Contents

1 Browser Based Models with TensorFlow.js

1.1 Training and Inference using Tensorflow.js in JavaScript

We typically think of training a neural network using GPU's or a large data center, but modern web browsers have come a long way: they contain fully fleshed out runtime environments. One of the exciting aspects of Tensorflow.js is that it allows us to train neural networks and perform inference directly from a browser. We'll see how, for example, a user can upload a picture or grab a snapshot from a webcam and then train a neural network and/or perform inference right in the browser, without ever needing to send that image up to the cloud to be processed by a server. This saves time as we cut down on communication costs, allows us to run our models offline, and preserves user privacy.

1.1.1 Getting Your System Ready

We're going to learn how to run all the examples and exercises for this course locally on a single machine. We'll use Chrome for a browser, Brackets for an HTML editor, and the Web Server for Chrome App as our web server. We will also use github.com/lmoroney/dlaicourse as a repository to store homework assignments.

1.1.2 API Stack

- At the highest level, we have the *Keras Model* layers API which we've learned how to use in the deep learning specialization.
- Beneath this sits the *Core API*, which is backed by a TensorFlow saved model. Using this Core API, we can interact with either a browser or node.js, and this is critically what allows us to use things like layers from keras.
  - A browser sits ontop of webgl, a JavaScript API for rendering graphics; we implicitly have access to a GPU here!
  - *Node.js* can rely on a TensorFlow CPU, GPU, or TPU.

1.1.3 Building a First Model

Let's start by creating the simplest possible web-page.

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

We'll next need to 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 learns the relationship between two numbers x, y where their ground truth relationship is $y = 2x - 1$. We will do this in a *separate script block*, that needs to be placed 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 first line defines a *sequential* model. The second line adds a single hidden layer, itself containing a single hidden neuron. We then compile the model using mean-squared-error as our loss function, which will work well to model a linear relationship $y = 2x - 1$, and we choose stochastic gradient descent as our method of optimization. Note that in the model summary, we'll be told there are two parameters in the model, since we are learning both a weight *and* a bias term. Before closing the script tag, let's insert some data that will be used to train the neural network.

```
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]);
```

Notice that we're defining this as a `tensor2d`, since we don't have something like `numpy` from Python. As its name suggests, we must specify the extents of the two dimensions via the second argument.

1.1.4 Training the Model

Training should by *asynchronous*, because it takes an indeterminate amount of time to complete. Our next piece of code will call an asynchronous function called `doTraining`, which, when it completes execution will do something. Because training can take an indeterminate amount of time, we don't want to block the browser while this is taking place, so instead we specify it as an asynchronous function that calls us back when it's done.

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

We call the function by passing it a model we just created above; when it calls back, the model is trained and at that point we can call `model.predict()`. In this example, we're predicted $\hat{y}$ for an input of $x = 10$. Note that we still must use a `tensor2d`, which in this case is just a scalar i.e. a tensor of dimension 1×1. To actually define an asynchronous training function, we can do so as follows:

```
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);
                      }
                  }
                });
}
```

This code should be placed at the top of the script block that we've been creating. Because `doTraining` is asynchronous, we use keyword `await` to wait for the result. Note that after feeding the `xs` and `ys` as input arguments, the rest of the input is a JSON list, with each list-item denoted by a name followed by a

colon, followed by a value. For callbacks, we can specify it on the fly in the list, where the callback itself is defined as a `list` and `onEpochEnd` is a function. To be clear, we're adding a function as a list-item. In our example, upon each epoch ending, we take the epoch number and `logs` as parameters so we can print out the information to console.

Because `doTraining` is asynchronous, when we call it we'll use a `then` clause to specify what should happen upon completion of execution.

1.2 Training Models with CSV Files

The Iris Dataset We've covered a simple example where we stored the values of our data in memory. A more commmon scenario is when data comes in from an outside source such as a database connection or an imported dataset. One of the most common ways of getting data into an ML model, and JavaScript ones are no exception, is reading data from CSV files. TensorFlow.js provides facilities for this. We'll work with an *Iris* dataset, which contains 150 samples taken from three types of Iris flower with 50 from each type. There are four features and a label for each observation.

1.2.1 Reading the Data

How can we do this using JavaScript and TensorFlow.js. Before we code, we can examine our CSV file. Note the first line contains column headers. The features encode $\{\text{sepal}, \text{petal}\} \times \{\text{length}, \text{width}\}$. We'll start by placing an asynchronous function into a JavaScript block; it must be asynchronous because we'll be waiting for some values, e.g. while training.

```
asynch function run() {
}
```

From here on out in this running example, the remainder of the code will be placed within the definition of this `run()` function. When we are done defining it, we will call it in our script. The first things we'll do is to load the data from a CSV, for which we'll use the `tf.data.csv` class to handle parsing the data.

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

There are some important details to note. The CSV resides at a URL: we don't have the server or protocol details, which means its going to try and load it from the same directory as the web-page which is hosting the application. But, it's critical to note that we're not 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.[1] In defining our training data, we make a call to `tf.data.csv` passing the URL; since tensorflow doesn't know anything about our features or labels, and so we instruct it which is our label via list syntax. In particular, we're specifying that the `Species` column is a label. After having loaded our data in this way, tensorflow.js will recognize that the Species column is a label and therefore part of the `y` set, and the remaining features are part of the `x`'s set.

The data are returned from `tf.data.csv` as dictionaries, and for training we'll want to convert them into arrays. We'll also one-hot encode our string labels. To create a one-hot encoding, we can do so as follows:

[1] What's nice about the Brackets IDE is that it has a built-in web server.

```
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);
```

In calling the `map` method on our `trainingData`, we will iterate over each training example. The values that werent flagged as labels are in the `xs` data-structure. If we call `Object.values(xs)`, we get back an array-of-arrays containing their values. Each row in the data-set had four features which yields a 4×1 array. These are then loaded into an array of length the size of the dataset (in this case 150 examples). Notice that we're also calling `Object.values(labels)`, which returns an array-of-arrays back as well (in this case each label is a 3×1 array and we'll have 150 of them). Ultimately, our function returns a set of features that we'll train on alongside one-hot encoded labels.

1.2.2 Designing the Neural Network

We'll choose to create a neural network with the following architecture: four features map to a single hidden layer with five neurons which then map to an output layer with three nodes that we'll use for classification. This is how what that looks like in code:

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

model.add(tf.layers.dens({
  input.shape: [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("E:␣" + epoch + "␣Loss:␣" + logs.loss);
      }
    }
  });
```

Notice that we're passing our `convertedData` as argument to our training method. Here, we're using a slightly different function: `model.fitDataset`. By creating importing our CSV file as a dataset, we've already done a lot of the pre-processing necessary to plug-and-play with this function. We pass the data in as first parameter, then pass a list of JSON style name values with things such as number of epochs or callback behaviors defined. If we want to use the model to do inference and get a prediction, we must first create an input tensor with feature values and pass it to the predict method of the model. Also, it's critical

that we use `await` so that we don't proceed with execution and try to make a prediction (see below) while our model is still training.

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

If we hadn't used `await` above when training our model, our call to predict would yield funky results as we'd be performing inference on a model that isn't (fully) trained. Here, the `testVal` object holds the sepal and petal length and width; when we pass it to the predict method, we get a tensor back with a prediction in it. Here's the script in its entirety.

```
<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);
                                }
                            }});

            // Test Cases:

            // Setosa
            const testVal = tf.tensor2d([4.4, 2.9, 1.4, 0.2], [1, 4]);

            // Versicolor
            // const testVal = tf.tensor2d([6.4, 3.2, 4.5, 1.5], [1, 4]);

            // Virginica
            // const testVal = tf.tensor2d([5.8,2.7,5.1,1.9], [1, 4]);

            const prediction = model.predict(testVal);
            const pIndex = tf.argMax(prediction, axis=1).dataSync();

            const classNames = ["Setosa", "Virginica", "Versicolor"];

            // alert(prediction)
            alert(classNames[pIndex])

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

Instructions for Running the Iris-Classifier Application First, we must open our web-server application; once launched, we must select which folder we'd like to run out of, which in this case is `dlaicourse.TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 1/Examples/`. Once the folder is selected, click on the Web-Server URL which then opens a new tab in your Chrome browser. From here, we can click on the `html` file we want to run, in this case the `iris-classifier.html` file.

1.3 Creating Convolutional Neural Networks in JavaScript

We've seen how we can get a simple neural network running in a browser, and this next section will teach more advanced concepts. Specifically, we'll build a convolutional neural network in a web-browser that can process images. In particular, we're going to work with MNIST dataset, which consists of tens-of-thousands of images; opening up 10k HTTP connections would be problematic, and instead we'll see how we can combine all images into a single spreadsheet file which can then be donwloaded, sliced, and trained on.

1.3.1 Creating a Convolutional Net with JavaScript

Although we're familiar with coding CNN's using Python, there are a few minor syntactical changes to be aware of when coding in JavaScript. The largest discrepancies are outside of the model definition, with e.g. code that fetches the images or processes an image for classification. In this section, we will

- train a convolutional neural network for image classification in the browser, and
- write a browser app that takes these images and passes them to the classifier.

Let's look at the code for creating a conv-net in JavaScript.

```
model = tf.sequential();
model.add(tf.layers.conf2d({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'}));
```

By declaring our model as a `tf.sequential()`, we are specifying that the model will be defined via a *sequence* of layers. Our first layer is a 2-d convolution, which takes in our 28×28 MNIST dataset in monochrome format. The parameter `kernelSize: 3` indicates we want 3×3 filters. By using a Rectified-Linear Unit, we're filtering out values less than zero. The next step specifies a 2×2 max-pooling step. The call to `tf.layers.flatten()` takes our 2-D image array and flattens it into a 1-D vector, from which we will attempt to learn a mapping to the 10 different possible output digits, i.e. the labels. To compile the model, we must specify a loss function and an optimizer, as well as any metrics we'd like to track; note that we pass parameters via a JavaScript dictionary (hence the braces).

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

Training the model is done with the `fit()` method.

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

We'll train our data in batches, which aside from theoretical benefits also prevents locking up the browser itself. Notice as well that we're allowing validation data to be passed so that we may report validation accuracy as we train. Shuffling the data helps prevent over-fitting if our data is grouped by label. We can of course also specify custom callbacks; we'll look at how to use `tf-vis` to render the outputs of our callbacks in the next section.

1.3.2 Visualizing the Training Process

There are some extra tools we can use to make the visualization of training a lot more friendly. The first step is to include the library `tfjs-vis` in your code with a script.

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

We can get access to the code and documentation at `https://github.com/tensorflow/tfjs-vis`. Previously, we had used a term `fitCallbacks` without talking about how its defined.

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

The `container` is responsible for rendering the feedback, and the `metrics` are what we are interested in tracking.

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

When we create the container, we just set a name and any required styles, and the visualization library will create the DOM elements to render the details. While training, the callback will create a container in which it will draw the feedback depending on the metrics we requested. Let's now segway to discuss how we handle training in browsers with lots of data.

1.3.3 Sprite Sheets

The last few videos shows how to train a model and visualize training progress. But in any training, we need data. MNIST contains 10's of thousands of images. When loading into Python, there's no issue. But in JavaScript we'd be making an HTTP request for each image, which at the very least is bad practice. One solution is to take all the images and stitch them into a sprite-sheet, which is a single spreadsheet which contains all images in the dataset. In this way, we can prevent excessive multiple HTTP calls to download the data. The MNIST sprite-sheet is stored here: `https://storage.googleapis.com/learnjs-data/model-builder/mnist_images.png`.

Using the Sprite-Sheet If we visit the above url, we'll see a single image that's 65,000 pixels tall and 784 pixels wide. Each 28×28 image is encoded into a row ($28^2 = 784$). The labels can be found at `https://storage.googleapis.com/learnjs-data/model-builder/mnist_labels_uint8`. The file is 650k bytes in size, i.e. 10 bytes per image. To understand this binary file, we'll need a hex-viewer, but if we open it in this way we can see that each row corresponds to a one-hot encoding of the label (a digit 0-9); in each row of the file there are nine zero bytes and one byte that is flipped on. Although this is not necessarily an efficient file format, it's easy to one-hot encode in memory because it's serialized as a one-hot encoding already. We have a `data.JS` file which is intended to handle this.

```
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 spritesheet and labels, and decode them along with a helper function called `nextBatch` which is used to batch into specified train and test batch sizes. The other methods are responsible for getting batches of training data, i.e. slices off the image according to the desired size. Note that the `nextTrainBatch` keeps each image as a 1×784 vector of pixels and the calling function can resize them to 28×28. It also returns the appropriate labeled data. In order to initialize the data class and load the sprite, getting it ready for batching, all we need is the following code.

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

Once you have a loaded instance of the data, we can now get the batches and resize them to be 28×28 like this.

```
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 intend to create an array containing the set of Xs and Ys for training. Our function does this by getting the next batch from the training data source. By default, with MNIST the training data size is 5,500, and so we're basically fetching 5,500 lines of 784 bytes. We then reshape the data into a four-dimensional tensor with 5,500 in the first dimension, then 28×28 representing the image, then 1 representing the monochromatic color-depth. That's the first element of the array, the second is the labels which are already one-hot encoded.

Tidy Clause The `tf.tidy()` let's us be good citizens of the web. The idea behind `tf.tidy()` is that once it's done using the allocated memory for the $5,500 \times 28 \times 28$ tensor, the runtime environment cleans up afterward. I.e. `d` gets cleaned up after we're done and it saves us a lot of memory. In particular, when it is executed, it cleans up all intermediate tensors allocated by a function excehpt those returned by the function.

1.3.4 MNIST Classifier in Code

In its entirety.

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

At the head, we load in the latest version of TensorFlow.js alongside the `tfjs-viz` library which will give us a progress report as we train. In the body of the page, we create a canvas of size 280×280 such that it will scale down nicely to our MNIST input image size of 28×28. When you draw something on the canvas, in order to create an image we'll need an image id and so we do that in the next line of code. We'll also add two buttons: one for *classifying* and another for *clearing* the canvas. And then critically, there are two scripts we call; the first is `data_mnist.js` which has the MNIST class we wrote previously: it downloads the sprite, slices it up, downloads the labels, slices them up, and gives you batches you can load into the classifier. The script `script_mnist.js` contains everything else: it has the magic for handling the canvas, as well as model training and inference.

data.js The most important part to pay attention to is the `load()` method, and `nextTrainingBatch()`. We first define some constants in our script, in particular our `load()` method will rely on the `MNIST_IMAGES_SPRITE` and `MNIST_LABELS_PATH`.

```
const IMAGE_SIZE = 784;
const NUM_CLASSES = 10;
const NUM_DATASET_ELEMENTS = 65000;

const TRAIN_TEST_RATIO = 5 / 6;

const NUM_TRAIN_ELEMENTS = Math.floor(TRAIN_TEST_RATIO * NUM_DATASET_ELEMENTS);
const NUM_TEST_ELEMENTS = NUM_DATASET_ELEMENTS - NUM_TRAIN_ELEMENTS;

const MNIST_IMAGES_SPRITE_PATH =
    'https://storage.googleapis.com/learnjs-data/model-builder/mnist_images.png';
const MNIST_LABELS_PATH =
    'https://storage.googleapis.com/learnjs-data/model-builder/mnist_labels_uint8';
```

We create download an image, slice it out, and store it in memory.

```
  async load() {
    // Make a request for the MNIST sprited image.
    const img = new Image();
    const canvas = document.createElement('canvas');
    const ctx = canvas.getContext('2d');
    const imgRequest = new Promise((resolve, reject) => {
      img.crossOrigin = '';
      img.onload = () => {
        img.width = img.naturalWidth;
        img.height = img.naturalHeight;

        const datasetBytesBuffer =
            new ArrayBuffer(NUM_DATASET_ELEMENTS * IMAGE_SIZE * 4);

        const chunkSize = 5000;
        canvas.width = img.width;
        canvas.height = chunkSize;

        for (let i = 0; i < NUM_DATASET_ELEMENTS / chunkSize; i++) {
          const datasetBytesView = new Float32Array(
              datasetBytesBuffer, i * IMAGE_SIZE * chunkSize * 4,
              IMAGE_SIZE * chunkSize);
          ctx.drawImage(
              img, 0, i * chunkSize, img.width, chunkSize, 0, 0, img.width,
              chunkSize);

          const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

          for (let j = 0; j < imageData.data.length / 4; j++) {
            // All channels hold an equal value since the image is grayscale, so
            // just read the red channel.
            datasetBytesView[j] = imageData.data[j * 4] / 255;
          }
        }
        this.datasetImages = new Float32Array(datasetBytesBuffer);

        resolve();
      };
      img.src = MNIST_IMAGES_SPRITE_PATH;
    });
```

Similarly for the labels, we download the labels which are already one-hot encoded.

```
    const labelsRequest = fetch(MNIST_LABELS_PATH);
    const [imgResponse, labelsResponse] =
        await Promise.all([imgRequest, labelsRequest]);
```

Then there's the function that's responsible for procuring the next training batch; it takes in the batch size and takes that slice off of the images and returns that for us as a *set* of images (and ditto for the test batch). Since there are 55k training images and 10k testing images, so this will get us 10 batches of 5,500 of training and 10 batches of 1k for testing.

```
  nextTrainBatch(batchSize) {
```

```
    return this.nextBatch(
        batchSize, [this.trainImages, this.trainLabels], () => {
          this.shuffledTrainIndex =
              (this.shuffledTrainIndex + 1) % this.trainIndices.length;
          return this.trainIndices[this.shuffledTrainIndex];
        });
  }

  nextTestBatch(batchSize) {
    return this.nextBatch(batchSize, [this.testImages, this.testLabels], () => {
      this.shuffledTestIndex =
          (this.shuffledTestIndex + 1) % this.testIndices.length;
      return this.testIndices[this.shuffledTestIndex];
    });
  }
```

This next function is the one that actually does the bookkeeping, by managing the index within the array of the data to be able to know where to slice from to get the next batch.

```
  nextBatch(batchSize, data, index) {
    const batchImagesArray = new Float32Array(batchSize * IMAGE_SIZE);
    const batchLabelsArray = new Uint8Array(batchSize * NUM_CLASSES);

    for (let i = 0; i < batchSize; i++) {
      const idx = index();

      const image =
          data[0].slice(idx * IMAGE_SIZE, idx * IMAGE_SIZE + IMAGE_SIZE);
      batchImagesArray.set(image, i * IMAGE_SIZE);

      const label =
          data[1].slice(idx * NUM_CLASSES, idx * NUM_CLASSES + NUM_CLASSES);
      batchLabelsArray.set(label, i * NUM_CLASSES);
    }
```

script.js In particular, notice that at the very bottom there is a call which signifies that as soon as the document is loaded it'll call the `run()` function.

```
document.addEventListener('DOMContentLoaded', run);
```

Let's see what the `run()` function does. It creates an object called `data` which is a new instance of our MNIST class. When it calls `load()`, it fetches the sprite and loads the images into memory. Next, we set up a model, which we'll look at momentarily.

```
async function run() {
        const data = new MnistData();
        await data.load();
        const model = getModel();
        tfvis.show.modelSummary({name: 'Model␣Architecture'}, model);
        await train(model, data);
        init();
        alert("Training␣is␣done,␣try␣classifying␣your␣handwriting!");
}
```

Here's our model, which should be fairly straightforward to understand at this point.

```
function getModel() {
        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']});

        return model;
}
```

The next step in `run()` is to use `tf-viz` to show us the model architecture (a pretty-print of what the layers look like). We then train our model; this is an asynchronous function and we pass it the model as well as our reference to data. We can take a look at how `train()` operates. The first thing it does is set up the `callbacks`. We then specify our data sizes, and we get the next training and test batch from the data, where in particular we're using `tidy()` to avoid having these objects sitting around in memory when we're done computing with them. Since the image was downloaded as part of a sprite, we need to call `reshape` to get it into a $28 \times 28 \times 1$ tensor (recall we're the last dimension describes the monochrome of the image). Lastly, when we want to actually train the model, we call `fit()` where we supply `trainXs` and `trainYs`, as well as supplying our additional validation data and our callbacks.

```
async function train(model, data) {
        const metrics = ['loss', 'val_loss', 'acc', 'val_acc'];
        const container = { name: 'Model Training', styles: { height: '640px' } };
        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: 20,
                shuffle: true,
                callbacks: fitCallbacks
        });
```

Once we're done training, `run()` calls `init()` which sets up our UI. The UI has our canvas, as well as the methods associated with it that allow us to draw with our mouse. There's also the `save` button which does the inferene and the `clear` button which resets our canvas; we simply add an event-listener to each of these.

```
function init() {
        canvas = document.getElementById('canvas');
        rawImage = document.getElementById('canvasimg');
        ctx = canvas.getContext("2d");
        ctx.fillStyle = "black";
        ctx.fillRect(0,0,280,280);
        canvas.addEventListener("mousemove", draw);
        canvas.addEventListener("mousedown", setPosition);
        canvas.addEventListener("mouseenter", setPosition);
        saveButton = document.getElementById('sb');
        saveButton.addEventListener("click", save);
        clearButton = document.getElementById('cb');
        clearButton.addEventListener("click", erase);
}
```

The important function is actually `save()`, which is responsible for our inference. We start by calling `rawPixels()`. This is actually obtained as output from our `draw()` function, which manages all that is needed to track your mouse movement and impress it upon the canvas. The second argument within the call to `tf.browser.fromPixels(rawImage, 1)` signifies we only want one color channel. Since our input is of size 280×280, we need to call `tf.image.resizeBilinear` to transform our data into a 28×28 array that is compatible with our model. The call to `expandDims(0)` ensures that our resized image has an additional dimension corresponding to the color channel: it's a 4-dimensional tensor that we pass in to our model, where the first dimension describes the number of images, the second two are the x's and y's of the images, and the last is the color depth; we add a fourth dimension to our tensor with `expandDims(0)`.

```
function save() {
        var raw = tf.browser.fromPixels(rawImage,1);
        var resized = tf.image.resizeBilinear(raw, [28,28]);
        var tensor = resized.expandDims(0);
    var prediction = model.predict(tensor);
    var pIndex = tf.argMax(prediction, 1).dataSync();

        alert(pIndex);
}

function draw(e) {
        if(e.buttons!=1) return;
        ctx.beginPath();
        ctx.lineWidth = 24;
        ctx.lineCap = 'round';
        ctx.strokeStyle = 'white';
        ctx.moveTo(pos.x, pos.y);
        setPosition(e);
        ctx.lineTo(pos.x, pos.y);
        ctx.stroke();
        rawImage.src = canvas.toDataURL('image/png');
}
```

So we see that it's the `draw()` function which creates our `rawImage` that we then perform inference on.