

APPENDIX

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

Solution:

1. First off, let's look at the HTML file **index.html**. It's very similar to the HTML we used in our MobileNet examples earlier in this chapter:

index.html

```
1 <html>
2   <head>
3     <script src="https://unpkg.com/@tensorflow/tfjs"></script>
4     <script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/coco-
5     ssd"></script>
6   </head>
7   <body>
8     <div id="console"></div>
9     <div class='cam-canvas-wrapper'>
10       <video autoplayplaysinline muted id="video" width="640"
11       height="480"></video>
12       <canvas id='canvas'></canvas>
13     </div>
14     <script src="tf.js"></script>
15   </body>
16 </html>
17 <style>
```

<http://packt.live/2NvzzlC>

We import the TensorFlow and COCO-SSD libraries, and then, in the HTML body, we add a **<div>** to contain the webcam video feed and the **canvas** elements. We set the **video** element's height and width so that it matches the webcam. Next, we import our **tf.js** file, and at the bottom, we add a **<style>** tag with some basic CSS styling to get the **canvas** element to align properly with the **video** element, with **<canvas>** on top of **<video>** (on the z-axis).

2. Let's move on to the JavaScript file. Now, we'll declare an asynchronous function called **objectDetection**, grab references to the **video** and **canvas** elements, and match the canvas' width and height to those of the **webcam** element. Next, we'll load the model in the same way that we did with the **MobileNet** model earlier and assign that to a variable called **model**. After that, we get the canvas context and set it up with color and font settings. Finally, for this part of the function, we call the **initWebcam()** async function, which is the same as we did in our previous webcam example:

```
// tf.js
const webcam = document.getElementById('video');
async function objectDetection() {
  let canvas = document.getElementById('canvas');
```

```
canvas.width = webcam.width;
canvas.height = webcam.height;
let model = await cocoSsd.load();
let context = canvas.getContext('2d');
context.strokeStyle = 'red';
context.fillStyle = 'red';
context.font = "25px Arial";
await initWebcam();

...

}
```

3. As before, we have an infinite **while** loop that will be used to detect objects with COCO-SSD. This time, instead of printing the results of the detection above the webcam stream, we will be plotting the results' bounding boxes and labeling them with the name of their class. First of all, we call the **clearRect()** method of the canvas context, which gives us a blank canvas to work with for each frame. Next, we call **beginPath()** to start a new path on the canvas and then pass in the current frame of the webcam to the COCO-SSD model for detection, thereby assigning the results to a variable:

```
// tf.js
async function objectDetection() {

  ...

  while (true) {
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.beginPath();
    const results = await model.detect(webcam);
```

If there is at least one result, we loop through the **results** array and plot the **results** class and score as canvas text, and a box around the object, on the condition that the model is at least 30% sure of its prediction. Let's walk through the two lines that are responsible for plotting the boxes. The **context.fillText()** method plots the text that we pass to it in the first argument, starting at the *x/y* coordinates provided in the second and third arguments. The text we pass is the **results** class, and its prediction score is converted into a percentage and rounded to two decimal places. After this, we plot the box by using the spread syntax to pass in the result's **bbox** property to the **context.rect()** method. At the end of the **if** statement, we call **context.stroke()** to draw the plotted text and shapes and then await the return of **tf.nextFrame()**:

```
if (results.length) {
  results.forEach((result, index) => {
    if (result.score > 0.3) {
      context.fillText(`${result.class} - ${((result.score * 100).
toFixed(2))}%`, result.bbox[0], result.bbox[1])
      context.strokeRect(... result.bbox);
    }
  })
}
context.stroke();
await tf.nextFrame();
}
```

4. We have one more function to write – **initWebcam()**. It's the same as the function we wrote earlier in this chapter, so we'll just show it here without going through it all:

```
async function initWebcam() {  
  if ('mediaDevices' in navigator && 'getUserMedia' in navigator.  
mediaDevices) {  
    try {  
      const stream = await navigator.mediaDevices.  
getUserMedia({video: true});  
      videoElem.srcObject = stream;  
    } catch (error) {  
      console.log(`Error getting video: ${error}`);  
    }  
  }  
}
```

5. Now, if we add a call to **objectDetection()** at the end of our JavaScript file, and load the HTML page, we should see a webcam feed with boxes around all the objects that the model can detect, and with their classes printed at the top of the boxes.

ACTIVITY 17.01: TEST-DRIVEN DEVELOPMENT

Solution:

1. Define the function signatures. Each function has the same signature and returns the same data type:

```
let add = (a, b) => {};  
let subtract = (a, b) => {};  
let multiply = (a, b) => {};  
let divide = (a, b) => {};
```

2. Next, write the success and failure test cases for each function. No body is needed at this time:

```
test('add returns number when successful', () => {});  
test('add returns NaN when unsuccessful', () => {});  
test('subtract returns number when successful', () => {});  
test('subtract returns NaN when unsuccessful', () => {});  
test('multiply returns number when successful', () => {});  
test('multiply returns NaN when unsuccessful', () => {});  
test('divide returns number when successful', () => {});  
test('divide returns NaN when unsuccessful', () => {});
```

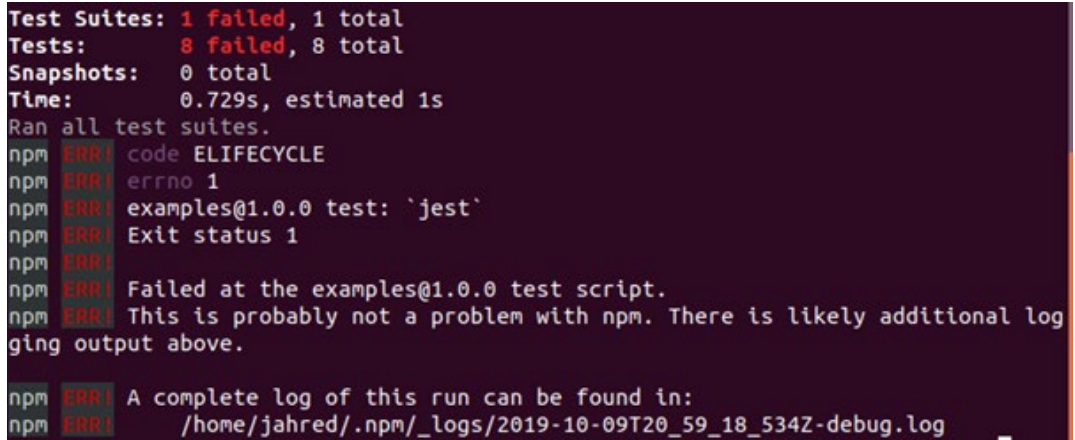
3. Now, populate the tests based on how each function should behave:

index.test.js

```
26 test('add returns number when successful', () => {  
27   expect(typeof add(1, 2)).toBe("number");  
28 });  
29 test('add returns NaN when unsuccessful', () => {  
30   expect(add("1", 2)).toBeNaN();  
31 });  
32 test('subtract returns number when successful', () => {  
33   expect(typeof subtract(1, 2)).toBe("number");  
34 });  
35 test('subtract returns NaN when unsuccessful', () => {  
36   expect(subtract("1", 2)).toBeNaN();  
37 });  
38 test('multiply returns number when successful', () => {  
39   expect(typeof multiply(1, 2)).toBe("number");  
40 });
```

<https://packt.live/3a38gdV>

4. Run the tests. They should all fail:

A terminal window with a dark background and light-colored text. It shows the output of a Jest test run. The summary indicates 1 failed test suite and 8 failed tests. Below the summary, it shows an npm error: 'code ELIFECYCLE', 'errno 1', and 'Exit status 1'. It also states that the test failed at the 'examples@1.0.0 test script' and provides a path to a debug log file.

```
Test Suites: 1 failed, 1 total
Tests:      8 failed, 8 total
Snapshots:  0 total
Time:       0.729s, estimated 1s
Ran all test suites.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! examples@1.0.0 test: `jest`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the examples@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional log
ging output above.

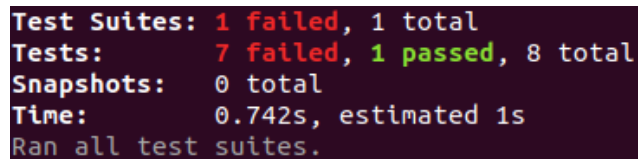
npm ERR! A complete log of this run can be found in:
npm ERR!     /home/jahred/.npm/_logs/2019-10-09T20_59_18_534Z-debug.log
```

Figure 17.18: Failing test cases

5. Update the code so that at least the first test passes:

```
let add = (a, b) => {
  return a + b;
};
```

6. Running the tests should show one test passing:

A terminal window showing the output of a Jest test run after a code update. The summary now shows 1 passed test and 7 failed tests. The rest of the output is identical to the previous screenshot.

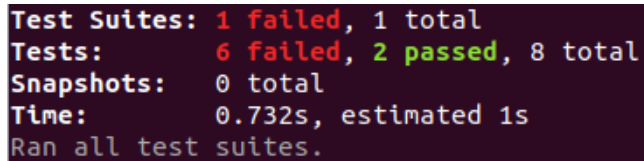
```
Test Suites: 1 failed, 1 total
Tests:      7 failed, 1 passed, 8 total
Snapshots:  0 total
Time:       0.742s, estimated 1s
Ran all test suites.
```

Figure 17.19: One test pass output

7. Now, update the code so that the next test passes:

```
let add = (a, b) => {
  if (typeof a !== "number" || typeof b !== "number") {
    return NaN;
  }
  return a + b;
};
```

8. Running the tests should now show two tests passing:



```
Test Suites: 1 failed, 1 total
Tests:      6 failed, 2 passed, 8 total
Snapshots:  0 total
Time:       0.732s, estimated 1s
Ran all test suites.
```

Figure 17.20: Two tests pass output

9. Repeat this for all the tests until all the tests pass.

This same methodology is how all TDD projects should be approached. This same process works identically, regardless of whether the function being tested is simple or complex. Simply define the scope of the function with tests, then flesh it out until all the tests pass.

ACTIVITY 17.02: SENTENCE PARSING TEST SUITE

Solution:

- Let's begin by identifying the commonalities of each function so that the tests can be reduced and you can avoid unnecessary clutter. You know the following about each test:
 - They should receive an **Identity** argument.
 - The argument should contain a value.
 - They should return an **Identity** if the argument is valid.
 - They should return a **Failure**, with a set error message, if the argument is not valid.
 - They should return a **Failure** if a **Failure** was passed as an argument. The argument and the returned value should match the content.All of these tests can be combined into a single matcher.

2. In the `functor_matchers.js` file, create a new matcher with the following form:

`functor_matchers.js`

```

25 let toBeValidFunctorFunction = function(received, arg, errMsg) {
26   let pass = true;
27   let successId = Identity.from(arg);
28   let failId = Identity.from({});
29   let failure = Failure.from(Math.random());
30   let result = received(successId);
31   let message = () => "";
32   pass = pass && (result instanceof Identity);
33   if (!pass) {
34     message = () => `Fails with expected success value ${arg} with result
35     ${result}`;
36     return {actual: received, message, pass};
37   }
38   result = received(failure);
39   pass = pass
40   && (result instanceof Failure)

```

<http://packt.live/36ZxAOP>

As you can see, the matcher expects two parameters: the successful value and the generated error message. With these, it is possible to test what happens for each of the required parameter permutations.

3. Add the `toBeValidFunctorFunction` matcher to the exported list:

```
export {toBeIdentity, toBeFailure, toBeValidFunctorFunction};
```

4. With the matcher in place, you can now start writing the tests. First of all, import the **Functors**, the matchers, and the sentence functions themselves, and then extend **expect** with the matchers:

```

import {Identity, Failure} from './functors';
import {toBeIdentity, toBeFailure, toBeValidFunctorFunction} from
  './functor_matchers';
import {validateString, splitStringIntoList, rejectEmptyStrings,
  mapStringLength, averageStringLength} from './sentence';
expect.extend({toBeIdentity, toBeFailure,
  toBeValidFunctorFunction});

```

- Next, write the required tests for each of the functions:

sentence.test.js

```
14 describe('Sentence Functions', () => {
15   test('validateString is valid', () => {
16     const arg = "The quick brown fox";
17     expect(validateString).toBeValidFunctorFunction(arg, "invalid
argument");
18     expect(validateString(arg)).toBeIdentity(arg);
19   })
20   test('splitStringIntoList is valid', () => {
21     const arg = "The quick brown fox";
22     const resp = ["The", "quick", "brown", "fox"];
23     expect(splitStringIntoList).toBeValidFunctorFunction(arg, "invalid
argument");
24     expect(splitStringIntoList(arg)).toBeIdentity(resp);
25   })
26   test('rejectEmptyStrings is valid', () => {
27     const arg = ["The", "", "quick", "", "brown", "", "fox"];
28     const resp = ["The", "quick", "brown", "fox"];
```

<http://packt.live/2NGhxNQ>

Note that you only need one test for each function since the matcher takes care of most of the work for you.

- Now, you need to write the functions themselves. Since most of the functionality in the **validateString** function will be required for all the functions, let's extract that into its own function first:

```
let valid = (value, typeCheck, transform) => {
  if (value instanceof Failure)
    return value;
  if (!(value instanceof Identity))
    return Failure.from("invalid argument");
  if (!typeCheck(value.get()))
    return Failure.from("invalid argument");
  return Identity.from(transform(value.get()));
};
```

- The original type comparison against **"string"** has now been replaced by a function argument. This way, the contents of the **Identity** value can be compared against any type you like. Also, the function accepts a **transform** function that modifies the end result if it's an **Identity** instance.

8. Now, write each of the functions and ensure that they are all exported:

sentence.js

```
1 let validateString = function(value) {
2   return process(value, (v) => typeof v == "string", (v) => v);
3 };
4 let splitStringIntoList = function(value) {
5   return process(
6     value,
7     (v) => typeof v == "string",
8     (v) => v.split(" ")
9   );
10 };
11 let rejectEmptyStrings = function(value) {
12   return process(
13     value,
14     (v) => Array.isArray(v),
15     (v) => v.filter((str) => str.length > 0)
```

<http://packt.live/33H7ei0>

Notice how similar each of the functions are. The differing aspect is mainly the **transform** function, which ensures that the returned Identity is processed.

9. Now, run the tests. If all went well, they should all pass.

The preceding solution follows the methodologies of TDD, **Keep It Simple, Stupid (KISS)**, and **Don't Repeat Yourself (DRY)**. By refining, improving, and thinking, you can ensure your code is clear, concise, and bug-free.

