```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```


```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```


Monochrome MNIST dataset


```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

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

To define the size of the convolutional filter, you'll use the kernel size property. By setting it to three, we are specifying that we want to use three by three filters.

```
model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

Our data will be in chunks of 5500 images at a time loaded in as an array of 5500 28 by 28 images with 10 labels. This array isn't the same shape as is needed for training. So flattened here we'll take the 28 by 28 pixels for each and flatten them out.

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
          kernelSize: 3, filters: 8, activation: 'relu'}));

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

model.add(tf.layers.conv2d({filters: 16,
          kernelSize: 3, activation: 'relu'}));

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

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

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
model.compile(
    { optimizer: tf.train.adam(),
      loss: 'categoricalCrossentropy',
      metrics: ['accuracy']
    });
```

To compile the model, as always, you will specify a loss function and an optimizer as well as any metrics that you might want to capture.

Something to note is that the parameters are passed in using a JavaScript dictionary hence the braces. If you're used to the Python way of doing it watch out for this, as it did cause me a lot of syntax

```
model.compile(
    { optimizer: tf.train.adam(),
      loss: 'categoricalCrossentropy',
      metrics: ['accuracy']
    });
```

Also, because it's a dictionary, you use name colon value, so optimizer: tf.train.adam as you can see here.

```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

Batching data for training instead of flooding the model with all of the data at once, it trains using a subset and then again with another subset, etc., is always a good idea. When doing it in the browser it's an even better idea so you don't lock up the browser itself.

```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

If you want the model to validate as it's training in order to report back an accuracy, then you would use a list of validation data like this.

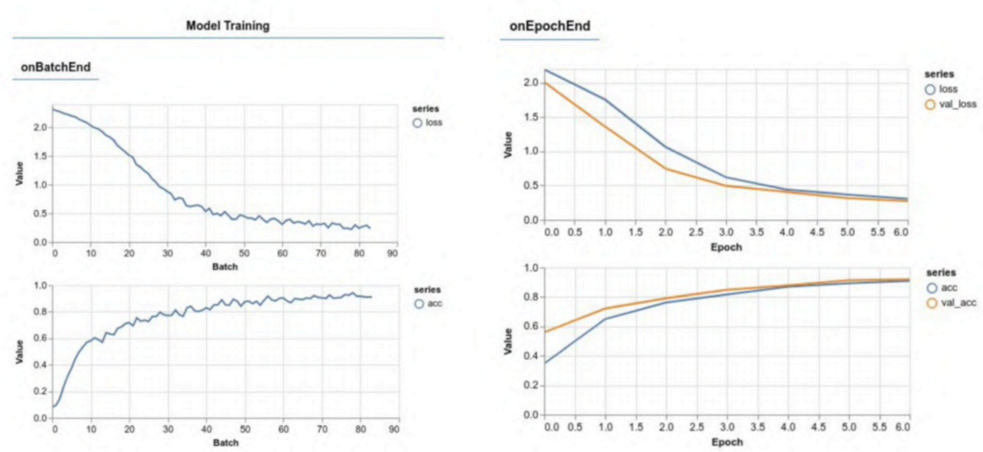
```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

If you want to shuffle the data to help randomize it for training, preventing potential over-fitting if multiple similar classes are in the same batch, then you can specify the shuffle option like this.

```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

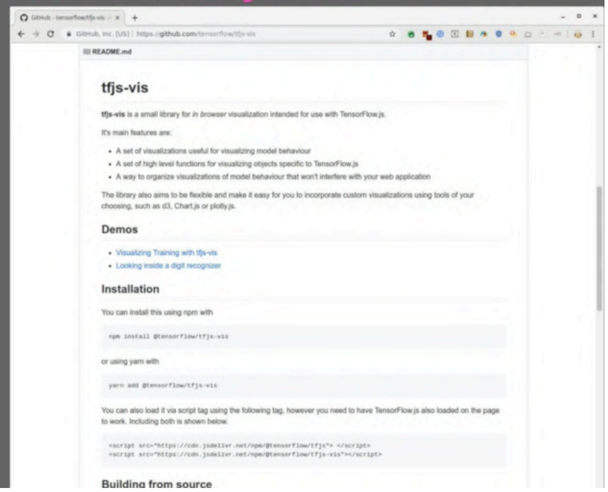
In Node.js, there's a really cool library called tf-vis, that you can use to render the outputs of your callbacks. We'll look at that in the next video.



include the library called tfjs-vis in your code with this script.

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

https://github.com/tensorflow/tfjs-vis



The library is open source, and you can get access to its source at this GitHub address. You'll also see the latest code there, and if there's updated details for how to include the script you'll find them at that site.

```
model.fit(trainXs, trainYs, {
    batchSize: BATCH_SIZE,
    validationData: [testXs, testYs],
    epochs: 20,
    shuffle: true,
    callbacks: fitCallbacks
});
```

To use the tf-visualization libraries with fitCallbacks, you simply declare it to be the return from tfvis.show.fitCallbacks. This function requires you to pass it a container where it will render the feedback, and a set of metrics that it should track.

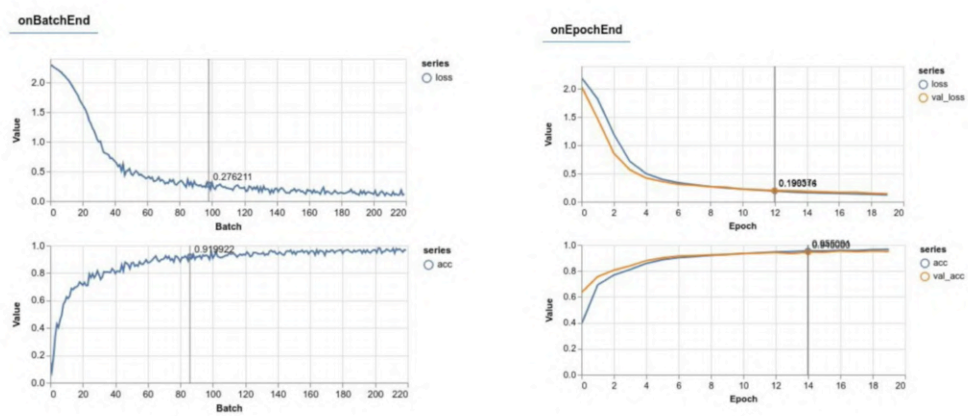
```
const fitCallbacks = tfvis.show.fitCallbacks(container, metrics);
```

So to declare them, you use this code. It's a straightforward as setting the metrics list to the metrics that you want to capture, like loss, validation loss, accuracy, and validation accuracy.

For the container, you just set a name and any required styles, and the visualization library will create the DOM elements to render the details.

```
const metrics = ['loss', 'val_loss', 'acc', 'val_acc'];
const container = { name: 'Model Training', styles: { height: '1000px' } };
const fitCallbacks = tfvis.show.fitCallbacks(container, metrics);
```

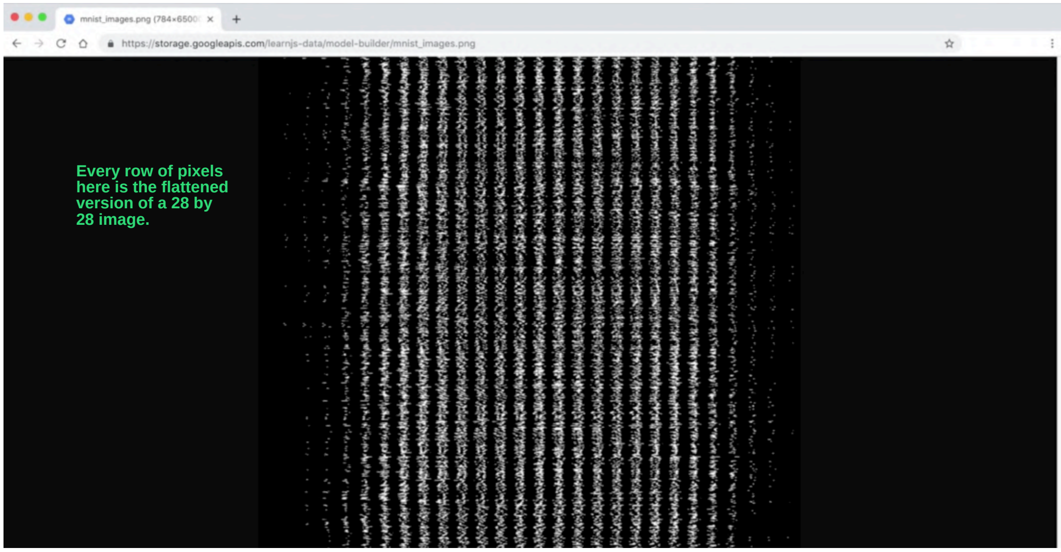
Now when you're training, the callback will create a container in which it will draw the feedback depending on the metrics that you select.



https://github.com/tensorflow/tfjs/tree/master/tfjs-vis

MNIST spreadsheet

```
https://storage.googleapis.com/learnjs-data/model-builder/mnist_images.png
```



The labels can be found at this URL. When you visit it, nothing will render in the browser, but a file will download. The file, if you inspect it, it's 650,000 bytes in size, which means that it is 10 bytes per image.

https://storage.googleapis.com/learnjs-data/model-builder/mnist_labels_uint8

```
           00 01 02 03  04 05 06 07  08 09
0000:0000  00 00 00 00  00 00 00 01  00 00
0000:000A  00 00 00 01  00 00 00 00  00 00
0000:0014  00 00 00 00  01 00 00 00  00 00
0000:001E  00 00 00 00  00 00 01 00  00 00
0000:0028  00 01 00 00  00 00 00 00  00 00
0000:0032  00 00 00 00  00 00 00 00  01 00
0000:003C  00 01 00 00  00 00 00 00  00 00
0000:0046  01 00 00 00  00 00 00 00  00 00
0000:0050  00 00 00 00  00 00 00 00  00 01
0000:005A  00 00 00 00  00 00 00 00  01 00
0000:0064  01 00 00 00  00 00 00 00  00 00
0000:006E  00 00 00 01  00 00 00 00  00 00
```

You'll need a hex viewer to understand the contents of the file.

Here's a screenshot where I used the viewer to take a look at it. With the 10 bytes per label, you simply have nine bytes that are zeros and one bite that's a one. This byte is the label value. So the first image is that of a seven, and then a three, and then a four, etc.

It's a very inefficient coding from a file size perspective, but it's very easy to transform it into a one-hot encoding in memory because it's a serialized one-hot encode already.

```
export class MnistData {
...
    async load() {
    // Download the sprite and slice it
    // Download the labels and decode them
    }

    nextTrainBatch(){
    // Get the next training batch
    }

    nextTestBatch(){
    // Get the next test batch
    }
}
```

The job of the load method is to download the sprite sheet, and labels, and decode them along with a helper function called next batch, which is used to batch them according to the specified train and test batch sizes.

```
export class MnistData {
...
    async load() {
    // Download the sprite and slice it
    // Download the labels and decode them
    }

    nextTrainBatch(){
    // Get the next training batch
    }

    nextTestBatch(){
    // Get the next test batch
    }
}
```

The train batch which gets the next batch of training data i.e. slices off the image according to the desired batch size.

Note that it keeps them as one by 784 pixels and the calling function can then resize them to 28 by 28. It also returns the appropriate labeled data

```
export class MnistData {
...
    async load() {
    // Download the sprite and slice it
    // Download the labels and decode them
    }

    nextTrainBatch(){
    // Get the next training batch
    }

    nextTestBatch(){
    // Get the next test batch
    }
}
```

Test batch does the same, but for testing data.

```
const data = new MnistData();
await data.load();
```

In order to initialize the data class and load the sprite getting it ready for batching, you only need this code.

```
const [trainXs, trainYs] = tf.tidy(() => {
    const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
    return [
        d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
        d.labels
    ];
});
```

We want to create an array containing the set of training Xs and training Ys, so this function will handle that.

```
const [trainXs, trainYs] = tf.tidy(() => {
    const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
    return [
        d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
        d.labels
    ];
});
```

It does this by getting the next training batch from the data source. By default with MNIST, the train data size is 5,500, so it's basically getting 5,500 lines of 784 bytes.

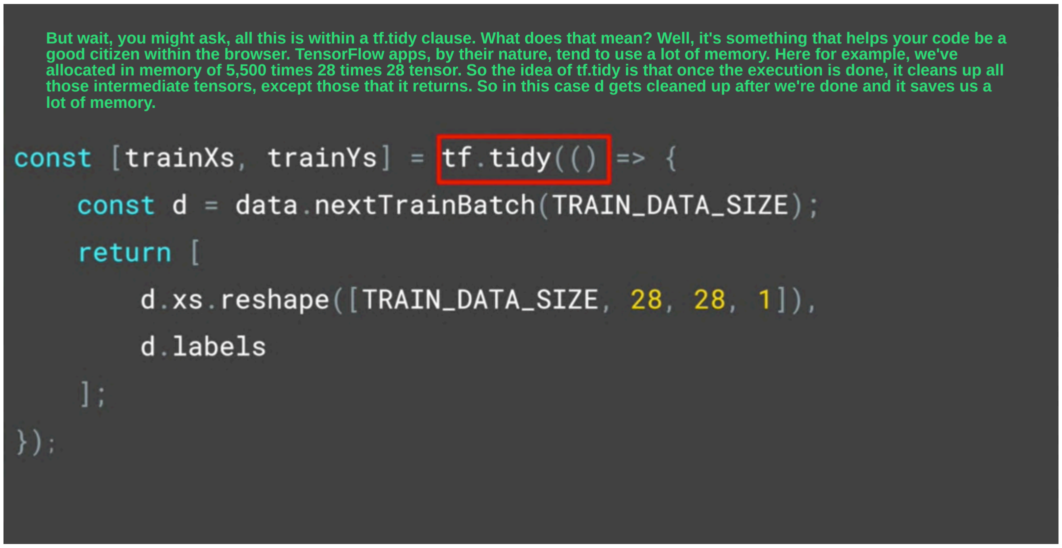
```
const [trainXs, trainYs] = tf.tidy(() => {
    const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
    return [
        d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
        d.labels
    ];
});
```

It then reshapes the data into a four-dimensional tensor with 5500 in the First Dimension. Then 28 by 28 representing the image, and then one representing the color depth. These images are monochrome, so it's just one in the color depth. It will return that as the first element in the array, mapping to train Xs.

```
const [trainXs, trainYs] = tf.tidy(() => {
    const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
    return [
        d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
        d.labels
    ];
});
```

As the labels are already one-hot encoded, it will return them as the second element in the array.

```
const [trainXs, trainYs] = tf.tidy(() => {
    const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
    return [
        d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
        d.labels
    ];
});
```

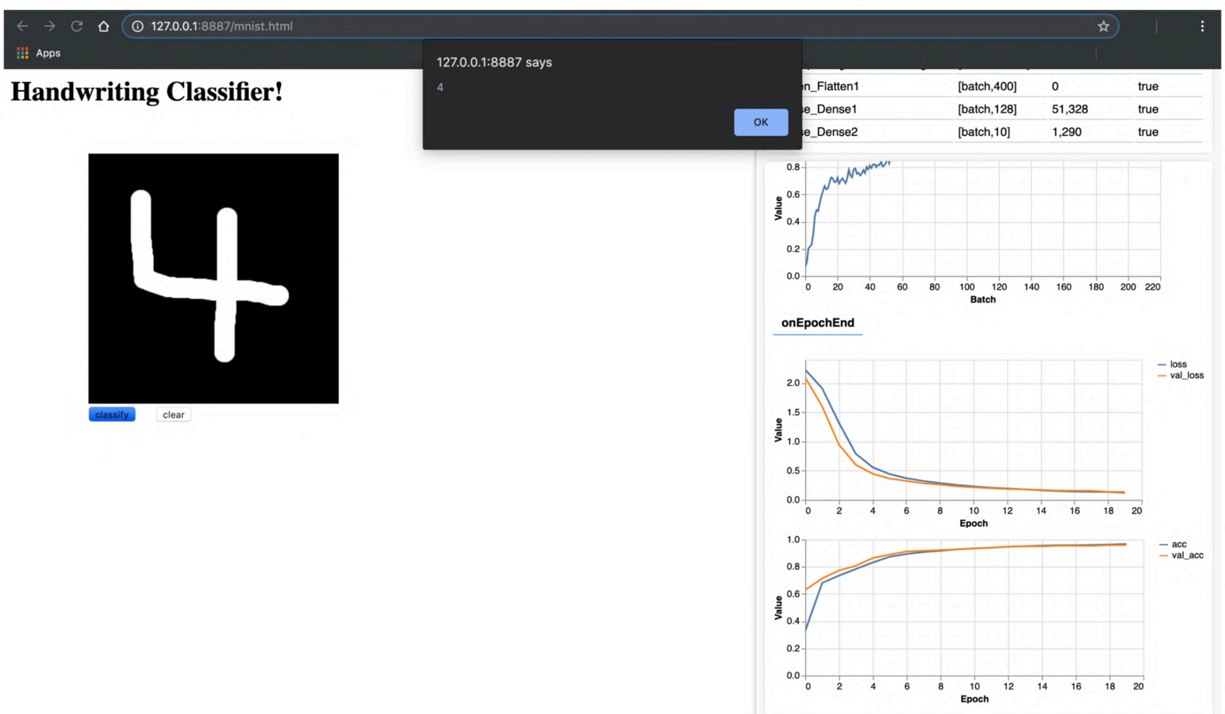


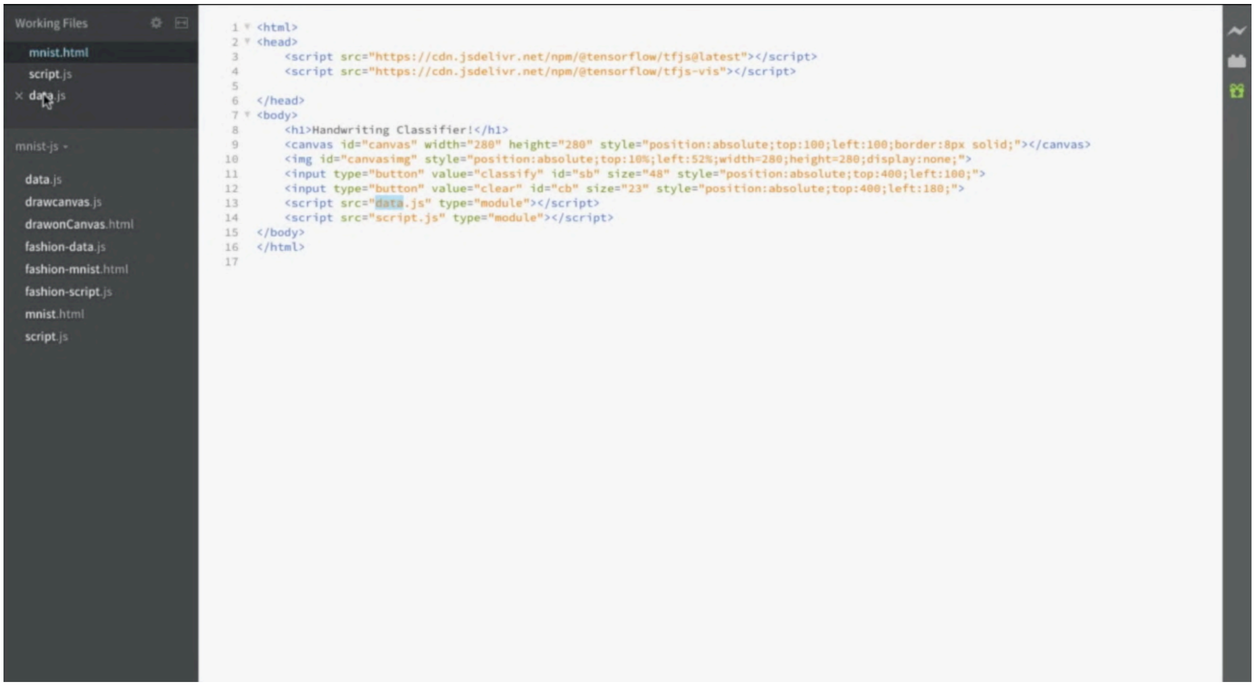
MNIST Classifier

In the next example, we will create a neural network that can classify the images of handwritten digits from the MNIST dataset. You can use Brackets to open the **script.js** file and take a look at the code. You can find the **script.js** file in the following folder in the GitHub repository for this course:

dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 2/Examples/

When you launch the **mnist.html** file in the Chrome browser (using the Web Server), tfjs-vis will automatically display the model architecture and the training progress. Once training has finished, you can draw digits on the black rectangle to be classified. After drawing a digit, and pressing the "classify" button, the code will alert the predicted digit. As you can see below, in this particular example, the predicted digit is a 4.




Working Files
mnist.html
script.js
data.js
mnist-js
data.js
drawcanvas.js
drawonCanvas.html
fashion-data.js
fashion-mnist.html
fashion-script.js
mnist.html
script.js
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis"></script>
</head>
<body>
<h1>Handwriting Classifier!</h1>
<canvas id="canvas" width="280" height="280" style="position:absolute;top:100;left:100;border:8px solid;"></canvas>
<img id="canvasimg" style="position:absolute;top:10%;left:52%;width=280;height=280;display:none;">
<input type="button" value="classify" id="sb" size="48" style="position:absolute;top:400;left:100;">
<input type="button" value="clear" id="cb" size="23" style="position:absolute;top:400;left:180;">
<script src="data.js" type="module"></script>
<script src="script.js" type="module"></script>
</body>
</html>