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.

1.4 Pre-trained TensorFlow.js models

In this section, we'll discuss how to take a TensorFlow model trained in Python and run it in a web browser using TensorFlow.js. Even though we've learned how to train a model in a web browser, today most models are trained in data centers (using TensorFlow). There's a suite of tools in TensorFlow, called TensorFlow.js in Python, that you can use to convert your TensorFlow saved models into JSON notation. In particular, we're going to look at a toxicity model, in which you can pass it some text and it will classify it in terms of various meanings of toxic, e.g. whether it contains threathening language, insults, obscenities, identity-based hate, or sexually explicit language, and another is MobileNet for image classification. MobileNet is a neural network architecture that is designed to run on mobile devices and web browsers and low compute resource environments. We'll see how we can use a model to perform inference locally, which is paramount to keeping user-data private and also saves on round-trip communication costs.

We've seen that in several scenarios, training in the browser requires some kind of compromise. E.g., creating 70k HTTP connections to download 70k images to train fashion-MNIST is not feasible, so we created a spritesheet as a workaround: but then we've got to take into account the memory that's being used when executing on a browser. In this instance, `tf.tidy()` comes to the rescue. This week, we'll see how easy it is to use off the shelf trained models in your browser using TensorFlow.js. We'll also look at some scripts that are used to convert pre-trained models into the format that is usable in JavaScript.

Pre-trained TensorFlow.js models Are available at `https://github.com/tensorflow/tfjs-models`. In particular, we'll first consider `https://github.com/tensorflow/tfjs-models/tree/master/toxicity`.

1.4.1 Toxicity Classifier

We can easily view the entirety of the html page. Notice that we first load in two scripts: (i) the latest version of TensorFlow.js and (ii) the toxicity model. We set our threshold to be 0.9, which will describe a probability label an instance as toxic. We will load the toxicity model, initializing with said threshold, and we will use this model to classify some sentence(s).

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/toxicity"></script>
<script>
const threshold = 0.9;
toxicity.load(threshold).then(model => {
    const sentences = ['you suck'];
    model.classify(sentences).then(predictions => {
        console.log(predictions);
        for(i=0; i<7; i++){
            if(predictions[i].results[0].match){
                console.log(predictions[i].label +
                            "␣was␣found␣with␣probability␣of␣" +
                            predictions[i].results[0].probabilities[1]);
            }
        }
    });
});
</script>
</head>
<body></body>
</html>
```

We can pass our sentences to the model, and as a result we get back a set of predictions; we will log-out the predictions so that we can take a look at them. Our model will output a probability for each of the categories of toxicity.

1.4.2 MobileNet

MobileNet is a small, low-latency, low-power model that is parameterized to meet the resource constraints of a variety of use cases. There are versions that can be built upon for classification, detection, embeddings, and other segmentation similar to how models like Inception operate. MobileNets are trained to recognize a thousand different classes, see a listing here: `http://www.laurencemoroney.com/wp-content/uploads/2019/04/labels.txt`.

Using the pre-trained MobileNet in JavaScript is quite simple. As before, you simply include the script for both TensorFlow.js and for MobileNet.

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0"> </script>
```

We're going to classify the contents of an image, and write out the predictions to the page. So, in the *body* of our page we'll need to place an image tag and a `div` to contain the output of the text.

```
<body>
    <img id="img" src="coffee.jpg"></img>
    <div id="output" style="font-family:courier;font-size:24px;height=300px"></div>
</body>
```

As the code is written, the image should reside in the same working directory as the MobileNet.html file which is where we've pointed our web-server to run this example. We'll next need a *script* to pass an image to the MobileNet model and get a set of predictions back. Because we want the code here to execute after the page has loaded, then at the very least we must place this code at the bottom of the html file after the closing *body* tag. Or, if you're familiar with the DOM model, you can call it when the DOM has finished loading. For now, we keep things simple by placing it below the closing body tag.

```
<script>
    const img = document.getElementById('img');
    const outp = document.getElementById('output');
    mobilenet.load().then(model => {
        model.classify(img).then(predictions => {
            console.log(predictions);
            for(var i = 0; i<predictions.length; i++){
                outp.innerHTML += "<br/>" + predictions[i].className + " : " + predictions[i].probability;
            }
        });
    });
</script>
```

Note that if we look at the results that have been logged to console, only to the top three classes and their associated probabilities have been output. Within the `for`-loop, we are adding a *break* character, the prediction `className`, and the probability for that `className` to the `innerHTML` of the `div`. Let's have a look at the 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-models/mobilenet@1.0.0"> </script>
</head>
<body>
    <img id="img" src="coffee.jpg"></img>
    <div id="output" style="font-family:courier;font-size:24px;height=300px"></div>
</body>
<script>
    const img = document.getElementById('img');
    const outp = document.getElementById('output');
    mobilenet.load().then(model => {
        model.classify(img).then(predictions => {
            console.log(predictions);
            for(var i = 0; i<predictions.length; i++){
                outp.innerHTML += "<br/>" + predictions[i].className + " : " + predictions[i].probability;
            }
        });
    });
</script>
</html>
```

Note that in the head block we see two *script* tags, and these load in TensorFlow.js and the model. Then, in the body of the page we're specifying that we have an image and a `div` with a nice big font: we're going to have "courier font" sized 24-pixel, and we'll make the `div` 300 pixels high. The second script block is placed *beneath* the script body, such that our libraries are loaded before the script executes. In this script, we'll obtain a handle to our image that we loaded, as well as our output. We load our model, and once it's done we can pass an image directly to it; we don't have to worry about any of the underlying implementation details like how to manipulate and transform raw tensors. We loop through our predictions (in this case the model spits out the top three categories of the thousand possible) and we'll create add HTML to the output which will just be a new line followed by the class name of thte prediction followed by the probability of the prediction.

1.5 Converting Models to JSON Format

We've seen how using existing pre-trained models is very easy to do in the browser using JavaScript. In this section, we'll quickly go over how to build and train a neural network for ourselves in Python and then write the output to a format amenable for ingesting in JavaScript. In particular, we'll revisit our network that learns the relationship where $y = 2x - 1$. In order to convert our model, we'll need to install the following module in Python:

```
pip install tensorflowjs
```

We can now create and train our network, in Python.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
print(tf.__version__)
model = tf.keras.Sequential([keras.layers.Dense(units = 1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')
xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
model.fit(xs, ys, epochs=500)
```

If we want to predict a value, we can do so via `print(model.predict([10.0]))` where we expect to realize a value very close to 19. In order to save our model, we're going to start by generating a directory to save our file in, and we do this using a timestamp.

```
import time
saved_model_path = '/tmp/saved_models/{}'.format(int(time.time()))

# For TensorFlow 2.0, use this:
# tf.keras.experimental.export_saved_model(model, saved_model_path)

# For TensorFlow 1.x, use this:
tf.contrib.saved_model.save_keras_model(model, saved_model_path)
```

These codes simply write the model to the specified path on disk. The following command will convert a saved model that was stored in a TensorFlow.js format. From the command line:

```
tensorflowjs_converter \
  --input_format=keras_saved_model \
  /tmp/saved_models/time_stamp_returned/from_prev_ste \
  /tmp/linear
```

where we emphasize that one of the arguments is simply the timestamp named directory that we created a few moments ago. The last argument is the output directory where we want the JSON file to be saved. Let's consider an HTML page with the model hosted in it.

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script>
    async function run(){
        const MODEL_URL = 'http://127.0.0.1:8887/model.json';
        const model = await tf.loadLayersModel(MODEL_URL);
        console.log(model.summary());
        const input = tf.tensor2d([10.0], [1,1]);
        const result = model.predict(input);
        alert(result)
        }
        run();
</script>
</head>
<body></body>
</html>
```

Note that the `run()` function is asynchronous because we have to wait to load the layers model. In general, this is similar to what we've seen before: we first load the script containing TensorFlow.js latest version, and then we write another script to execute code in a block. First, notice that we have a model-URL, which must be loaded over HTTP. While this file resides in the same directory as our HTML, we still use the URL path to it. To translate a JSON file into a model we use `await tf.loadLayersModel()`, passing the relevant URL.

1.6 Re-training MobileNet

This section we'll learn how to build a system where we can use a webcam to capture an image, then train and run a classifier that lets us play a game of rock, paper, scissors. This involves getting JavaScript in the browser, doing training within the browser, transfer learning off an existing model, and even interacting with a webcam. In particular, to execute on transfer learning we're going to freeze some of the layers of a pre-trained network, then train a new neural network using these extracted features from MobileNet.

1.6.1 Building a webpage with a webcam

We want to build a simple web-page that contains a video `div` in which we'll render the webcam. Let's build a page that renders a live-stream of the webcam, and it will also initialize everything we need to start capturing from the webcam and converting the data into tensors which will be used to train the network; the `webcam.js` script has been open-sourced by Google to facilitate capturing images and transforming them into tensors.

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="webcam.js"></script>
</head>
<body>
  <div>
    <div>
      <video autoplay playsinline muted id="wc" width = "224" height = "224"></video>
    </div>
  </div>
</body>
<script src="index.js"></script>
</html>
```

The video emphtag defines a video area on the page that allows us to display what the webcam is seeing. Lastly, the `index.js` is a file where we'll write most of our code: we'll use it to initialize our webcam. By declaring the `MobileNet` and `Model` variables at the top of the script, we allow them to be shared across other functions within the script. In particular, the third line below creates a webcam object (defined in `webcam.js`) which is initialized by pointing to the video element in the hosting webpage that we call `wc` for the webcam.

```
let mobilenet;
let model;
const webcam = new Webcam(document.getElementById('wc'));
```

We're then interested to call the `init()` function, which for now will just set up the webcam, load our MobileNet model by chopping off the last few layers to facilitate transfer learning, and initialize a prediction.

```
async function init(){
        await webcam.setup();
        mobilenet = await loadMobilenet();
        tf.tidy(() => mobilenet.predict(webcam.capture()));

}
```

Our next function of interest is the `loadMobilenet()` that we just saw get called above within `init()`; this function will eventually allow us to re-train the model. As before, we'll load the JSON model from its hosted URL into an object, using `tf.loadLayersModel()` to do so. From here, we can select a particular layer (in this case called "p13 relu") and choose to re-train the layers between the input and this layer. We'll construct a *new* model: it accepts what MobileNet accepts as input (an image tensor), and outputs features in a form specified by the transformations up through the selected output layer.

```
async function loadMobilenet() {
  const mobilenet = await tf.loadLayersModel('https://storage.googleapis.com/tfjs-models/tfjs/mobilenet_v1_0.25_224/model.json');
  const layer = mobilenet.getLayer('conv_pw_13_relu');
  return tf.model({inputs: mobilenet.inputs, outputs: layer.output});
}
```

Let's now consider how we're going to train our model, as transfer learning works a bit differently in TensorFlow.js. Instead of adding new densely connected layers on top of our frozen layers from the original model, we'll create a *new* model with the input shape being the output of the (intermediate layer of) MobileNet; we'll treat this as a separate model that we then train. At prediction time, we first get a prediction from truncated MobileNet which produce a set of embeddings (or features), which we then pass in to our new model to get a prediction. Since we're defining a new model, we'll use the usual syntax of `tf.sequential()`. We set up a single hidden layer with a hundred neurons and then output one of our three specific classes corresponding to rock, paper, scissors.

```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
        }
      }
    });
}
```

Thus, the general flow of our program will be, at a high level:

```
const embeddings = mobilenet.predict(img);
const predictions = model.predict(embeddings);
```

1.6.2 Capturing the Data

In this section, we're going to see how we can capture a sample from the webcam, encode it for training, and then train a new neural network with the transferred features from MobileNet. The first set of changes we incorporate into our HTML file is the addition of three buttons: one for each type of example that we want to capture. Correspondingly, there are three output `div`s which will render the number of samples that have been captured for each type of category we intend to classify. Lastly, there's a button to start the training.

```
<button type="button" id="0" onclick="handleButton(this)" >Rock</button>
<button type="button" id="1" onclick="handleButton(this)" >Paper</button>
<button type="button" id="2" onclick="handleButton(this)" >Scissors</button>
<div id="rocksamples">Rock Samples:</div>
<div id="papersamples">Paper Samples:</div>
<div id="scissorssamples">Scissors Samples:</div>
<button type="button" id="train" onclick="doTraining()" >Train Network</button>
```

Notice that each of the three buttons for gathering data share the same `handleButton` as their `onClick` event-handler; the id's of these buttons are useful in other functions. We also remark that we've named the `div`s for reporting the number of samples of each type. Let's look at how our `onClick` event-handler works for when we are gathering data. Note that when we call `handleButton`, we pass as argument `this`, which is a reference to the button or HTML element, and so in turn we have access to an `id` attribute that distinguishes Rock from Paper from Scissors. We can `switch` on this `id`, where in particular we find the corresponding `div` and update its text to display the appropriate number of samples collected so far.

```
function handleButton(elem){
        switch(elem.id){
                case "0":
                        rockSamples++;
                        document.getElementById("rocksamples").innerText = "Rock␣samples:" + rockSamples;
                        break;
                case "1":
                        paperSamples++;
                        document.getElementById("papersamples").innerText = "Paper␣samples:" + paperSamples;
                        break;
                case "2":
                        scissorsSamples++;
                        document.getElementById("scissorssamples").innerText = "Scissors␣samples:" + scissorsSamples;
                        break;
        }
        label = parseInt(elem.id);
        const img = webcam.capture();
        dataset.addExample(mobilenet.predict(img), label);

}
```

After incrementing the appropriate category, we can extract the `id` attribute and parse it as an integer to construct a `label`. We'll then capture the contents of the webcam. Notice that we are *not* adding the captured image directly to the dataset, but instead we're adding the prediction of the image alongside its associated label; this is because the input to our model is the output of our truncated MobileNet, i.e. some higher-level learned features from MobileNet. We then train our new network on these embeddings instead of the raw webcam data, and this effectively facilitates transfer learning. Lastly, notice that we output the label as a zero, one, or two (and it's not yet one-hot encoded).

The Dataset Class The first step is that in our HTML file, we'll need to create a script tag which loads our dataset class.

```
<script src="rps-dataset.js"></script>
```

Before we can use the dataset class in our `index.js` file, we have to declare it.

```
const dataset = new RPSDataset();
```

We can inspect our `rps dataset` class, which is defined within `rps-dataset.js`. First, note the `labels`: as we add new examples to our training set, we must keep track of their labels and that's what our `labels` array is used for; when we initialize the class we simply set this to an empty array. Next, we have a function `addExample(example, label)`, where the `example` is actually the embedding obtained from truncated MobileNet, and the `label` value is zero, one, or two for rock, paper, or scissors accordingly. Next, we use a little bit of `if-else` logic to handle the accumulation of data.

```
class RPSDataset {
  constructor() {
    this.labels = []
  }

  addExample(example, label) {
    if (this.xs == null) {
      this.xs = tf.keep(example);
      this.labels.push(label);
    } else {
      const oldX = this.xs;
      this.xs = tf.keep(oldX.concat(example, 0));
      this.labels.push(label);
      oldX.dispose();
    }
  }
```

For the first sample, our `xs` are `null`, and so we set the `xs` to `tf.keep()` for the `example` embedding; we also append to our labels array. Note that `tf.keep()` tells TensorFlow that we want to keep this tensor even if a `tidy()` is later called. For all subsequent samples, we simply append the new example to the old. We choose to keep `labels` as integers (zero, one, or two) as it is a more space-efficient representation when compared with a one-hot encoding, which we will save until the last step.

```
  encodeLabels(numClasses) {
    for (var i = 0; i < this.labels.length; i++) {
      if (this.ys == null) {
        this.ys = tf.keep(tf.tidy(
            () => {return tf.oneHot(
                tf.tensor1d([this.labels[i]]).toInt(), numClasses)}));
      } else {
        const y = tf.tidy(
            () => {return tf.oneHot(
                tf.tensor1d([this.labels[i]]).toInt(), numClasses)});
        const oldY = this.ys;
        this.ys = tf.keep(oldY.concat(y, 0));
        oldY.dispose();
        y.dispose();
      }
    }
  }
}
```

When we train, we'll want to rely on `ys`, but so far we've only accumulated `labels`. The above function first sets the `ys` to `null` and then one-hot encodes the results into `dataset.ys`.

Training the Network with Captured Data We're at the point where we can think about training our network. First, we need to one-hot encode the labels in the dataset. We define the layers used in our model, taking care to note that we're starting with inputs as taken from truncated MobileNet embedding space.

```
async function train() {
  dataset.ys = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.ys, {
    epochs: 10,
    callbacks: {
      onBatchEnd: async (batch, logs) => {
        loss = logs.loss.toFixed(5);
        console.log('LOSS: ' + loss);
        }
      }
   });
}
```

1.7 Performing Inference from the Webcam Feed

We've created a page that hosts TensorFlow.js, a downloaded and truncated MobileNet model set up for transfer learning, as well as methods for gathering data which can be used to train a model. In this video, we'll focus on how to pull frames from the webcam and pass them to the model for inference. We'll start by editing our HTML page, adding a `div` with basic instructions, a button to start inference, and another to end inference, as well as a `div` to output the results. Note that we have yet to write the functions that start and stop predicting, but they will end up outputting to the `div` with predictions that we've named "prediction".

```
        <div id="dummy">Once training is complete, click 'Start Predicting' to see predictions, and 'Stop Predicting' to end</div>
        <button type="button" id="startPredicting" onclick="startPredicting()" >Start Predicting</button>
        <button type="button" id="stopPredicting" onclick="stopPredicting()" >Stop Predicting</button>
        <div id="prediction"></div>
```

Next let's look at our `startPredicting()` function, which simply toggles a boolean `isPredicting` and calls the `predict()` function. A very similar design pattern is reflected in `stopPredicting()`.

```
function startPredicting(){
        isPredicting = true;
        predict();
}

function stopPredicting(){
        isPredicting = false;
        predict();
}
```

The general flow of `predict()` is that it first obtains a prediction by reading a frame from the webcam and classifying it, then evaluates the prediction and updates the UI with an "I see ___", and lastly there is some cleanup that is performed. Examine the `predictedClass` function: it reads a frame from the webcam, runs it through truncated MobileNet to obtain an embedding, then feeds this through our model to obtain a prediction; we translate our prediction into a 1D tensor before classifying according to the probability that is largest (this last step of calling `argmax` returns a zero, one, or two). With our classification in hand, we can now update the user-interface to reflect what the model has seen. This process starts with a call to our `predictedClass.data()` where we extract the predicted class id. We then enter a self-explanatory `switch` statement before updating the "prediction" `div` on the webpage.

```
async function predict() {
  while (isPredicting) {
    const predictedClass = tf.tidy(() => {
      const img = webcam.capture();
      const activation = mobilenet.predict(img);
      const predictions = model.predict(activation);
      return predictions.as1D().argMax();
    });
    const classId = (await predictedClass.data())[0];
    var predictionText = "";
    switch(classId){
                case 0:
                        predictionText = "I see Rock";
                        break;
                case 1:
                        predictionText = "I see Paper";
                        break;
                case 2:
                        predictionText = "I see Scissors";
                        break;
        }
        document.getElementById("prediction").innerText = predictionText;


    predictedClass.dispose();
    await tf.nextFrame();
  }
}
```

It's in the call to `predictClass.dispose()` that we actually trigger the `tf.tidy()` we saw previously. We also call `tf.nextFrame()` which is a TensorFlow function that prevents us from locking up the UI on the web browser.

1.8 Concluding remarks on browser based models with TensorFlow.js

We've seen how to use TensorFlow.js to deploy and train in a web browser. But one of the most important deployment scenarios is actually on mobile devices such as iOS or Android. For example, it's possible to take a phone that's disconnected from the internet, have it take a picture of a plant and perform inference to tell if the plant is diseased. Some of these deployment scenarios are made possible with TensorFlow Lite.

2 Device-based Models with TensorFlow Lite

We've seen how to build learning algorithms that run on a laptop or the cloud, but there's something even more magical about getting them to work in your hand, perhaps on your smartphone or even a lightweight embedded processor like an Arduino Raspberry Pi which can be bought for some tens of dollars. We can easily imagine image classification or object detection on a mobile phone, but it can be even more exciting to think about smart embedded systems that respond to visual or audio input. Laurence gives an example of a miniature robotic car that he built, which uses a Raspberry Pi running TensorFlow lite in order to learn how to drive: as one manually drives the car around a course, the system captures images and annotates them with device from the controller such as how much acceleration, braking, or turning was being applied...these then become labeled observations for training.

2.1 Machine Learning Models in Mobile and Embedded Systems

Here, we learn what it means to bring a machine learned model to a mobile device, including how to shrink the model to fit on the device, as well as the architecture of TensorFlow Lite. When we want to make our models available on resource constrained devices, it takes additional consideration.

2.1.1 Features and Components of Mobile AI

The goal a mobile AI model is to be lightweight and operate on small, low-power devices such as phones; they may not be as accurate as those which run on clusters in the cloud. TensorFlow Lite is designed to run on devices with low latency and without the need for an internet connection: this prevents the need for a round-trip communication cost. By utilizing on-device ML to operate, TensorFlow Lite also protects user privacy and reduces power consumption as we don't require a power-hungry network connection. Formally, TensorFlow Lite consists of two components: a *converter* and an *interpreter*. The Converter transforms TensorFlow models into a form that is efficient for reading by the interpreter. It also introduces some optimizations to reduce the model size and make the model more performant from an efficiency point of view. On the other hand, the interpreter is what runs on the mobile device and deals with inference of converted models; critically, the interpreter is responsible for executing models and client applications using a reduced set of TensorFlow's operators. It utilizes a custom memory allocator which is less dynamic, and this ensures minimal load and execution latency.

2.1.2 Architecture and Performance

After training a model, it has to go through a conversion before it can be used on a device; typically, models are saved using some common format and we then apply a TensorFlow Lite converter tool to flatten the model whereby it is prepped for mobile or embedded devices. We'll learn about this process in detail in this section. Another interesting point of discussion is how to perform inference of a compute-heavy machine learning model on a mobile device, since this is an inherently resource intensive task relative to the device which has limited processing and power. Inference must be performed rapidly and without overhead to enable realtime applications; to enable this, TensorFlow Lit can employ hardware acceleration libraries or APIs for select devices. For example, Android devices have a neural network API. Secondly, inference can be boosted with Edge TPU's which are solely built for operating on deep learning models (they are performant and have a lower power footprint whilst also being small in size). Another form of acceleration in TensorFlow Lite is what's known as a delegate: a way to pass your graph execution to hardware that's specialized to perform inference. E.g. there is a GPU delegate that can be used to accelerate models on devices which have an available GPU. GPU's have a Single Instruction Multiple Data architecture, meaning they can run many mathematical operations in parallel which makes them amenable toward deep learning inference.

Under the Hood: What Happens When Converting a Model? First, a canonical representation of the network is built, and then this undergoes a series of transformations such as removing unneccessary operations or substituting operations with ones that have a faster implementation. Then, a *shader* is generated and compiled based on the optimized graph using a Shader runtime; for Android this runtime is the opengl es, and for iOS they use Metal. Here's a video on TensorFlow Lite GPU Delegates.

2.1.3 Optimization Techniques

Optimization is necessary because of the generally limited resources on mobile and embedded devices. It's critical that deployed machine learning models have optimal model size, low latency, and low power consumption. It's even more important on edge devices where resources are further constrained and model evice and efficiency of computation can become a major concern. Some known methods are *quantization* (which reduces the precision of the numbers in the weights and biases of the model), *weight pruning* (which reduces the overal number of parameters), and *model topology transforms* (whose goal is to convert to a more efficient model architecture). We'll mostly focus on quantization, which can give the biggest and most immediate gains.

Quantization The basic idea is to optimize your model with reduced precision representations of weights, and optionally activations for both storage and computation. One immediate benefit is that this optimization is available across all types of CPU platforms. We've also reduced the cost of memory access and the cost of storing intermediate activations. Note that there are some hardware accelerators that support fixed point accelerators but which do not support floating point operations. So it turns out that it's possible to *either* start out with a model that's already optimized (and we'll learn about these in this course section), *or* we can convert using TF Lite to optimize them for mobile performance.

2.1.4 Saved Model Format

The process that we go through looks like this.

- We first train a model using TensorFlow.
- We save the model as a `SavedModel`.
- We then use the TFLite converter to produce a TFLite model.

When we create a model using TensorFlow via the Keras API (or even with a low level API), we will save out the model as a Keras model, a `SavedModel`, or a set of concrete in-memory functions. The converter takes these and converts to the TFLite format which is a flat buffer that can be used on the mobile device, along with optional back-ends like neural network APIs or GPUs. In Python, we can call `tf.lite.TFLiteConverter()` to do the conversion.

SavedModel The preferred standard methodology for saving models in TensorFlow 2.0 is to use the `SavedModel` format. It's been designed as a universal, language neutral recoverable way of serializing a TensorFlow model. With this format, we can easily deploy within TFLite, TF.js, TF Serving, or TF Hub. Even model versioning is supported. The format relies on a *metagraph* which is essentially a data-flow graph comprising of any variables associated with the model, any additional assets such as a list of labels or classes that may be required, along with signatures for prediction (for example, the classification signature in the case of image classification). To understand the interfaces of signatures of a `SavedModel`, we can call a commnand line tool to get details about it:

```
saved_model_cli show --dir /tmp/mobilenet/1\
                     --tag_set serve \
                     --signature_dev serving_default
```

This result will tell us what the expected dimensions of the inputs and outputs of the neural network are. To export a `SavedModel` built with Keras, we simply use

```
pretrained_model = tf.keras.applications.MobileNet()
tf.saved_model.save(pretrained_model, '/tmp/saved_model/1/')
```

In this case the number "1" in the save-path indicates a version number, and allows TensorFlow Serving to select the latest version of a model within a directory.

Example: single hidden layer with a single hidden neuron Let's create the simplest possible neural network.

```
import tensorflow as tf

# Store the data for x's and y's, where y = 2x-1.
x = [-1, 0, 1, 2, 3, 4]
y = [-3, -1, 1, 3, 5, 7]
```

```python
# Create a simple Keras model.
model = tf.keras.models.Sequential(
    [tf.keras.layers.Dense(units = 1, input_shape=[1])])
model.compile(optimizer = 'sgd', loss = 'mean_squared_error')
model.fit(x, y, epochs = 500)
```

Once we have a model, there's three steps to go through: (i) export the `SavedModel`, (ii) convert the model, and (iii) save the model.

```python
import pathlib

# Export the SavedModel.
export_dir = '/tmp/saved_model'
tf.saved_model.save(model, export_dir)

# Convert the model.
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()  # Produces a flattened version of the model.

# Save the model.
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Using a pre-existing model The process is very similar.

```python
import tensorflow as tf
import pathlib

# Load the MobileNet tf.keras model.
model = tf.keras.applications.MobileNetV2(weights='imagenet',
                                          input_shape=(224,224,3))
# Saving the model for later use by tflite_convert.
model.save('model.h5')

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Command Line Interface There is even a command line interface in the event that you have the model sitting on disk.

```
# Saving with the command line from a SavedModel.
tflite_convert --output_file=model.tflite --saved_model_dir=/tmp/saved_model

# Saving with the command line from a Keras model.
tflite_convert --output_file=model.tflite --keras_model_file=model.h5
```

Quantization One simple method to achieve this is post-training quantization. I.e. rather than quantizing a model during training and effectively having to change your training code, you simple quantize as part of the process of converting the model to the TFLite format. The simplest example could be converting all floats in the weights of the model to integers. In practice this can result in a 3x decrease in latency with only a relatively minor degredation in model accuracy. Let's look at an example where we over-ride the converter to optimize primarily for size.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimization = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_quant_model = converter.convert()
```

We could have also specified that we wish to prioritize for latency or just leave the setting at its default in which case it balances priority for both small size and low latency. Note that in performing post-training quantization, we can make our models up to four times smaller It's also possible to further optimize by using calibration data whereby we first run inference on a small set of examples and then determine the appropriate scaling parameters to use when converting the model for integer quantization. How might that work? We first define a generator which is designed to take samples from our training dataset. We then se tour optimizer to balance for both size and latency. Finally, we pass our generator to the converter as a representative dataset: this means it's used for evaluating optimizations (i.e. by recording dynamic ranges). In particular, we can record multiple inferences on a floating point TF Lite model using the user-provided representative dataset as an input. We can then use the values logged from inferences to determine the scaling parameters needed to execute all tensors of the model in integer arithmetic. This allows us to quantize the activations along with the weights, and the resulting model will have as many quantized operations as possible.

```
# Define a generator.
def generator():
  data = tfds.load(...)
  for _ in range(num_calibration_steps):
    image, = data.take(1)
    yield [image]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

# Set the optimization mode.
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Pass the representative dataset to the converter.
converter.representative_dataset = tf.lite.RepresentativeDataset(generator)
```

For ops that don't have a quantized implementation, they'll fall back to floating point operations.

TF-Select Note that conversions can sometimes be difficult in the sense that we may stumble across TensorFlow operations that are unsupported by our TFLite converter. Since the set of TFLite operations is smaller than the set of TensorFlow regular C++ operations, not ever model is convertible. Even for supported operations, very specific usage patterns are sometimes expected or anticipated for performance reasons. For a full listing of supported operations, see `https://www.tensorflow.org/lite/guide/ops_compatibility`. Using `TF-Select`, we can use a subset of the TensorFlow operations when the built-in ones from TFLite do not suffice, keeping in mind that this will increase our model size marginally. Example usage:

```
import tensorflow as tf
```

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.target_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
                        tf.lite.OpsSet.SELECT_TF_OPS]
tflite_model = converter.convert()
```

Note the only difference here is that we specify `target_ops` to also include a set of TensorFlow `select` operations. For more on TF-Select, see `https://www.tensorflow.org/lite/guide/ops_select`.

Paths in Optimization If you don't intend to quantize your model, you'll of course end up with a floating-point model. However, if you do wish to quantize just the weights you can do so by setting one of the post-training optimization modes in the converter. Although the converter will do its best to quantize all the ops, your model may still end up with a few floating point operations; the optimization provides latencies close to fully fixed point inference. We can get further latency improvements by quantizing both the weights and the activations with a representative dataset. If you wish to have a model that outputs only integers, a quantization target specification can be set in the converter to generate a model with quantized operations. Note that it's possible to test a model using Python before deploying it on an embedded system. To do this:

```
# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Point the data to be used for testing and run the interpreter.
interpreter.set_tensor(input_details[0]['index'].input_data)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```