I'm using a set of 6x values in a one-dimensional array, and thus the second parameter is 6, 1, and I'll do the same for y. So if you tweak this code to add or remove parameters, remember to also add the second array to match its size.

```
const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
```

Training should be an asynchronous function because it will take an indeterminate time to complete. So your next piece of code will call an asynchronous function called doTraining, which we haven't written yet, but when it completes, it will actually do something.

Do training is an asynchronous function that you're creating in a few minutes. As mentioned before, training can take an indeterminate amount of time, and we don't want to block the browser while this is going on. So it's better to do it as an asynchronous function that calls us back when it's done. You call it and parse it the model that you just created.

```
doTraining(model).then(() => {
    alert(model.predict(tf.tensor2d([10], [1,1])));
});
```

Then when it calls back, the model is trained, and at that point we can call model.predict. We can use this, for example, to try to predict the value for 10. But note how we parse the data into the model. We again have to create a Tensor 2D with the first dimension being an array containing the value that we want to predict, in this case 10, and the second being the size of that array, which in this case is one by one.

```
doTraining(model).then(() => {
    alert(model.predict(tf.tensor2d([10], [1,1])));
});
```

Here's the complete asynchronous function for training the model. This code should go at the top of the script block that you've been creating.

As with Python model.fit is the function call to do the training. You're asking the neural network to fit the Xs to the Ys. As it's an asynchronous operation, in JavaScript, you await the result.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

You then pass it the Xs and the Ys as parameters.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

The rest of the parameters are adjacent lists, which you can see is enclosed in braces with each list item denoted by a name, followed by a colon, followed by a value.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

For 500 epochs, we enter it like this in the list.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

For callbacks instead of having a custom callback class, like we did in Python, we can just specify the callback in the list.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

The definition of the callback is itself a list with the list item on epoch end is defined as a function. This is a powerful aspect of JavaScript where I can add functions as list items.

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch:" + epoch + " Loss:" + logs.loss);
                      }
                  }
                });
}
```

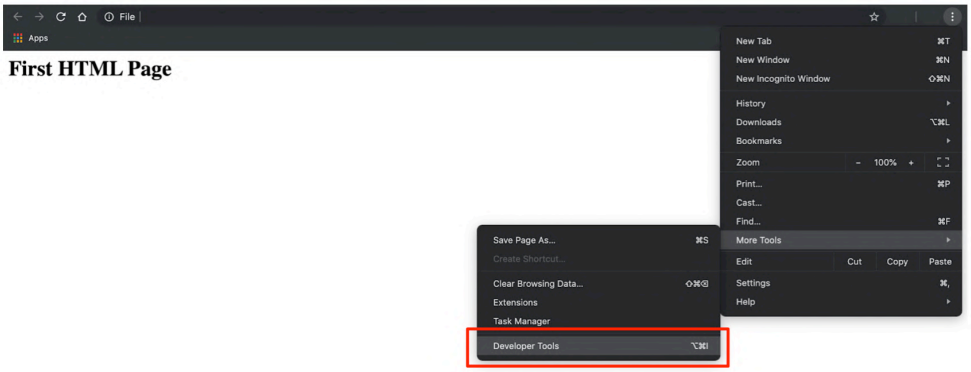
In this case, on epoch end gets the epoch number and the logs as parameters. So I can print out the epoch and the loss for that epoch.

Your First Model

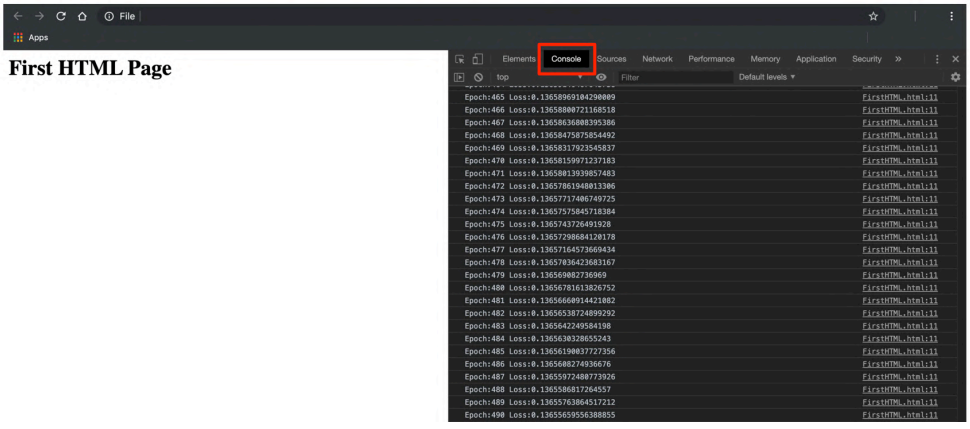
We are going to start by taking a look at a simple example. You can use Brackets to open the **FirstHTML.html** file and take a look at the code. You can find the **FirstHTML.html** file in the following folder in the GitHub repository:

[dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 1/Examples/]

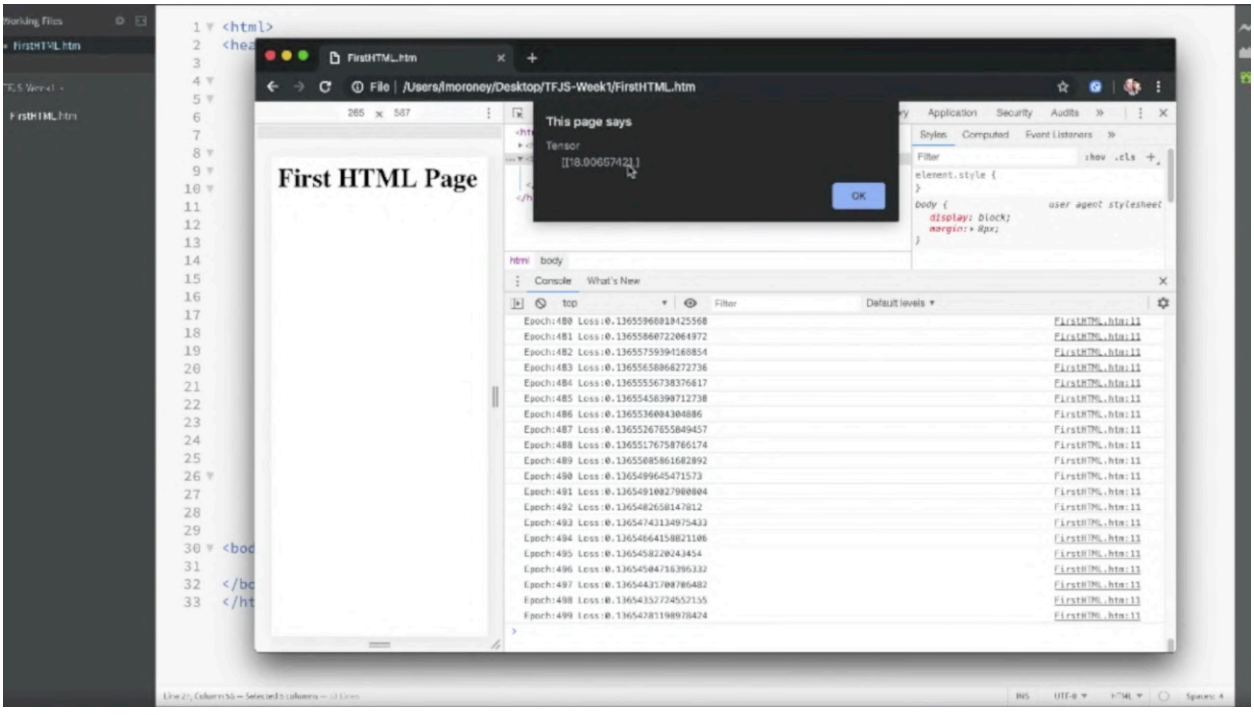
When you launch the **FirstHTML.html** file in the Chrome browser make sure to open the Developer Tools to see the training progress.



After opening the Developer Tools you should see the training progress in the Console:



```
<html>
<head></head>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
    <script lang="js">
        async function doTraining(model){
            const history =
                  await model.fit(xs, ys,
                    { epochs: 500,
                      callbacks:{
                          onEpochEnd: async(epoch, logs) =>{
                              console.log("Epoch:"
                                          + epoch
                                          + " Loss:"
                                          + logs.loss);
                          }
                      }
                    });
        }
        const model = tf.sequential();
        model.add(tf.layers.dense({units: 1, inputShape: [1]}));
        model.compile({loss:'meanSquaredError',
                       optimizer:'sgd'});
        model.summary();
        const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
        const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
        doTraining(model).then(() => {
            alert(model.predict(tf.tensor2d([10], [1,1])));
        });
    </script>
<body>
    <h1>First HTML Page</h1>
</body>
</html>
```



UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Iris Data Set

Download: Data Folder, Data Set Description

Abstract: Famous database; from Fisher, 1936

Data Set Characteristics:	Multivariate	**Number of Instances:**	150	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	4	**Date Donated**	1988-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	2512455

Source:

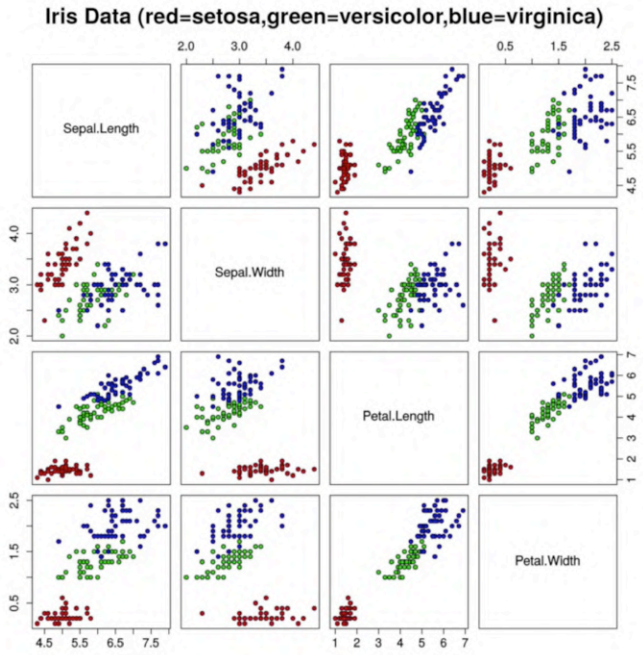
Creator:

R.A. Fisher

Donor:

Michael Marshall (MARSHALL%PLU '@' io.arc.nasa.gov)

https://archive.ics.uci.edu/ml/datasets/iris



By Nicoguaro - Own work, CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=46257808

https://archive.ics.uci.edu/ml/datasets/iris

One thing to note is that the first line should contain the column name definitions.

Don't forget these or your code won't be able to take advantage of the shortcuts and

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
4.9,3,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
4.6,3.1,1.5,0.2,setosa
5,3.6,1.4,0.2,setosa
5.4,3.9,1.7,0.4,setosa
4.6,3.4,1.4,0.3,setosa
5,3.4,1.5,0.2,setosa
```

Start by putting an asynchronous function into a JavaScript block. It has to be asynchronous because we will be awaiting some values for example when we're training.

```
async function run() {
}
```

```
const csvUrl = 'iris.csv';
const trainingData = tf.data.csv(csvUrl, {
    columnConfigs: {
        species: {
            isLabel: true
        }
    }
});
```

To load the data from the CSV, you'll use code like this. It uses the tf.data.csv class to handle loading and parsing the data, but there are a number of things you'll need to pay attention to. First of all, the CSV is at a URL. I don't have the server or protocol details, which means it's going to try to load it from the same directory as the web page it's hosting it. But it's important to note that it isn't loading from the file system directly. It's going through the HTTP stack to get the file, so you'll need to run this code on a web server. I'm using the brackets IDE in this course and it has a built-in web server and I'd recommend you use that or something similar.

```
const csvUrl = 'iris.csv';
const trainingData = tf.data.csv(csvUrl, {
    columnConfigs: {
        species: {
            isLabel: true
        }
    }
});
```

Tf.data.csv takes care of CSV management for you. It saves you writing a lot of code, so it's strongly recommended. To get the file all you need to do is make a call to it passing the URL.

```
const csvUrl = 'iris.csv';
const trainingData = tf.data.csv(csvUrl, {
    columnConfigs: {
        species: {
            isLabel: true
        }
    }
});
```

Tensorflow doesn't know from this file what your features are labels are so you have to at least flag, which column should be treated as label, you do that with this syntax. And note that species is hard coded and recognize. Look back to the CSV and you'll see that species is the column definition for the last column where we have the label. tf.data.csv is smart enough to use that string from the first line which is why you should leave the first line of column names in the CSV file.

```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0 ]

    return{ xs: Object.values(xs), ys:Object.values(labels)};

  }).batch(10);
```

```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0 ]

    return{ xs: Object.values(xs), ys:Object.values(labels)};

  }).batch(10);
```

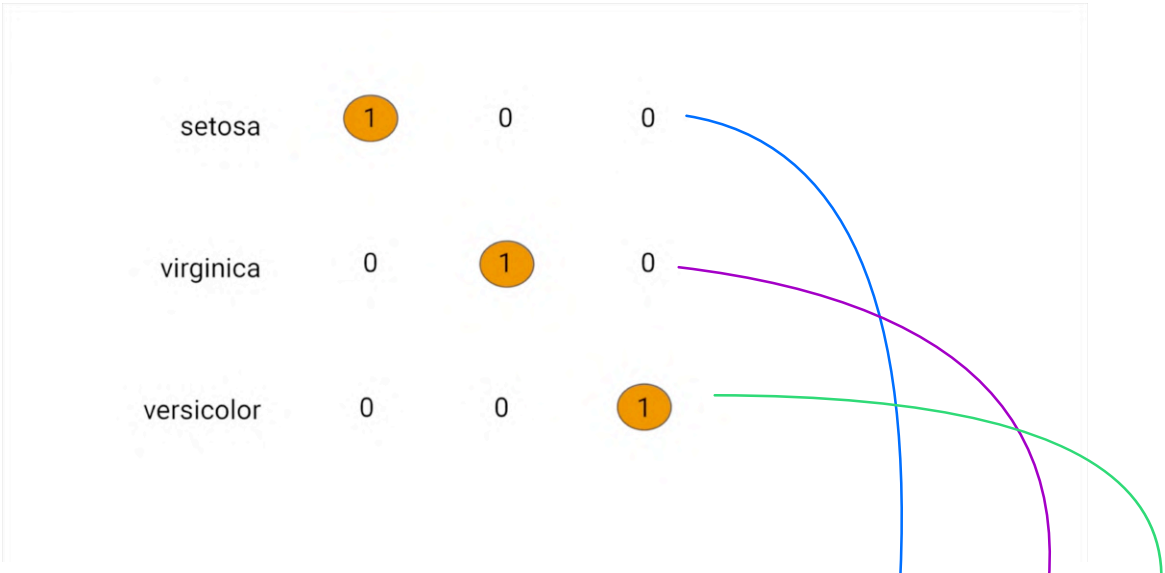
The column that we specified as a label the species is in the set of y's and we can process that set to create an array of values. In this case creating an array of three values per label. The values in the label will be 2 0 and 1 with the position of the 1 being based on the species. So if it's setosa, it will be in the first element of the array. If it's virginica, it will be the second. And if it's versicolor, it will be in the third. This is the one hot encoding that I mentioned earlier.

```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0 ]

    return{ xs: Object.values(xs), ys:Object.values(labels)};

  }).batch(10);
```

The values that weren't flagged as labels are in the x's set. So if I call objects.values(xs) on that, I will get back an array of arrays of their values. Each row in the data set had four features giving me a 4 by 1 array. And these are then loaded into an array with the length of the number of rows in the CSV in this case 150. So I will return that as my set of features that I'll train on.

```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0 ]

    return{ xs: Object.values(xs), ys:Object.values(labels)};

  }).batch(10);
```

After processing the text labels into one hot encoded arrays, I can now call object.values on this to get the array of arrays back also. I now have data that can be loaded into a model trainer.

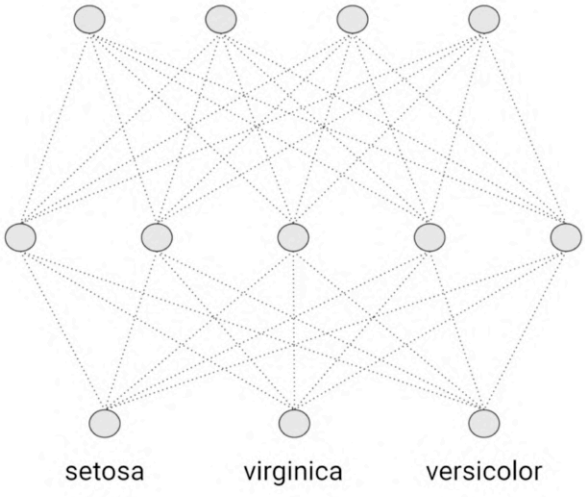



```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0 ]

    return{ xs: Object.values(xs), ys:Object.values(labels)};

  }).batch(10);
```

The first element is one if the species is setosa and zero otherwise.

The second element is one if the species is virginica and zero otherwise.

The third element is one if the species is versicolor, and zero otherwise.



```
const model = tf.sequential();

model.add(tf.layers.dense({
      inputShape: [numOfFeatures],
      activation: "sigmoid", units: 5}))

model.add(tf.layers.dense({activation: "softmax", units: 3}));
```

```
const model = tf.sequential();

model.add(tf.layers.dense({
      inputShape: [numOfFeatures],
      activation: "sigmoid", units: 5}))

model.add(tf.layers.dense({activation: "softmax", units: 3}));
```

Then we add the hidden layer with five neurons. By specifying the input shape with the number of features, which is calculated to be four, we are effectively doing what Flatten did in Python earlier on.

```
const model = tf.sequential();

model.add(tf.layers.dense({
      inputShape: [numOfFeatures],
      activation: "sigmoid", units: 5}))

model.add(tf.layers.dense({activation: "softmax", units: 3}));
```

```
model.compile({
    loss: "categoricalCrossentropy",
    optimizer: tf.train.adam(0.06)})
```

To do the training, we use model.fitdataset. This is a version of fits that you haven't seen before, but it's nice in that you don't have to do a lot of data pre-processing. You've done it already by creating the data as a dataset.

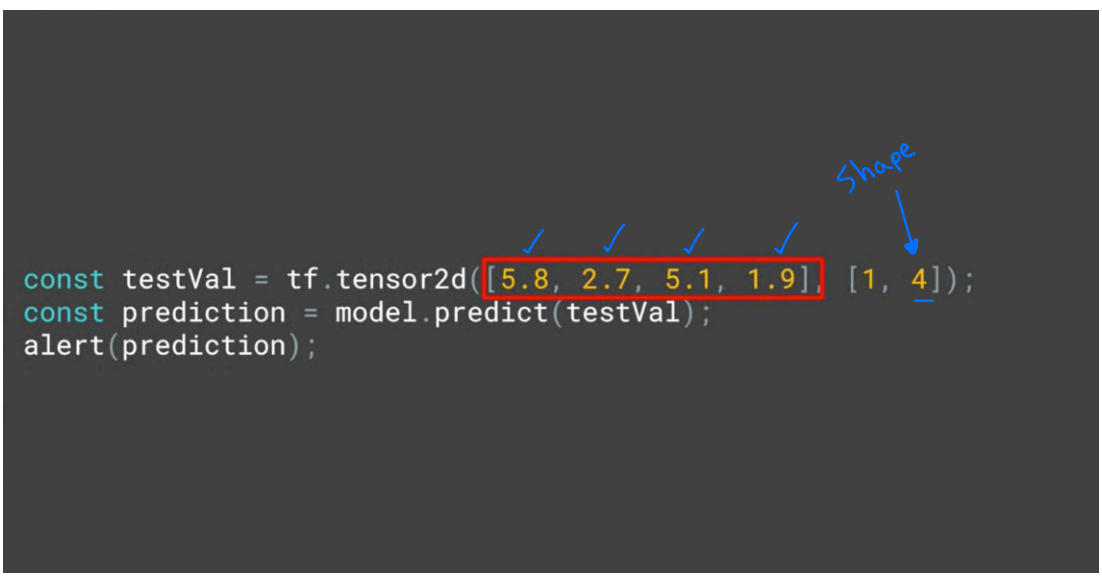
You can pass the data in as the first parameter as you can see here.

```
await model.fitDataset(
    convertedData,
    {
        epochs:100,
        callbacks:{
        onEpochEnd: async(epoch, logs) =>{
          console.log("E: " + epoch + " Loss: " + logs.loss);
        }
    }
});
```

```
await model.fitDataset(
    convertedData,
    {
        epochs:100,
        callbacks:{
        onEpochEnd: async(epoch, logs) =>{
          console.log("E: " + epoch + " Loss: " + logs.loss);
        }
    }
});
```

The callbacks specifies the list itself and which we specify the behavior on epoch end. Well, we'll just log the epoch number and the current loss.

```
await model.fitDataset(
    convertedData,
    {
        epochs:100,
        callbacks:{
        onEpochEnd: async(epoch, logs) =>{
          console.log("E: " + epoch + " Loss: " + logs.loss);
        }
    }
});
```



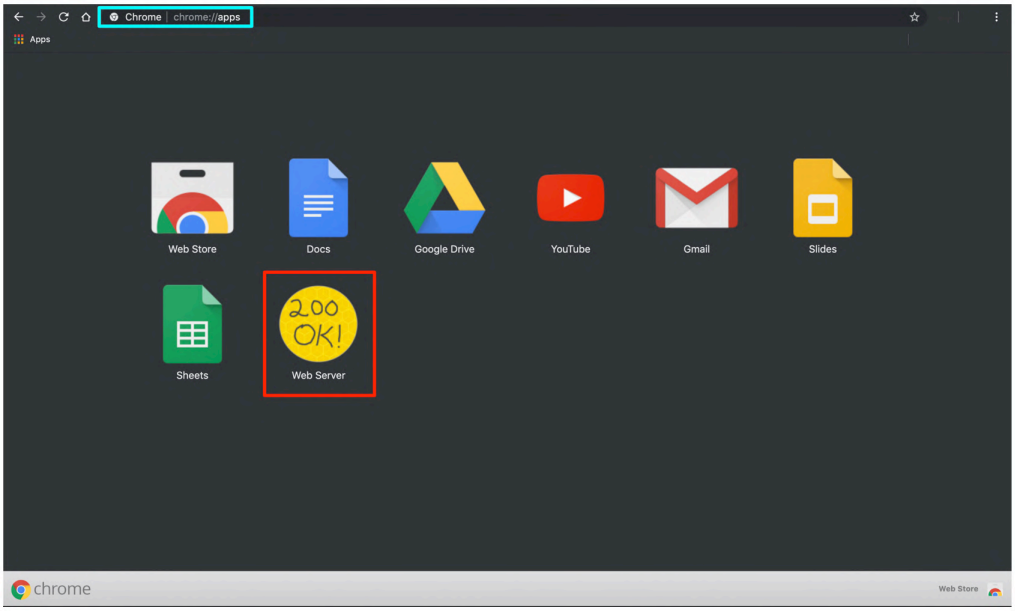
We then pass that to the predict method and we'll get a Tensor back with a prediction in it.

```
const testVal = tf.tensor2d([5.8, 2.7, 5.1, 1.9], [1, 4]);
const prediction = model.predict(testVal);
alert(prediction);
```

Using the Web Server

For the remaining examples and exercises in this course you will need to run a web server locally on your machine. This is because in javascript every call has to be done through and HTTP request. Therefore, even if you have your files locally, you can't just load them directly, you have to make an HTTP request to those files. This is what the Web Server for Chrome App is used for.

1. Open the Chrome browser and go to the Chrome apps (chrome://apps/):

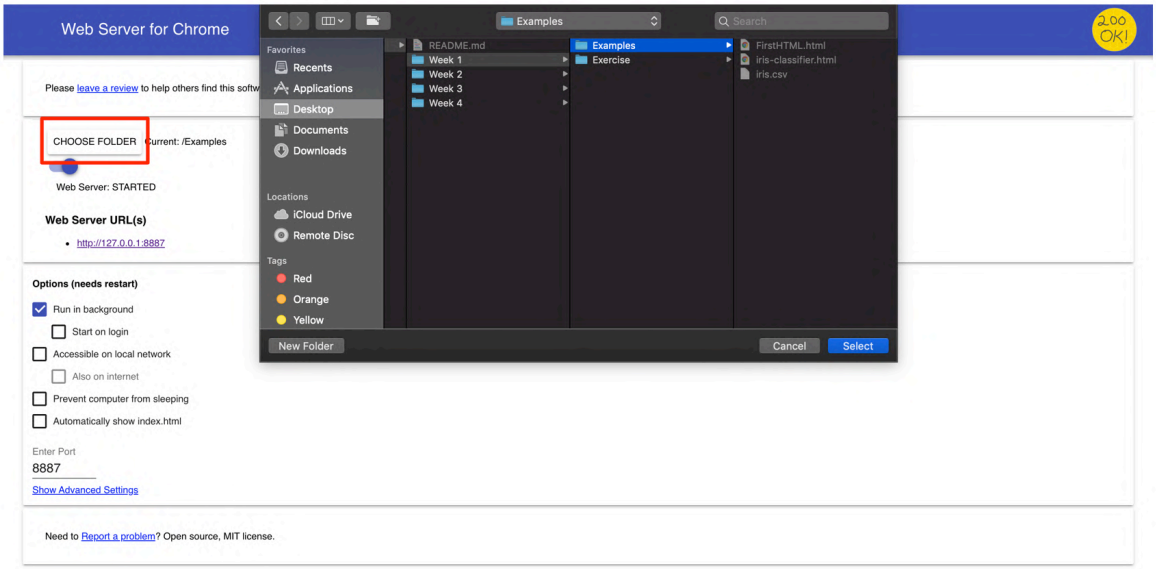


2. Click on the Web Server icon to launch it.

3. Click on "CHOOSE FOLDER" and select the folder that contains the examples or exercises you want to run. For this example, we are going to run the **iris-classifier.html** file. On my computer, this file is located in the following folder:

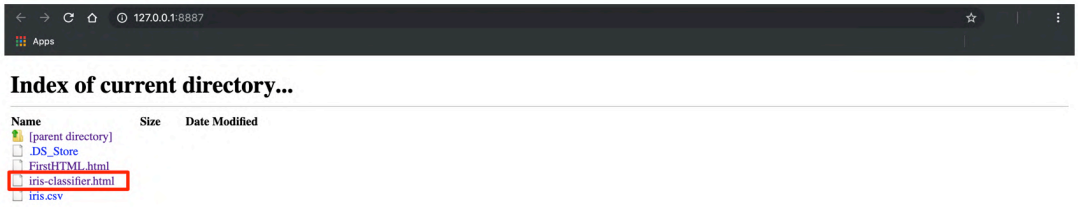
~/Desktop/dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 1/Examples/

Therefore, I will select that folder.



4. Once you have chosen the correct folder, you can click on the Web Server URL (http://127.0.0.1:8887).

5. Once you click on the Web Server URL, this will open a new tab in your Chrome browser. You can now click on the **html** file you want to run. In this case, we are going to run the **iris-classifier.html** file.

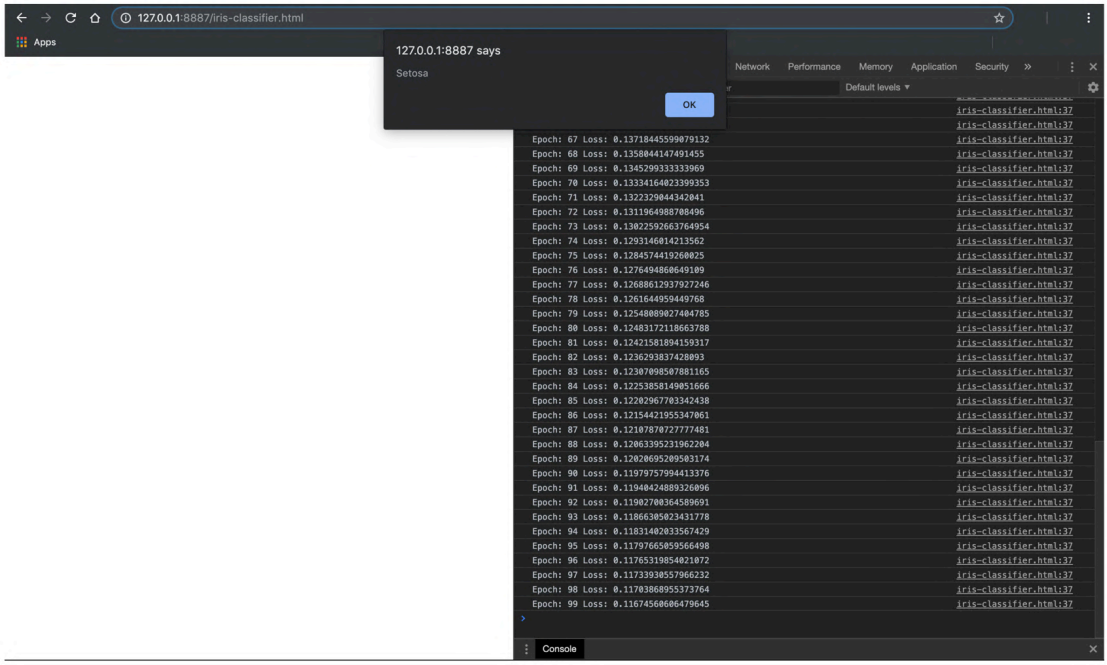


Iris Classifier

In the next example, we will create a neural network that can classify the 3 types of Iris plants found in the Iris dataset. You can use Brackets to open the **iris-classifier.html** file and take a look at the code. You can find the **iris-classifier.html** file in the following folder in the GitHub repository for this course:

[dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 1/Examples/]

When you launch the **iris-classifier.html** file in the Chrome browser (using the Web Server) make sure to open the Developer Tools to see the output in the Console. After training has finished, the code will alert the name of the predicted Iris plant. As you can see below, in this particular example, the predicted Iris plant is a Setosa.




```
<html>
<head></head>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
    <script lang="js">
        async function run(){
            const csvUrl = 'iris.csv';
            const trainingData = tf.data.csv(csvUrl, {
                columnConfigs: {
                    species: {
                        isLabel: true
                    }
                }
            });

            const numOfFeatures = (await trainingData.columnNames()).length - 1;
            const numOfSamples = 150;
            const convertedData =
                  trainingData.map(({xs, ys}) => {
                      const labels = [
                            ys.species == "setosa" ? 1 : 0,
                            ys.species == "virginica" ? 1 : 0,
                            ys.species == "versicolor" ? 1 : 0
                      ]
                      return{ xs: Object.values(xs), ys: Object.values(labels)};
                  }).batch(10);

            const model = tf.sequential();
            model.add(tf.layers.dense({inputShape: [numOfFeatures], activation: "sigmoid", units: 5}))
            model.add(tf.layers.dense({activation: "softmax", units: 3}));
            model.compile({loss: "categoricalCrossentropy", optimizer: tf.train.adam(0.06)});

```

```
            await model.fitDataset(convertedData,
                                {epochs:100,
                                 callbacks:{
                                     onEpochEnd: async(epoch, logs) =>{
                                         console.log("Epoch: " + epoch + " Loss: " + logs.loss);
                                     }
                                 }});

            const testVal = tf.tensor2d([4.4, 2.9, 1.4, 0.2], [1, 4]);
            const prediction = model.predict(testVal);
            alert(prediction)

        }
        run();
    </script>
<body>
</body>
</html>
```