16

STEPPING INTO THE FUTURE

OVERVIEW

By the end of this chapter, you will be able to list some high-level machine learning concepts; explain what **Tensorflow.js** is; integrate TensorFlow.js into your JavaScript application; use a pretrained machine learning model to classify images from the web; enhance an existing model with new training data and train a machine learning model from scratch.

In this chapter, we will learn how to make use of TensorFlow.js, a JavaScript library for defining, training, and running machine learning models in a web browser. There is a focus on using pre-existing models in the browser as using the browser is generally computer-intensive and takes a long time. JavaScript and the browser lend themselves to interactivity, which means it is the perfect place to display the power of machine learning. We can do things such as access the user's webcam and photos, and have them interact with buttons in a way most people are used to.

INTRODUCTION

So far in this course, you've learned a bit about JavaScript's history and a lot about modern JavaScript usage. From traditional browser environments to server-side implementations and mobile frameworks, you've also seen that JavaScript can even be used for desktop applications and **Internet of Things** (**IOT**) devices. JavaScript is truly a diverse language, and recent years have seen it find a place in machine learning. In this chapter, we'll look at the most mature and feature-rich machine learning framework currently available for JavaScript: Tensorflow.js. First off, let's make sure we're all on the same page about what machine learning is and how it relates to **artificial intelligence** (**AI**), and then we'll take a look at what makes Tensorflow.js so exciting.

AI is an umbrella term that's used to describe a computer system that behaves in a more human way than traditional computers. AI has many applications; from driverless cars to astronomy, from stock markets to healthcare, and countless others, and doubtless many applications that are yet to be thought of.

MACHINE LEARNING

Machine learning algorithms are a subset of AI. Machine learning systems adapt to data inputs without explicit instruction from a programmer. They can work with pretty much any kind of data, with different types of machine learning being used for different kinds of data. The kind of data dictates the kind of problem or problems that the machine learning model will need to solve in order to correctly classify the data or predict the result. At the highest level, machine learning algorithms are taught in one of three main ways: supervised learning, unsupervised learning, or reinforcement learning:

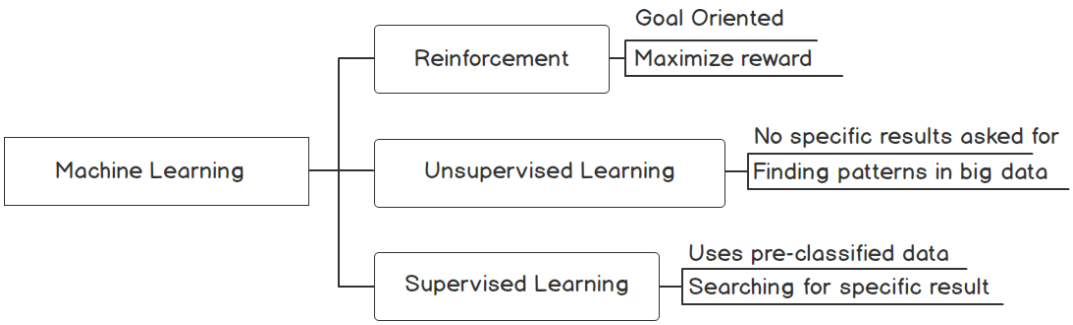

Figure 16.01: Machine learning and its branches

Reinforcement learning

Reinforcement learning is where an algorithm is given a goal as a way of measuring performance, and it learns to maximize success through trial and error, without being given any training data containing "success" targets. Reinforcement learning algorithms become better and better at a task over time and have been used to play games expertly, to provide personalized recommendations, and widely in robotics.

Unsupervised learning

Unsupervised learning describes machine learning models where no pretraining or specific results are provided. Unsupervised learning systems are typically useful for finding patterns in large sets of data that would otherwise be difficult for humans to identify. This training style is useful for applications where we wish to identify links between genes and biological traits.

Supervised learning

Supervised learning describes a machine learning system that has been trained on a set of data with defined target results. In other words, the algorithm is given existing data, along with the result or class of that data, and it will learn to predict the result or class of future data. They're used for things such as classifying images, predicting next week's weather, and natural language processing.

In this chapter, we'll be focusing on supervised learning. We'll look at a few types of supervised learning models and algorithms, including MobileNet, a **k-Nearest Neighbors** (**KNN**) algorithm, and a **Common Objects in Context Single Shot Multibox Detector** (**COCO-SSD**). We'll also look at how to import supervised models, train them in the browser, and use them in context.

What is TensorFlow.js?

In this chapter, we will be looking at supervised learning models that are implemented with TensorFlow.js. TensorFlow is an open source machine learning library that was created by *Google's Google Brain team*. With it, you can create and train your own machine learning models, import existing models that you can use in your applications, and build upon existing models by adding additional training data to them. TensorFlow gives developers very easy access to machine learning, and it's a highly capable machine learning library, so experienced ML developers aren't hampered by a lack of functionality.

Before we get started with TensorFlow.js, let's look at a visualization of a neural network and talk about its components:

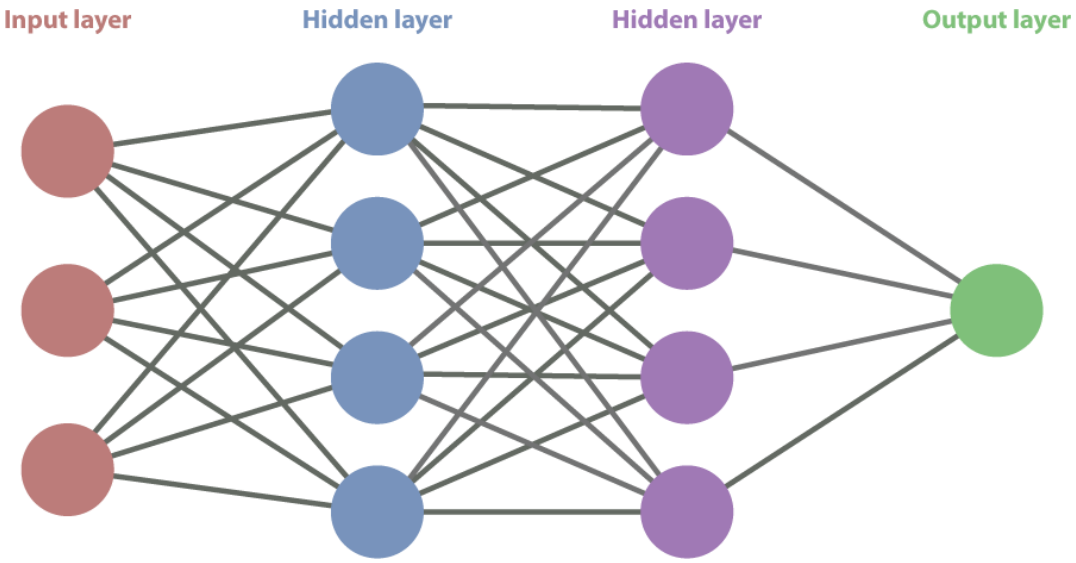

Figure 16.02: Layers and nodes of a neural network

A neural network is made up of multiple layers: an input layer, which receives whatever data we're working with; one or more hidden layers, which process the data through various nodes (or neurons); and an output layer, which produces the final output of the algorithm. The output from each layer becomes the input for the next. Each node assigns a weight to its input data and processes the data through an activation function.

The weighting either increases or decreases that node's significance in respect of the entire network, and over time, the system can determine which weightings improve the algorithm's accuracy and which are detrimental.

With supervised machine learning, we can train a model by passing it a set of data inputs and outputs and having it learn the relationships between the inputs and outputs. Then, a well-trained model will be able to take a new, unseen input, pass it through the set of rules it established during training, and return an accurate prediction as to what the output should be.

Here's a simple example of the kind of training we can do, and how we would organize the data for the model:

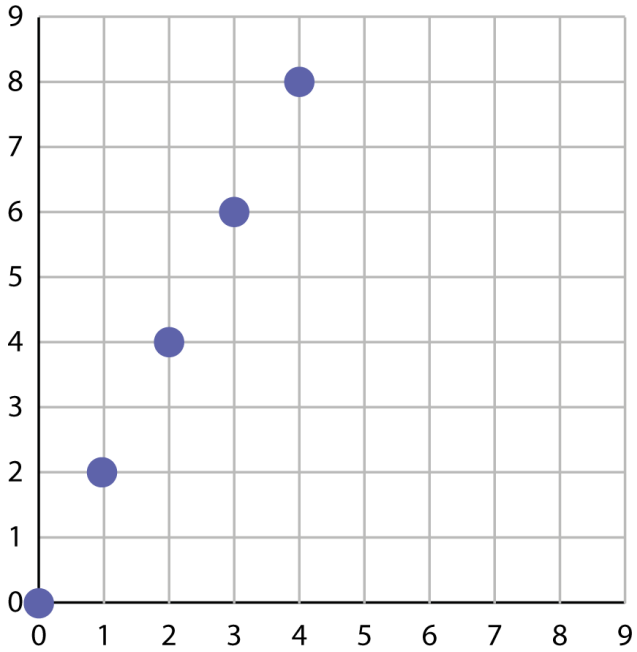

Figure 16.03: Example training data

To our eye, it's obvious that these points form a line. If we want to predict Y's value when X is 5, we can infer what it'll be. A neural network can also make this inference (albeit through a different process to our own – it doesn't see the line). We can train it on the data we have to allow it to make this prediction. We would pass the training data as a list of *X*s and *Y*s, or inputs and outputs, like so:

```
let coordinates = [
  [0,0],
  [1,2],
  [2,4],
  [3,6],
  [4,8]
]
```

The preceding two-dimensional array may look like the points on a line – and in a sense, it is – but it is also a mapping between an input and an output; that is, if our *X* value is 2 our *Y* value is 4, and if our *X* value is 4 our *Y* value is 8. After training on these *X*s and *Y*s, the model would be able to predict *Y* when *X* is 5. You'll often hear *X*s and *Y*s used as shorthand for inputs and outputs, even when the data structure is more complex.

CONVOLUTIONAL NEURAL NETWORKS

A **convolutional neural network** (**CNN**) is a deep learning neural network commonly used for processing image and video data. It works by taking a set of input data and processing it through multiple layers. The layers between the input and output are referred to as "hidden layers" since the state of the data at each of these layers is unavailable to the data scientist or developer.

A somewhat simplified way of considering the function of having multiple hidden layers is to imagine that each layer does some specific analysis for a larger process, for example, a network in which an image is processed to see whether it is a bus. The first layer may check to see whether the image has any wheels, then the second may check whether the body on top of the wheels is box-shaped, and the third layer may detect the size of the box to see whether it is too large or too small to be a bus.

Of course, the preceding explanation is a simplified one. It is important to remember that, as developers or data scientists, we don't design what each layer in a neural network does. Furthermore, we don't necessarily know what each layer is doing. The bus example is very unlikely to be accurate, but it gets across the idea that each layer can specialize in some aspect of a larger analysis process. It is likely that the first few hidden layers in a neural network determine larger shapes or edges that are present in an image. By identifying shapes and edges, the next layers in the network can then use those identified shapes to predict the object in the image.

MOBILENET

A **MobileNet** is a lightweight CNN designed to run in situations where computational power and memory are limited, for example, on an embedded device, mobile phone, or web browser. Despite their low resource footprints, MobileNets can produce surprisingly accurate results. Making use of the MobileNet built-in with TensorFlow.js requires three steps. First, the MobileNet script must be included in your node:

```
<script src="https://unpkg.com/@tensorflow-models/mobilenet"></script>
```

Secondly, you need to load the model. You can do so using **`await`** so that your code isn't blocked:

```
let model = await mobilenet.load();
```

Finally, simply pass an HTML image object to the **`classify`** method on the MobileNet instance:

```
const results = await model.classify(image);
```

EXERCISE 16.01: CREATING AN IMAGE CLASSIFICATION APP USING TENSORFLOW

We talked about the power of a lightweight CNN known as MobileNet. Now, we're going to make use of it in a small application. We'll create an image classification app that will attempt to classify images we feed into it.

We will create a simple HTML file with an image and then write a short script to run the MobileNet's **`classify`** method. By the end of this exercise, you'll be able to make use of this general-purpose image classifier in your own projects.

We'll start off by creating an HTML file called **`index.html`**:

1. In our **`<head>`** tag, we are importing our dependencies: these are the TensorFlow.js CDN and the MobileNet model:

```
<html>
  <head>
    <script src="https://unpkg.com/@tensorflow/tfjs"></script>
    <script src="https://unpkg.com/@tensorflow-models/mobilenet"></
script>
  </head>
```

2. In our **`<body>`** tag, we have a **`<h1>`** element with an ID of **`"status"`**, an **`<img>`** element with an **`id`** of **`"image"`**, and a reference to our JavaScript file, which we'll call **`tf.js`** (you can call it something else if you're feeling adventurous). The image you put as the source is up to you. Obviously, the more common and clear the image is, the better the detection results will be. The image we're using has been included in the associated exercise file:

```
  <body>
      <h1 id="status"></h1>
      <br/>
      <img id="image" src="images/cat1.jpg" height="800px"/>
```

```
        <script src="tf1.js"></script>
      </body>
    </html>
```

3. Next, we'll write out our JavaScript file, **`tf.js`**. First, we get references to our image and status elements:

```
// tf.js
const image = document.getElementById('image');
const statusElem = document.getElementById('status');
```

4. Then, we write an asynchronous function to attempt to classify our image. The function sets the **`innerText`** of our **`status`** element to **`'Loading MobileNet...'`**:

```
async function classifyImage() {
  statusElem.innerText = 'Loading MobileNet...'
```

5. We then load the model by calling the **`load()`** method of the **`MobileNet`** object that we have access to through the imported script in our HTML file. We call this method using the **`await`** keyword since the load method returns a promise, and we assign the resolved (or rejected) promise to a variable called model.

```
  let model = await mobilenet.load();
```

6. When the model has finished loading, we call the **`classify()`** method of the model, passing in the image in our HTML file. TensorFlow will process the image using the MobileNet model and attempt to classify the image. It will return classification results as an array, which we assign to our **`results`** variable. This method also returns a promise, so we are again using the **`await`** keyword to wait for the promise's resolution (or rejection):

```
  const results = await model.classify(image);
```

7. We can then check whether the **`results`** variable has a 0th element, and if so, assign its **`className`** and **`probability`** properties to our **`status`** element. The **`className`** property is a descriptor of what the model thinks the image is of, and the probability represents how confident it is about the match. The closer the score is to **`1`**, the higher the confidence:

```
  if (results[0]) statusElem.innerText = `${results[0].className} -
${results[0].probability}`
}
```

8. Finally, we call the **`classifyImage`** function:

```
classifyImage();
```

9. We can now load the HTML file and see what MobileNet makes of the image you chose. It'll take a few seconds to load the model, after which we should see some results. How did it do? Did it correctly classify the image? Bear in mind that we are currently only accessing the 0th result of the array, and the algorithm will have made several predictions about what the image is of. Only the prediction with the highest probability will be displayed in the status. Hopefully, you saw some success. If you used the image from the preceding example, you should have seen the following output:

Figure 16.04: Loading the TensorFlow model

The image after being predicted will appear as follows:

Figure 16.05: MobileNet's highest prediction

So that's it; that is all that's required to implement object detection via machine learning in a JavaScript application. Very easy, and very cool. We've used the MobileNet model, which comes built into the TensorFlow.js library to classify a cat image with a certainty of *31.8%*. It was as simple as using **`mobilenet.load`** to load the model into the browser and then running **`model.classify`** to analyze our image.

We've included some additional images in the GitHub repository to demonstrate some of the limitations you'll face. You can try changing the image source from the preceding example code to try them out. With **`cat2.jpg`**, for example, the classifier sees the box that the cat is in rather than the **`cat`** itself. Then, with **`cat3.jpg`**, we've purposely used an image where the cat is hard to see. Once you've tried all three, find some more images from around the internet and experiment to see the results.

In the following image, you will find an unclear image of a fox:

Figure 16.06: MobileNet's highest prediction of an unclear image

As you can see, with an unclear image, the result isn't as good, and what is more worrying is that the certainty is quite high for an incorrect answer. You'll find that, for MobileNet, you'll want to use clear images to get the best results. Analyzing more difficult images, such as the preceding one, requires a more specialized model that has undergone extensive training for edge cases.

THE KNN ALGORITHM

KNN is a machine learning algorithm for finding the nearest neighbor nodes to a node. It stands for k-nearest neighbors, where "k" is an input variable specifying the number of neighbors to be found. So, if you wanted to find the two nearest neighbor nodes, it would be *k=2*.

It's important to note that "nearest" in this context does not necessarily mean in terms of physical distance – it means the closest matches based on the provided data points. For instance, if you were evaluating users on a music streaming site to find a favorite music genre (based on song history) with options such as "rap," "rock," and "classical," we may run KNN with a *k* of 5 to find the five most similar users and return their favorite music genre. If the result was four neighbors with a preference for "rock" and one with a preference for "rap," we would determine that the users' favorite genre is rock music.

PROCESSING LIVE VIDEO WITH TENSORFLOW

We've seen how easy it is to work with images using the pre-built MobileNet that comes with TensorFlow.js. We also have the ability to work with video streams. This has many interesting applications when used with the browser's capability to access a user's webcam both on desktop and mobile.

Tapping into a user's webcam is quite simple. First off, create a **`video`** element with HTML:

```
<video autoplayplaysinline muted id="video" width="896" height="670"></
video>
```

Once you have a **`video`** element, connect it to the user's input:

```
const stream = await navigator.mediaDevices.getUserMedia({video: true});
document.getElementById('video').srcObject = stream;
```

Keep in mind that this assumes that the browser has access to the **`navigator`** object, which isn't always the case. So, you should be sure to wrap your use of the preceding code snippet with something like **`if(navigator)`** and create some fallback behavior for browsers that don't have access to a media device.

Running the preceding code on most browsers should cause a permissions dialog to pop up, like the one shown in the following screenshot:

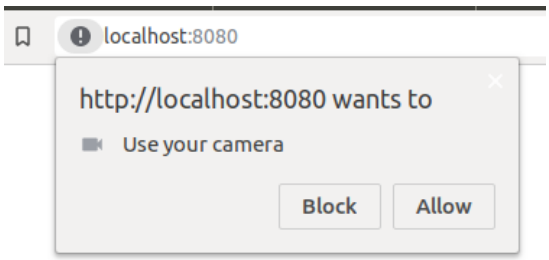

Figure 16.07: Permissions dialog box for giving access to the video element

Once you've enabled it, you should see video being streamed. If not, try refreshing the page so that it starts up with the correct permissions. You should see a live feed of your video, as shown here:

Figure 16.08: Live video stream using TensorFlow

To feed the video stream content into TensorFlow, you'll need to run **`classify`** on the **`video`** element and save the results to a variable. When this function is run, you'll actually get the results of a single frame:

```
let results = await model.classify(videoElem);
```

It can be run in a continuous loop so that each time a frame has been classified and another frame has been processed, it will create the effect of video processing by successively classifying frame by frame:

```
 while(true) {
  const results = await model.classify(videoElem);
  console.log(results);
  await tf.nextFrame();
}
```

`await tf.nextFrame()` simply waits for the next frame to load and blocks the loop from running again if frames are being processed faster than they are produced. If you're finding that the loop is causing your computer to lag, you can always reduce the number of calls by running **`model.classify`** every few seconds instead with a **`setTimeout`** function.

EXERCISE 16.02: CREATING A WEBCAM IMAGE CLASSIFICATION APP USING TENSORFLOW

Let's dig deeper into processing video with TensorFlow.js. In this exercise, we're going to take what we've learned so far and display the results from classifying the frames of a video on a website. We'll use the **`infinite`** loop method and have our website update as new data comes in. Let's plug in our webcam and feed that into the model. Let's get started:

1. The HTML file will be almost the same as in the previous exercise – the only difference is that we can remove our **`image`** tag and replace it with a **`<video>`** tag, which will house our webcam stream. Your HTML file should look like this:

```
<html>
  <head>
    <script src="https://unpkg.com/@tensorflow/tfjs"></script>
    <script src="https://unpkg.com/@tensorflow-models/mobilenet"></
script>
  </head>
  <body>
    <p id="status"></p>
    <video autoplayplaysinline muted id="video" width="896"
height="670"></video>
    <script src="tf.js"></script>
  </body>
</html>
```

2. Now, for our JavaScript file, we can build on the code from the previous exercise. This time, we need to initialize the webcam and pass the video stream data from that to our TensorFlow model. The first part of our JavaScript file will be the same as in the previous exercise: getting references to the video and status HTML elements. Then, we'll call **`initWebcam()`**, a function that we'll define further down in the file:

```
// tf.js
const videoElem = document.getElementById('video');
const statusElem = document.getElementById('status');

initWebcam();
```

3. Next, we'll define a function to classify images that set the status and load the MobileNet model:

```
async function classifyImage() {
  statusElem.innerText = 'Loading MobileNet...'
  let model = await mobilenet.load();
```

4. Again, we will start an infinite **`while`** loop. Inside the loop, we pass the webcam's video stream data to the **`model.classify()`** method and assign our results to the **`results`** variable:

```
  while (true) {
    const results = await model.classify(videoElem);
```

5. Next, we check to see if there is at least one result. If there is, we set **`innerText`** of the status element to the first result's class and score values:

```
    if (results.length) {
      let result = results[0];
      statusElem.innerText = `${result.className} - ${result.
probability}`;
    }
```

6. Then, we use TensorFlow's **`nextFrame()`** method, which helps avoid blocking the browser's execution thread (it returns a promise that resolves after calling the browser's **`requestAnimationFrame()`** method):

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

7. Now, we need to define the **`initWebcam`** function, which is called in the previous function. It checks to see whether the **`mediaDevices`** property is available in our browser's **`navigator`**, and then whether the **`getUserMedia`** method is supported:

```
async function initWebcam() {
  if ('mediaDevices' in navigator && 'getUserMedia' in navigator.
mediaDevices) {
```

8. If **`getUserMedia`** is supported, we use a try...catch statement to request access to the webcam. We use the **`await`** keyword to call the **`getUserMedia`** method to request a video stream, and then set the steam to our **`video`** element's **`srcObject`** property:

```
    try {
      const stream = await navigator.mediaDevices.
getUserMedia({video:
true});
      console.log(stream);
      videoElem.srcObject = stream;
```

9. If there's an error, for example, if the user doesn't grant access to the webcam, then the **`catch`** statement handles the error:

```
    } catch (error) {
      console.log(`Error getting video: ${error}`);
    }
  }
}
```

10. Finally, we call the **`classifyImage`** function to begin the process. We'll wrap it in an event listener that will fire once the **`video`** element we'll be classifying has loaded data from the user's webcam:

```
videoElem.onloadeddata = function() {
  classifyImage();
}
```

11. Now, go ahead and reload the HTML page. You'll likely be asked for permission to use the machine's webcam. Allow this. You'll see the view from your webcam, along with a continuously updating result showing what the model predicts is in the webcam stream – very cool:

Figure 16.09: Webcam image classification with highest probability

The results aren't always accurate, but you'll often get good results for clearly displayed items. For example, I was able to get a *70%* probability prediction that my watch was an analog clock.

12. Let's make one more change to the app that will make it a lot better. Instead of just printing the result of the highest-scoring object, we'll print results for any objects that score above **`0.3`**. The change we need to make is in the **`if`** statement, inside the **`classifyImage()`** function. We'll simply iterate through the results and conditionally add any that are above our score threshold of **`0.3`**:

```
if (results.length) {
  let status = ''
```

```
results.forEach(result => {
    if (result.probability> 0) status += `${result.className} -
${result.probability} \n`
  })
statusElem.innerText = status;
}
```

After updating the code, I tried using a plastic water bottle I had lying around. This gave me the highest probability prediction of all the items I tried, at over *96%* probability. The next highest results were less than *1%* each and not even close. It's hard to say whether they were guessing based on the bottle or my body in the background:

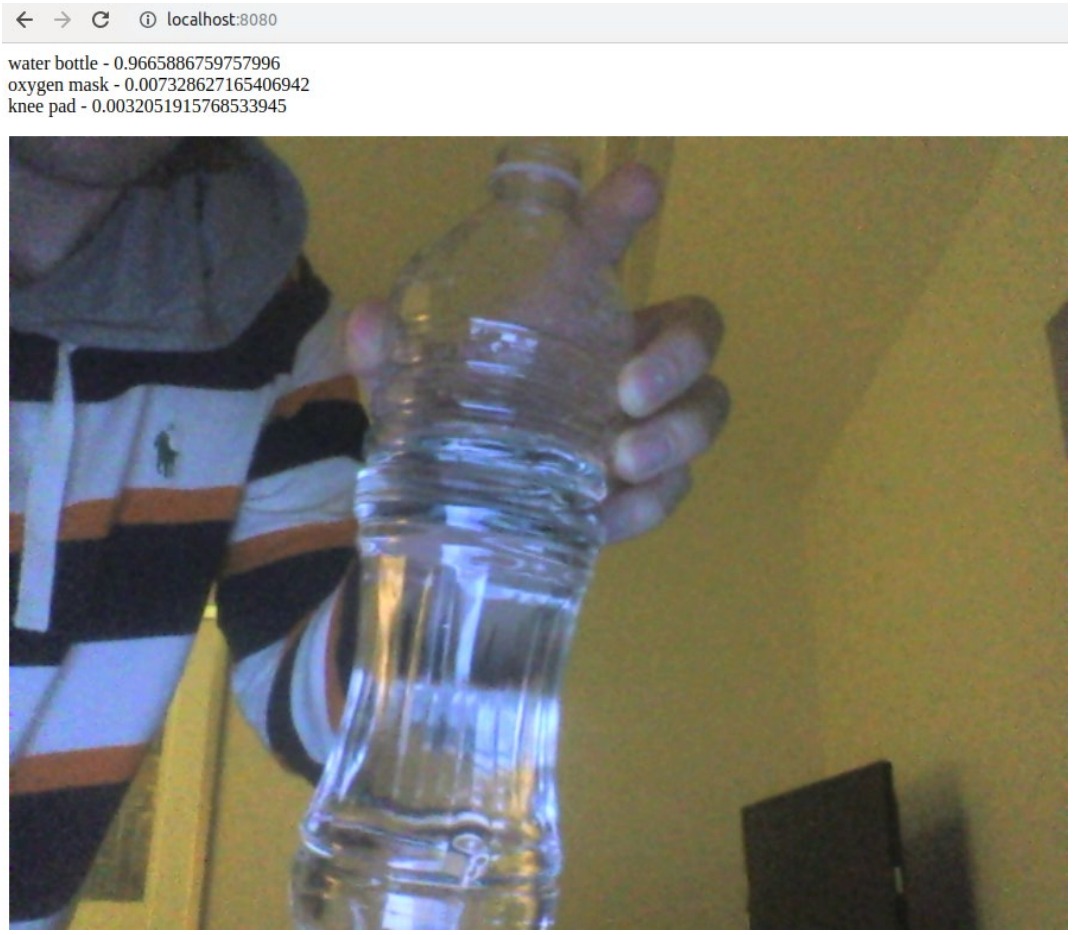

Figure 16.10: Multiple predictions of webcam image using TensorFlow

With the watch image from the first example, I got back **`analog clock`**, **`wall clock`**, and **`barometer`**, which all seemed relatively close. That being said, the watch and water bottle were some of the best examples I was able to find.

Other objects, such as a pair of glasses, didn't seem to work as well when I held them up to the webcam. Some objects would quickly change between possible objects, while others would consistently give the wrong answer. For example, take a look at this coffee cup, which was thought to be a bottle of hair spray:

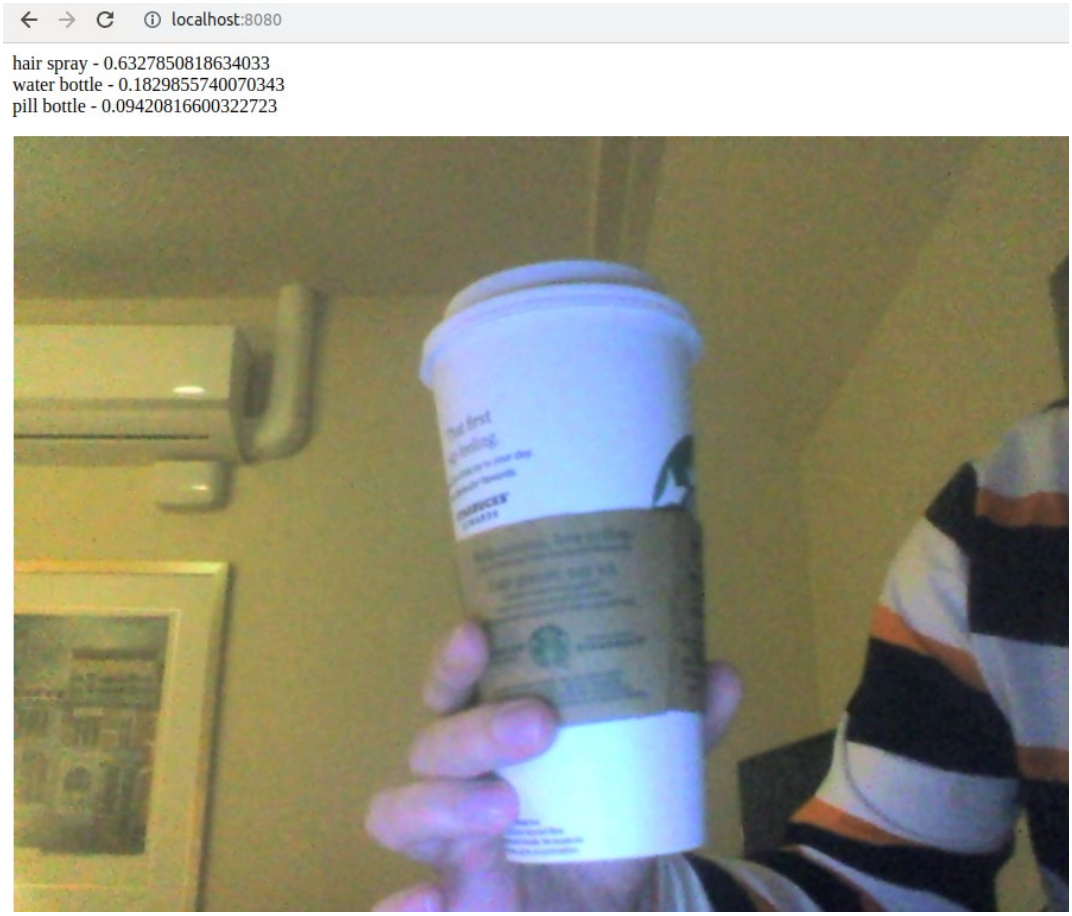

Figure 16.11: Incorrect predictions using webcam images with TensorFlow

Take some time to experiment with this implementation to see which objects are classified accurately and which aren't.

In its current state, the application leaves a lot to be desired; the predictions' **`classNames`** and **`probability`** properties tend to change frequently. We'll come back to this later in this chapter and make the application much more useful and interesting. This was just a quick demo to show how easy it is to incorporate machine learning into our apps.

TRANSFER LEARNING

As well as using pretrained machine learning models as they are, we can also add additional training to them so that they perform better for specific tasks. For example, while a particular image classification model may be good for classifying a wide range of general image classes, it may not be very good at classifying more specific or specialist classes, or correctly differentiating between sub-categories of class; for example, it may detect "dog" well, but not know the difference between "Husky" and "Dalmatian."

We can add additional training data to a model and associate that data with new classes so that the features layer of the model is able to interpret the features of new images as the new classes we also provide. This technique is known as **transfer learning**, and it's a quick and effective way of extending a model's abilities to better meet particular needs. It's much less resource-intensive than training a new model from scratch, both in terms of the size of the training data required and in terms of hardware resources. Transfer learning doesn't just apply to image classification models; the same principle can be applied to any kind of machine learning model.

In the next section, we'll import a KNN model and use **`knnClassifier`** to train the model to identify specific objects. This is great when you want to train a model for your specific use case. Of course, making a model to deal with all the edge cases can take a very long time, but you'll see that you can get something functional trained and running in less than half an hour.

THE K–NEAREST NEIGHBOR ALGORITHM

K-nearest neighbor (KNN) is an algorithm that's used for classification and regression problems in machine learning. The specifics of KNN are complex, and beyond the scope of this chapter, but essentially, it classifies a piece of data based on closely related data points. If a model is taught that all round fruits are apples, and all curvy fruits are bananas, then, when presented with an unknown curvy fruit, the KNN algorithm will predict that it's a banana.

EXERCISE 16.03: MODEL TRAINING IN THE BROWSER

In this exercise, we'll be training our own model to tell the difference between two different objects. Which objects you want to train for is up to you. You only need to follow two recommendations. Firstly, it needs to be two objects you have at hand because we're going to be using the camera. Secondly, the two objects should be significantly different from each other; this will make training the model quicker and easier.

I'm going to be demonstrating using the example of clocks and cups, simply because I happen to have those items at hand and they're significantly different from each other. Feel free to switch our watch or cup with whatever you have around.

In order to implement transfer learning on the model, we need to import a Tensorflow.js utility model, that is, KNN Classifier. KNN Classifier allows us to create a classifier using the K-nearest neighbor algorithm. We can use the same basic HTML file that we used in the previous exercises. Let's get started:

1. We'll start off by importing the KNN Classifier model. To do so, add the following line below the MobileNet import in the HTML **head** tag:

```
<script src="https://unpkg.com/@tensorflow-models/knn-classifier"></
script>
```

   We will train our model by presenting new objects to the webcam and calling a training function to let the model know that a new object class is in view. To do this, we will need to add some buttons to our HTML page – we'll add two for now, but you can add as many as you like; one button for each new object class you want to train.

2. Add the buttons right below the **<video>** element. You can use **
** to separate them from the video:

```
<br/>
<button id='button-a' onclick='trainModel("watch")'>Train Watch</
button>
<button id='button-b' onclick='trainModel("cup")'>Train Cup</button>
```

   We'll use these buttons to train the model, depending on what is being presented to the webcam. The pen and cup strings can be replaced with any item or class you like; either specific objects like these or more situational descriptors such as "smile" and "sad face." Choose any words you like, depending on what items you have around you, or what different scenes you're able to present to the webcam. You can also add more than two buttons if you choose.

Next, we'll make some changes to our JavaScript file. We need to create a new KNN classifier, write a model training function, which is the **`onclick`** event handler function for our buttons, and then change our **`infinite`** loop to display predictions made by the classifier, instead of those made by the pretrained model.

3. At the top of **`tf.js`**, below the **`statusElem`** variable declaration, we will create a KNN Classifier and instantiate a variable that will hold our model training function. We can leave that variable undefined for now:

```
  const classifier = knnClassifier.create();
  let trainModel;

  initWebcam();
```

4. Then, inside the **`classifyImage()`** function, we'll set the **`status`** element's **`innerText`**, load the model like we did previously, and initialize the webcam:

```
async function classifyImage() {
statusElem.innerText = 'Loading MobileNet...';
const model = await mobilenet.load();
```

5. Next, we'll define the **`trainModel()`** function, which creates an **`activation`** and adds it to the classifier as an example of the image classification specified by the class ID (the class ID will be the index of the class example we're training the model on):

```
  trainModel = classId => {
    const activation = model.infer(videoElem, 'conv_preds');
    classifier.addExample(activation, classId);
  }
}
```

6. Next, we'll make some changes to the **`while`** loop. The **`if`** statement will check to see whether the classifier has been trained on at least one class:

```
while (true) {
  if (classifier.getNumClasses() > 0) {
```

7. If it has, we get an **`activation`** from the MobileNet model, assign it to a variable of the same name, and pass that **`activation`** to the classifier to make a prediction, thus assigning the result to another variable, called **`result`**:

```
    const activation = model.infer(videoElem, 'conv_preds');
    const result = await classifier.predictClass(activation);
```

The result of the prediction will be an object with the **`label`**, **`confidences`**, and **`classIndex`** properties. The **`label`** property will be the **`classLabel`** string of the class the model is predicting with the highest confidence. The **`confidences`** property is an object with each class label and its corresponding confidence score as key/value pairs, and the **`classIndex`** property is the index at which the **`classLabel`** was added to the classifier when we first trained it on the class.

8. We'll assign the **`status`** element's **`innerText`** property to the label and confidence of the model's class prediction:

```
    statusElem.innerText = `${result.label} -
${result.confidences[result.label]}`
  }
  await tf.nextFrame();
}
```

9. Our code is now complete. Next, open up your browser and start the training process. Hold your first item up to the webcam (a cup, in my case) and press the button associated with the item. You will see the status text now displaying **`cup - 1`**:

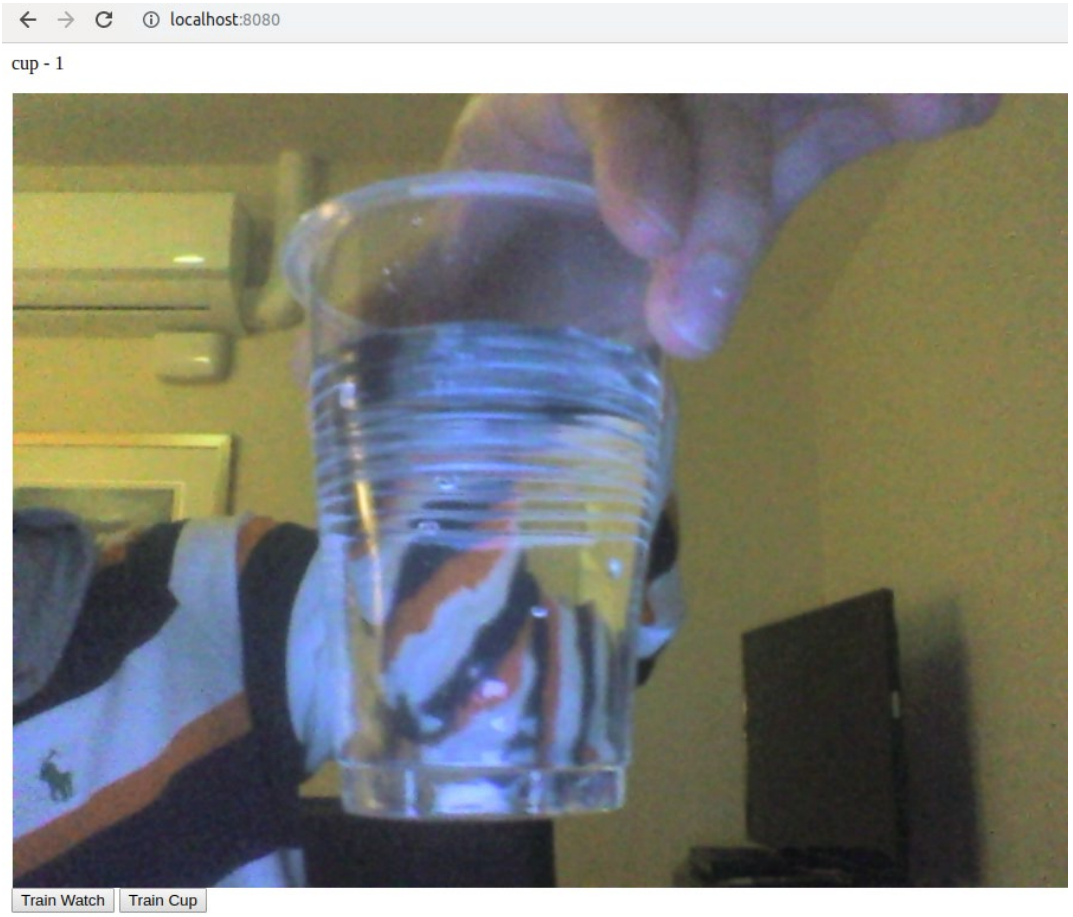

Figure 16.12: Training the model to predict an image

10. Next, hold up yo ur second item and repeat step 9 but using the other button. The result isn't guaranteed, but you'll likely see a value of **0.5** for one of the items:

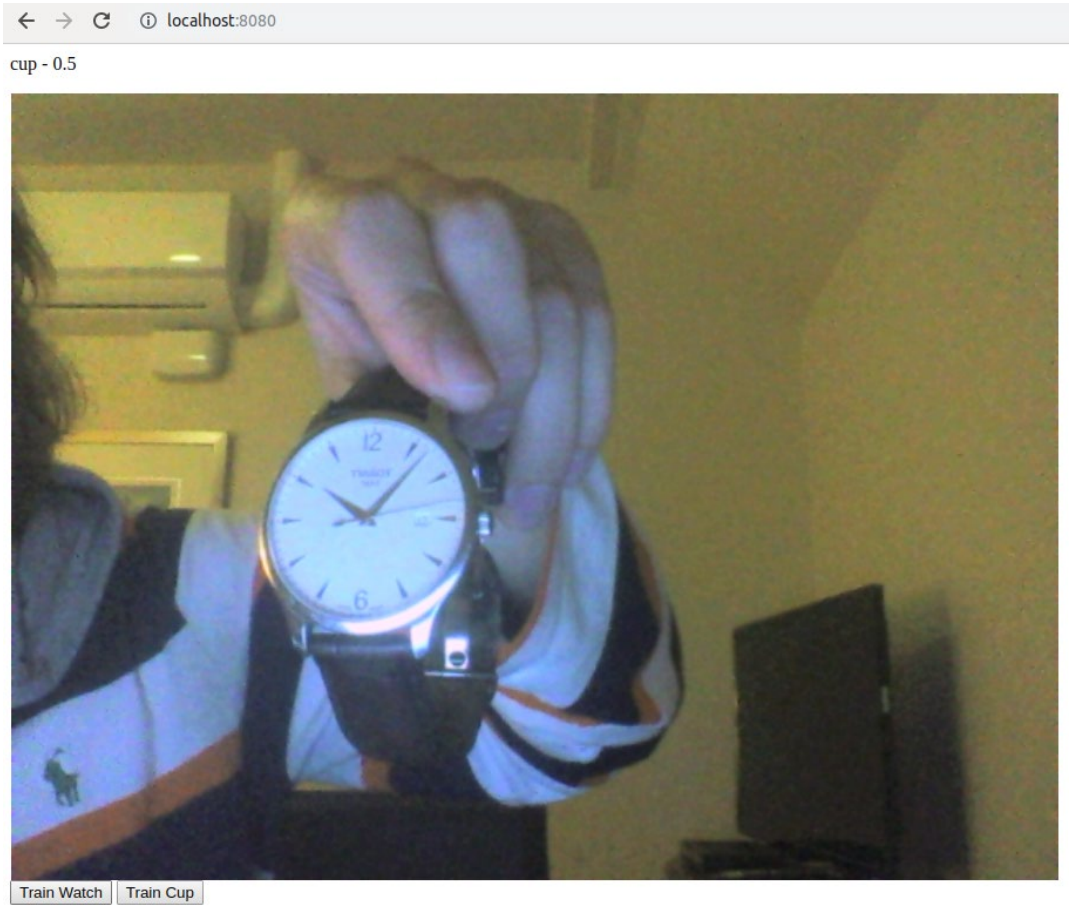

Figure 16.13: Result after training the model

From here, continue to train the model by switching between the two items and pressing the button for each. With more entries, the accuracy of the model should improve. You may want to move on to showing additional types for your objects. For example, you could experiment with different types of cups and see if you are able to get the model to make predictions about cups it hasn't seen before.

The accuracy results we will see after each round of training, with each round consisting of one click of the button for each item are listed in *figure 16.14*. That was with a single version of each item, so there was no variation. After only three rounds of training, the model was able to identify the specific cup and watch with 100% certainty. Of course, this doesn't mean it is *100%* accurate, only that the model was certain of it. After the fourth round, a different cup was tried, which it predicted was a cup with 0.66 probability:

Round number	Watch	Cup
#1	0.5	0.5
#2	0.66	0.66
#3	1	1
#4	1	1

Figure 16.14: Accuracy results

Try to include variations of the same class and train the model for each variation. The aim is to provide as many examples of a class as possible to allow the model to accurately predict that class. Do this for all of the classes you want to train the model on. Remember that you're not limited to two buttons either – you can modify the code to add as many as you want. The predictions above the webcam stream will update continuously as before, but this time, with good training data, their accuracy will increase with every click of the **`button`**.

In this exercise, we looked at training a KNN Classifier to recognize specific objects based on the input we show it. We've considered how we can improve a model by showing a greater number of examples and variations. By utilizing these tools, it should be possible to train the built-in KNN model to recognize objects for your specific use case.

TRAINING A NEW MODEL

So far, we've looked at how we can import and use an existing TensorFlow model, and also how we can build on that model using transfer learning to make it more specialist in one particular area. Next, we'll look at training a model from scratch.

Using TensorFlow's high-level Layers API, we can train a model from a new dataset and have it make predictions on similar, unseen data.

In this example, we're going to use data on wheat seed kernels. A kernel's variety can be established by taking various measurements of the kernel (for example, it's length, diameter, and so on). We will use a dataset of measurements from 210 kernels, each one belonging to one of three varieties, to train a new model from scratch. We can then feed it the measurements of a kernel that the model has never seen before, and it will be able to predict which variety of kernel those measurements belong to. The accuracy of the prediction will depend largely on the parameters we set when training the model, and how many training **epochs**, or **iterations**, we train it for.

Let's check out the first row of our data to see what we're working with. Comments describing what each value represents have been added so that you have an idea of the data we're working with. Take note of the very last value, which represents the classification of the kernel; it can be 0, 1, or 2. Our goal in training this model will be to have it take the first seven values and predict the eighth value (category) without us telling it to:

```
[
 15.26, // area
 14.84, // perimeter
 0.871, // compactness
 5.763, // length of kernel
 3.312, // width of kernel
 2.221, // asymmetry coefficient
 5.22,  // length of kernel groove
 0      // kernel type 0, 1, or 2
]
```

This array of numbers represents the data for one kernel. For each kernel, we have eight data points; seven are the kernel's geometric measurements, and the eighth data point represents the kernel's variety. There are three varieties in our dataset, so the eighth data point will always be 0, 1, or 2. So, to put this in common machine learning parlance, the first seven data points are our *X*s (inputs), and the last one is our *Y* (the output, or target).

There are three main steps we will go through in this example:

- Preparing the training data
- Training the model
- Using the model for prediction

Let's go through each one, starting with preparing the data.

EXERCISE 16.04: PREPARING THE DATA

Before we can train a model, we have to make sure that the training data is in a format that TensorFlow can understand. We will also divide our data into training data and testing data so that the model can check its predictions as it goes through the training process. Let's get started:

1. We'll write a simple HTML file **`index html`**, with a single **`div`** element in the body, with the **`id`** of 'output', and two script tags for our **`ts`**.js file and the seed data we'll be training the model on:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/@tensorflow/tfjs"></script>
  </head>
  <body>
    <div id='output'></div>
    <script src='seedData.js' type='application/javascript'></script>
    <script src="tf.js"></script>
  </body>
</html>
```

2. Now we'll create a tf.js file, and start off by writing a function to prepare the data:

```
//tf.js
function getSeedData() {
```

3. We'll declare a two-dimensional array variable to hold our input data (the *X*s):

```
  const dataByClass = [
    [],
    [],
    []
  ];
```

4. And another one to hold the target data, or output data (the *Y*s):

```
  const targetsByClass = [
    [],
    [],
    []
  ];
```

5. We will then sort the seed data into its inputs and outputs and push the respective parts of the data to the **dataByClass** array and the **targetsByClass** array:

```
for (const seed of seedData) {
  const seedClass = seed[seed.length - 1];
  const data = seed.slice(0, seed.length -1);
  dataByClass[seedClass].push(data);
  targetsByClass[seedClass].push(seedClass);
}
```

It may not be immediately obvious why we're dividing our data in this way, so let's run through the last few steps. This is the start of our data preparation; we want to divide our data into the input data (the *X*s), which are the seed measurements in this particular dataset, and our target outputs (the *Y*s), which are the seed varieties. Because there are three different seed varieties in our set, we're further dividing the input data and target outputs into three groups, one for each seed variety. So, we've declared two arrays, **dataByClass** and **targetsByClass**, which will hold our processed data. Next, we're looping through each seed in our dataset and adding the seed data to its **dataByClass** subarray, based on the seed's variety (we're removing the seed variety from the data since we don't want TensorFlow to attempt to use that in the learning process). We're then pushing the variety to the same subarray in **targetsByClass**.

6. If we close the **getSeedData** function here and add a console log, we can take a look at what the two multi-dimensional arrays look like at this point. The first one, **dataByClass**, is displayed in the following screenshot. It consists of three subarrays of **70** (notice that it adds up to the total number of seed data points, that is, **210**). Each of the three arrays represents all the seeds for a type:

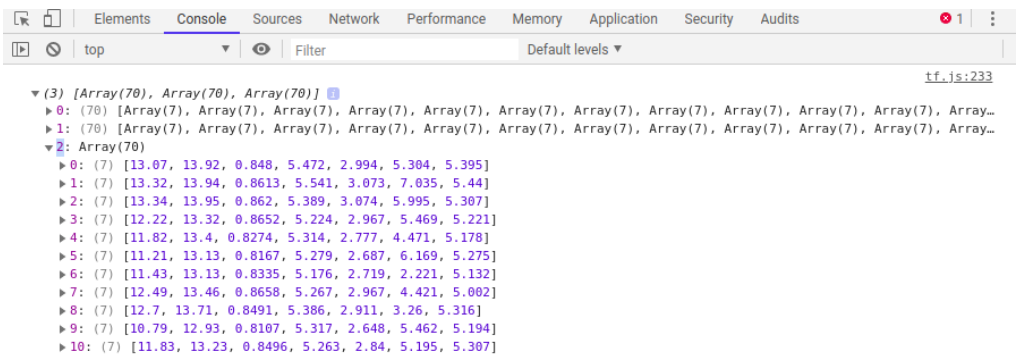

Figure 16.15: Arrays representing all seeds

7. The following screenshot shows what **`targetsByClass`** looks like. Simply display the corresponding class for the three groups, that is, one array containing *70 0s*, another with *70 1s*, and one with *70 2s*:

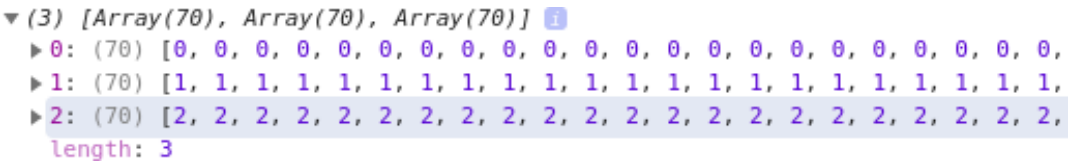

Figure 16.16: Arrays displayed by class

We haven't finished with the data processing yet, so let's look at what comes next.

8. We will declare four more array variables to hold our training input data, training output data, and the testing input data and testing output data. The last two datasets will be used by the model during training to test how accurate it is becoming:

```
  const trainingXs = [];
  const trainingYs = [];
  const testingXs = [];
  const testingYs = [];
```

9. Next, we will convert the input and output data into TensorFlow tensors using the **`converToTensors()`** method – which we will define next – and push the results into the corresponding array, that is, either **`trainingXs`**, **`trainingYs`**, **`testingXs`**, or **`testingYs`**:

```
  for (let i = 0; i < dataByClass.length; i++) {
    const [trainX, trainY, testX, testY] =
convertToTensors(dataByClass[i],
targetsByClass[i], 0.2);
    trainingXs.push(trainX);
    trainingYs.push(trainY);
    testingXs.push(testX);
    testingYs.push(testY);
  }
```

10. We'll now return these results and concatenate them using TensorFlow's **`concat`** function:

```
  return [
    tf.concat(trainingXs, 0), tf.concat(trainingYs, 0),
    tf.concat(testingXs, 0), tf.concat(testingYs, 0)
  ];
}
```

    Again, let's run over the last few steps in more detail. This part of the processing is where we split the data into training and testing data. It's common to split data into *80%* training data and *20%* testing data, and that's what we're doing here. We are also making use of another function, **`convertToTensors()`**, which takes some data and converts it into TensorFlow tensors. We'll look at that function shortly. The converted data is then pushed into one of four arrays, that is, **`trainingXs`**, **`trainingYs`**, **`testingXs`**, or **`testingYs`**. Finally, we are using TensorFlow's **`tf.concat()`** method to concatenate the four two-dimensional arrays into one-dimensional arrays and returning the result.

    Before we start training our model, let's go over the **`convertToTensors()`** method, which was used in the previous code block.

11. We'll declare the function that takes the **`data`**, **`targets`**, and **`split`** parameters as arguments:

```
function convertToTensors(data, targets, split) {
```

12. We assign the number of data points to the **`numExamples`** variable:

```
  const numExamples = data.length;
```

13. We assign the number of data points to be used for testing to the **`numTestExamples`** variable:

```
  const numTestExamples = Math.round(numExamples * split);
```

14. Now, we'll assign the number of data points to be used for training to the **`numTrainExamples`** variable:

```
  const numTrainExamples = numExamples - numTestExamples;
```

15. Next, we'll convert the data into a TensorFlow tensor and convert the targets into one-hot format using TensorFlow's **`tf.oneHot()`** method:

> **NOTE**
>
> One-hot encoding will be explained in the next section.

```
    const xDims = data[0].length;
    const xs = tf.tensor2d(data, [numExamples, xDims]);
    const ys = tf.oneHot(tf.tensor1d(targets).toInt(), 3);
```

16. **`Finally`**, we divide the training and testing data and targets, and then assign each group to a variable, that is, either **`xTrain`**, **`xTest`**, **`yTrain`**, or **`yTest`**, and then return these variables:

```
    const xTrain = xs.slice([0,0], [numTrainExamples, xDims]);
    const xTest = xs.slice([numTrainExamples, 0], [numTestExamples,
xDims]);
    const yTrain = ys.slice([0,0], [numTrainExamples, 3]);
    const yTest = ys.slice([0,0], [numTestExamples, 3]);
    return [xTrain, yTrain, xTest, yTest];
}
```

ONE-HOT ENCODING

One-hot encoding is a way of specifying one item in a series. In our seeds data, the seed variety is indicated by the last number, which is either 0, 1, or 2. One-hot encoding gives a list of binary values, with the selected item being represented as 1, and all the others as 0. So, seed variety 0 would be represented as *[1, 0, 0]* (as it's the 0th item), variety 1 would be *[0, 1, 0]*, and variety 2 would be *[0, 0, 1]*.

This format is commonly used in machine learning, and TensorFlow even has some built-in methods for working with it. The method takes an array of numbers representing the categories of a list of seeds and an integer representing the number of categories. So, if we had a list of three seeds with one of each type and we wanted to convert it into one-hot encoding, we could run the following code:

```
tf.oneHot(tf.tensor1d([0, 1, 2], 'int32'), 3).print();
```

The output would be displayed as follows:

```
> tf.oneHot(tf.tensor1d([0, 1, 2], 'int32'), 3).print();
  Tensor
      [[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]]
<- undefined
>
```

Figure 16.17: List of seeds converted into one-hot encoding

Let's try the same thing but with a random list of seeds:

```
tf.oneHot(tf.tensor1d([0, 1, 1, 2, 2, 0, 0], 'int32'), 3).print();
```

The output will be displayed as follows:

```
> tf.oneHot(tf.tensor1d([0, 1, 1, 2, 2, 0, 0], 'int32'), 3).print();
  Tensor
      [[1, 0, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 0],
       [1, 0, 0]]
<- undefined
>
```

Figure 16.18: Random seeds converted into one-hot encoding

EPOCHS

An epoch is an iteration of analysis on a dataset. When training a model, you'll often have the opportunity to decide on the number of epochs you want to run. Typically, the more epochs you run, the more accurate your model will be; however, in some cases, excessive epochs can contribute to overfitting a model to a dataset. You'll also need to consider the length of time each epoch takes on your specific hardware and the fact that, typically, you will see diminishing returns on additional epochs over time.

When fitting a model, you'll be able to specify the number of epochs you want to use. For example, let' say we wanted to use **`40`**:

```
await model.fit(xTrain, yTrain, {
  epochs: 40,
  validationData: [xTest, yTest],
  callbacks: {
    onEpochEnd: async(epoch, logs) => {
      console.log(`Epoch: ${epoch}, Logs: ${logs.loss}`);
      await tf.nextFrame();
    }
  }
});
```

ACTIVATION FUNCTIONS

An activation function is the type of process performed at a given layer of a neural network. An **`activation`** function defines how a neuron transforms a given input into an output. You won't need to understand how activation functions work to use them.

In our case, we'll make use of sigmoid and softmax.

EXERCISE 16.05: TRAINING THE MODEL

Our next step is to train the model on our data. We'll add this code to the existing tf.js file that we started in the previous exercise. Again, let's break this function down into steps:

1. We'll declare an asynchronous function called **`trainModel()`**. This function takes in the **`xTrain`**, **`yTrain`**, **`xTest`**, and **`yTest`** tensors that we got back from our **`getSeetData()`** function. We set **`trainModel()`** to be an **`async`** function because when we come to actually train the model, we will await the training's completion:

```
async function trainModel(xTrain, yTrain, xTest, yTest) {
```

2. Inside the function block, we initialize a **`model`** variable as a sequential TensorFlow model:

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

3. We then add two layers to it. The first layer is for the inputs, that is, **`xTrain`**, and has seven units. The activation function is set to **`sigmoid`** and the input shape is set by the training data's shape:

```
model.add(tf.layers.dense(
  {units: 7, activation: 'sigmoid', inputShape: [xTrain.shape[1]]}
));
```

4. The second layer, for the outputs, is given three units, and has an activation of softmax:

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

   There are many activation functions to choose from, and it would be the subject of another chapter to go into depth on these. For a quick overview, the sigmoid activation function will normalize any input data and pass it to the next layer as a number between **`0`** and **`1`**. The softmax function ensures that the sum of all output values from any given input is **`1`**. This will make sense when we see the result of the model making a prediction.

5. The next step is to compile the model, to which we pass the **`adam`** optimizer, a loss function of **`categoricalCrossentropy`**, and the metrics of **`accuracy`**:

```
model.compile({
  optimizer: tf.train.adam(0.01),
  loss: 'categoricalCrossentropy',
  metrics: ['accuracy']
})
```

6. We then carry out the training properly with TensorFlow's **`model.fit()`** method. In this function, we pass in the **`xTrain`** data, the **`yTrain`** data, and a configuration object:

```
await model.fit(xTrain, yTrain, {
```

7. The object sets the number of **`epochs`** or **`iterations`** over which the training should take place. For every **`epoch`**, the training is validated against using the **`xTest`** and **`yTest`** data (as you'll no doubt remember, this is the test input data and test output data):

```
  epochs: 40,
  validationData: [xTest, yTest],
```

8. A **`callback`** function is called for the end of each epoch, which simply logs the progress of the training in the form of the epoch number and the loss value for that particular epoch:

```
      callbacks: {
        onEpochEnd: async(epoch, logs) => {
          console.log(`Epoch: ${epoch}, Logs: ${logs.loss}`);
          await tf.nextFrame();
        }
      }
    })
```

9. When the training is complete, the model is returned to the calling function:

```
    return model;
    }
```

This completes the model training processes. The **`trainModel`** function we created here can be called with the data returned from **`getSeedData`**. Doing so will return a trained model. In the next exercise, we'll provide some inputs and check the results.

EXERCISE 16.06: USING THE MODEL FOR PREDICTION

In this exercise, we'll continue from where we left off in *Exercise 16.05, Training the Model*, and test our model for predicting data. Again, we'll add the following code to the save tf.js file that we started in the last exercise:

1. Let's write a new function to tie all these processes together:

```
    async function learnSeeds() {
```

2. We're assigning the return values from **`getSeedData()`** to the **`xTrain`**, **`yTrain`**, **`xTest`**, and **`yTest`** variables and then calling the **`trainModel()`** method with these variables as parameters, and then we're assigning the resulting trained model to the **`model`** variable:

```
      const [xTrain, yTrain, xTest, yTest] = getSeedData();
      model = await trainModel(xTrain, yTrain, xTest, yTest);
```

3. We can now assign a new two-dimensional tensor to a new variable called **`input`**, which will contain the measurements of a new wheat seed kernel that hasn't been seen by the model yet (after removing its **`variety`** value):

```
      const input = tf.tensor2d([12.2, 16.37, 0.8284, 5.213, 2.894,
    5.635,
    5.103], [1, 7]);
```

4. Then, we simply call the **`model.predict()`** method with this input and log the result:

```
  const prediction = model.predict(input);
  console.log(prediction.toString());
}
learnSeeds();
```

Now, if you run this code, you should see something like the following displayed in the console:

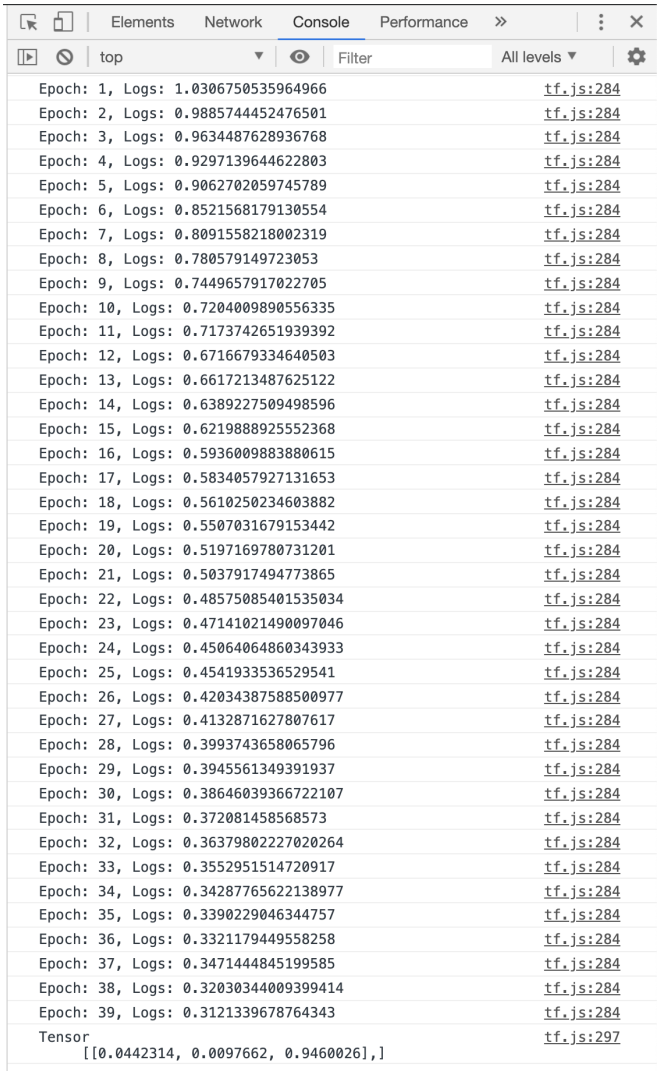

Figure 16.19: Prediction result

You can see that the model predicts with *94%* confidence that the seed is of the third class, and we can validate this ourselves by looking at the seed's original data. We can try to improve the model's accuracy by changing some of the parameters that we set during the training process, for example, the epoch parameter, the number of units in the input layer, or the split ratio of training data versus testing.

ACTIVITY 16.01: USING TENSORFLOW.JS AND COCO-SSD OBJECT DETECTION TO CREATE AN APP

So far, we've been implementing some pretty cool, but ultimately quite basic, machine learning functionality. In this activity, we're going to build an object classifier with some better functionality using another pretrained model called **COCO-SSD**. COCO stands for Common Objects in Context, and SSD stands for Single Shot Multibox Detector (the "M" got left out of the acronym, apparently). It's an object detection model that takes an image or video stream as an input and will attempt to detect objects in that image. It was trained on the COOC image dataset of 90 image classes. As well as inferring the class of object that the model has detected, COCO-SSD will provide the bounding box of where it thinks the object is within the image. The results are provided as an array of JavaScript objects, each one representing a detected object, which will be in the following format:

```
{
bbox: (4) [3.6824073791503906, 4.639701247215271, 888.1363487243652,
661.2557280063629],
    class: "person",
    score: 0.6901552677154541
}
```

Your aim for this activity is to create an application with TensorFlow.js and COCO-SSD object detection, which draws labeled bounding boxes around all the detected objects that were received through the computer's webcam stream. For the bounding boxes and labels, use what you learned in *Chapter 7, Copying the Hood,* regarding the Canvas API.

Here's a screenshot of the desired outcome:

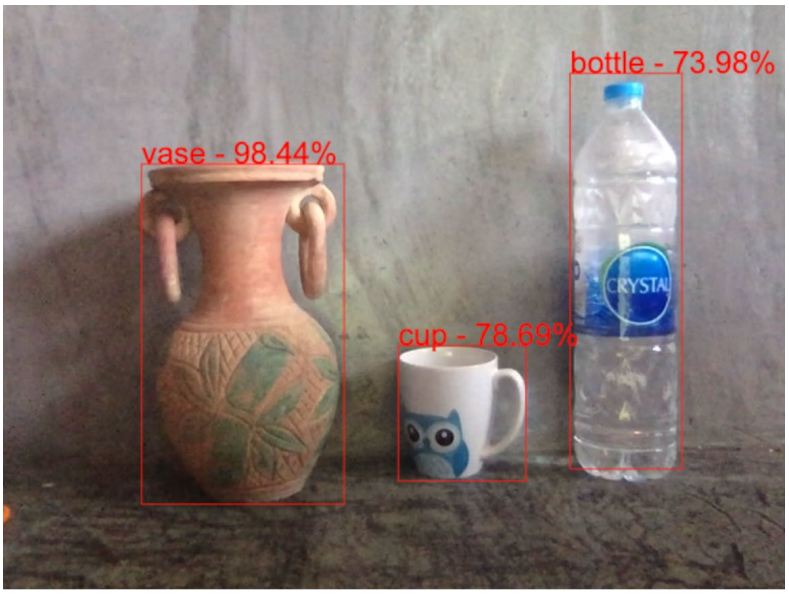

Figure 16.20: COCO-SSD bounding boxes example

STEPS

To get you started, here are some high-level steps to follow to create this application:

1. Create an HMTL page with the TensorFlow.js library and the **`COCO-SSD`** library imported. You can use the **`CDN`** method of importing the **`COCO-SSD`** JavaScript library with this link:

   <script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/coco-ssd"></script>.

   A JavaScript object called **`cocoSsd`** is made available from the CDN import, and it works similarly to the MobileNet model we used earlier in this chapter. It has a **`load()`** method and a **`detect()`** method instead of MobileNet's **`classify()`** method.

2. Give the application access to the webcam feed in the HTML and create a **`canvas`** element that will overlay the webcam feed. This will be where we draw the bounding boxes and labels.

3. In JavaScript, load the **`COCO-SSD`** model.
4. Initialize the webcam.
5. Use the **`COCO-SSD`** model to detect objects that are presented to the webcam.
6. Use the results array's elements to draw bounding boxes and labels on the **`canvas`** element.

SUMMARY

In this chapter, we scratched the surface of a very deep subject. We looked at what machine learning is and what it can be used for at a very high level. We then covered how neural networks are put together and implemented some basic machine learning applications using the three techniques of using a pretrained ML model: transfer learning, extending the ability of an existing model, and training a brand-new model from scratch. AI is already widely used in many aspects of life and will become ever more important as processing power increases, smarter models are trained, and new machine learning applications are discovered.