Image Classification on React Native with TensorFlow.js and MobileNet

Aman Mittal
Oct 17, 2019 · 7 min read

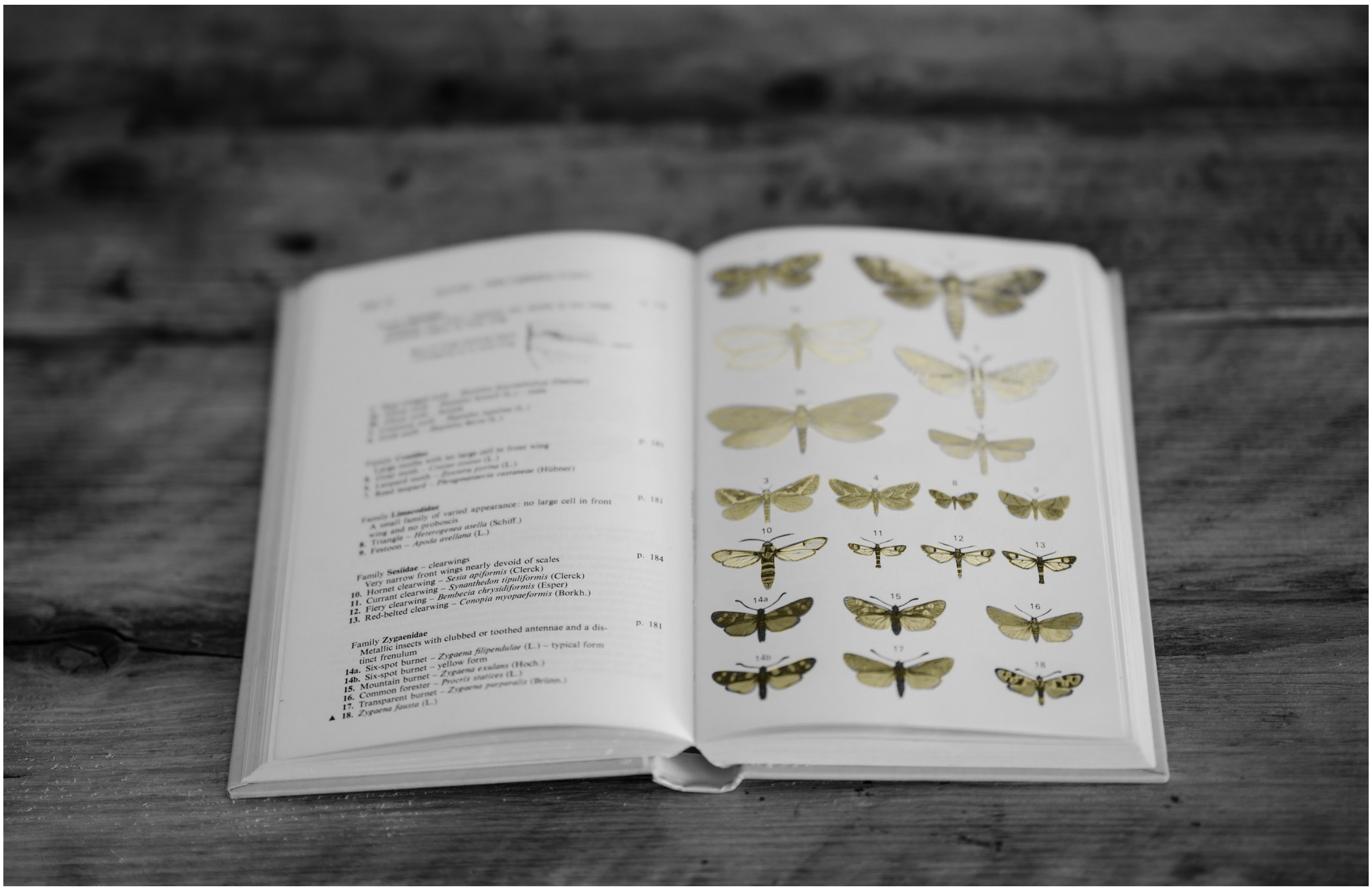


Photo by Annie Spratt on Unsplash

Recently, the alpha versions of **TensorFlow.js** for React Native and Expo applications were released. It currently allows developers to load pre-trained models and to train new models. Here's the announcement tweet:

TensorFlow
@TensorFlow

Attention React Native developers and

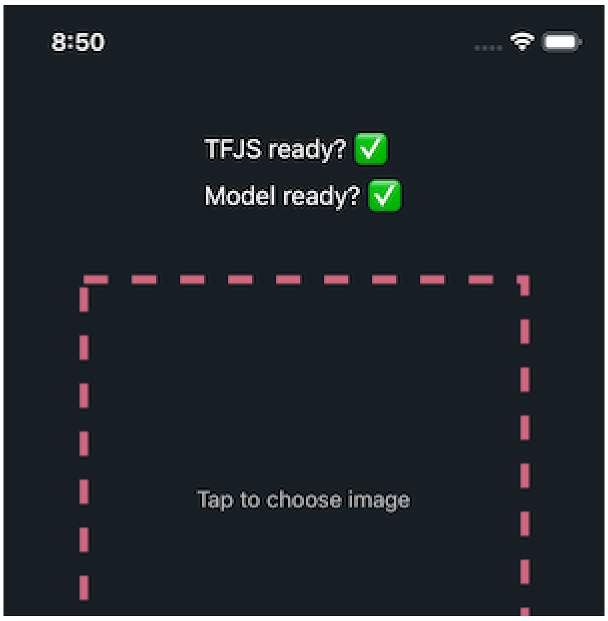
Attention React Native developers and enthusiasts!

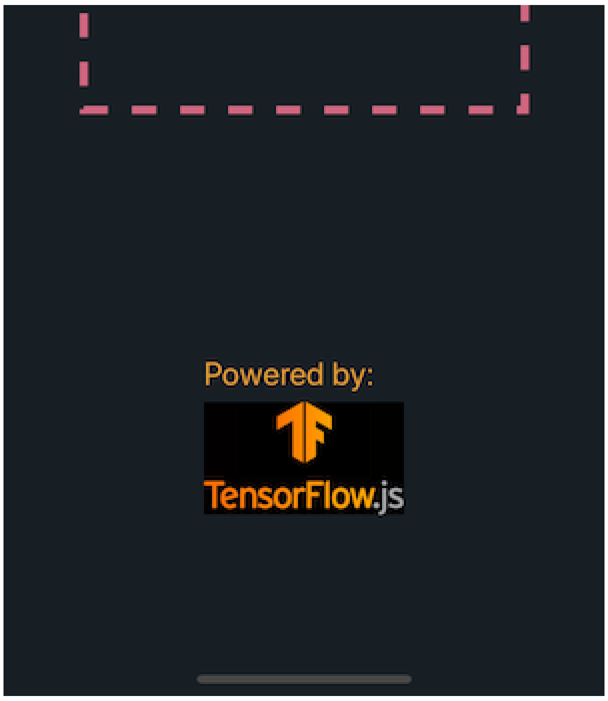
Check out the alpha release of TensorFlow.js for React Native and try out all the capabilities such as loading pre-trained models and training on device. Feedback is welcome!

Details here → goo.gle/2lwGuQh

TensorFlow.js provides many pre-trained models that simplify the time-consuming task of training a new machine learning model from scratch. In this tutorial, we're going to explore TensorFlow.js and the MobileNet pre-trained model architecture to classify input images in a React Native mobile application.

By the end of this tutorial, the app will look like the following:





> *Sidenote: Previously, I did dip my hands into Google's Vision API to build an image classification app that determines whether a given image is a hotdog or not. If you're interested in reading that example, please follow the link below:*

Build a "Not Hotdog" clone with React Native
Image classification with Google's Vision API
heartbeat.fritz.ai

Here's the link to the complete code in a **GitHub repo, for your reference**.

. . .

Mobile machine learning is among the most cutting-edge and fastest-growing technologies. But how does it work, and what can you actually build with it? We wrote a new (free) ebook to help tackle these questions head on.

. . .

Table of Contents

. . .

Requirements

- Nodejs >= 10.x.x installed on your local dev environment
- `expo-cli`
- Expo Client app for Android or iOS, used for testing the app

. . .

Integrating TF.js in an Expo app

To start using the TensorFlow library in React Native, the initial step is to integrate the platform adapter. The module `tfjs-react-native` is the platform adapter that supports loading all major tfjs models from the web. It also provides GPU support using `expo-gl`.

Open the terminal window and create a new Expo app by executing the command below.

```
expo init mobilenet-tfjs-expo
```

block1.shell hosted with ❤ by **GitHub** **view raw**

Next, make sure to generate an Expo managed app. Then navigate inside the app directory and install the following dependencies:

```
e @tensorflow/tfjs @tensorflow/tfjs-react-native expo-gl @tensorflow-models/mobilenet jpeg-js
```

block2.shell hosted with ❤ by **GitHub** **view raw**

> *Note: If you'd like to use `react-native-cli` to generate an app, you can follow the clear instructions to modify the `metro.config.js` file and other necessary steps, explained here.*

Even though you're using Expo, it's still necessary to install async-storage, as the tfjs module depends on that.

. . .

Testing the TF.js integration

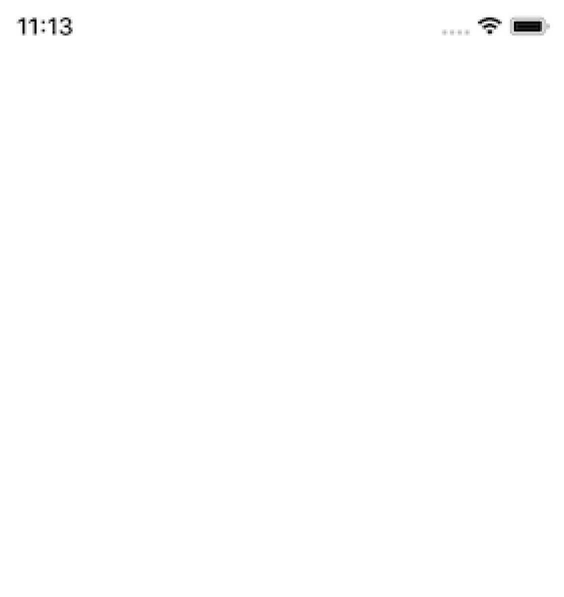
Before we move on, let's test to ensure that tfjs is successfully loaded into the app before the app is rendered. There's an asynchronous function to do so called `tf.ready()`. Open the `App.js` file, import the necessary dependencies, and define an initial state `isTfReady` with a boolean false.

```
import React from 'react'
import { StyleSheet, Text, View } from 'react-native'
import * as tf from '@tensorflow/tfjs'
import { fetch } from '@tensorflow/tfjs-react-native'

class App extends React.Component {
  state = {
    isTfReady: false
  }

  async componentDidMount() {
    await tf.ready()
    this.setState({
      isTfReady: true
    })

    //Output in Expo console
    console.log(this.state.isTfReady)
  }

  render() {
    return (
      <View style={styles.container}>
        <Text>TFJS ready? {this.state.isTfReady ? <Text>Yes</Text> : ''}</Text>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center'
  }
})

export default App
```

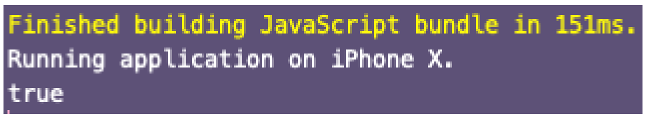
block3.js hosted with ❤ by **GitHub** **view raw**

Since the lifecycle method is asynchronous, it will only update the value of `isTfReady` to true when tfjs is actually loaded.

You can see the output in the simulator device, as shown below.



Or in the console, if using the `console` statement.



```
Finished building JavaScript bundle in 151ms.
Running application on iPhone X.
true
```

. . .

> Subscribe to the Fritz AI Newsletter for inside looks at all things mobile machine learning: tackle common challenges, explore its value, and learn how it can help scale your business.

. . .

Loading the MobileNet model

Similar to the previous section, you have to load the MobileNet model as well before providing the input image. Loading a pre-trained TensorFlow.js model from the web is an expensive network call and will take a good amount of time. Modify the `App.js` file to load the MobileNet model. Start by importing it:

Next, add another property to the initial state:

Then, modify the lifecycle method:

Lastly, let’s display an indicator on the screen when the loading of the model is complete.

When the model is being loaded, it will display the following message.

When the MobileNet model is loaded, you’ll get the following output.

· · ·

Asking user permissions

Now that both the platform adapter and the model are currently integrated in the React Native app, we need to add an asynchronous function to ask for the user’s permission to allow access to the camera roll. This is a mandatory step when building iOS applications using the image picker component from Expo.

Before proceeding, run the following command to install all the packages provided by the Expo SDK.

Next, add the following import statements in the `App.js` file:

In the `App` class component, add the following method:

Lastly, call this asynchronous method inside `componentDidMount()`:

. . .

Convert a raw image into a Tensor

The application will require the user to upload an image from their phone's camera roll or gallery. You have to add a handler method that loads the image and allows TensorFlow to decode the data from the image. TensorFlow supports JPEG and PNG formats.

In the `App.js` file, start by importing the `jpeg-js` package, which will be used to decode the data from the image.

It decodes the width, height, and binary data from the image inside the handler method `imageToTensor`, which accepts a parameter of the raw image data.

The `TO_UINT8ARRAY` array represents an array of 8-bit unsigned integers. The constructor method `Uint8Array()` is the **new ES2017 syntax**. There are different kinds of typed arrays, each having its own byte range in the memory.

. . .

Load and classify the image

Next, we add another handler method called `classifyImage` that will read the raw data from an image and yield results upon classification in the form of `predictions`.

The image is going to be read from a source, so the path to that image source has to be saved in the `state` of the app component. Similarly, the results yielded by this asynchronous method have to be saved, too. Modify the existing state in the `App.js` file for the final time.

Next, add the asynchronous method.

The results from the pre-trained model are yielded in an array. An example is shown below:

. . .

Allow user to pick the image

To select an image from the device's camera roll using the system's UI, you're going to use the asynchronous method `ImagePicker.launchImageLibraryAsync` provided by the package `expo-image-picker`. Import the package itself:

Next, add a handler method called `selectImage` that will be responsible for:

- letting the image be selected by the user
- populate the source URI object in the `state.image`—if the image selection process isn't canceled
- and lastly, invoke `classifyImage()` method to make predictions from the given input

The package `expo-image-picker` returns an object. In case the user cancels the process of picking an image, the image picker module will return a single property: `canceled: true`. If successful, the image picker module returns properties such as the `uri` of the image itself. That’s why the `if` statement in the above snippet holds so much significance.

. . .

Run the app

To complete this demonstration app, you’ll need to add touchable opacity to where the user will click to add an image.

Here’s the complete snippet of the `render` method in the `App.js` file:

And here’s the list of the complete `styles` object:

Run the application by executing the `expo start` command from a terminal window. The first thing you’ll notice is that upon bootstrapping the app in the Expo client, it will ask for permissions.

Then, once the model is ready, it will display the text “Tap to choose image” inside the box. Select an image to see the results.

Predicting results can take some time. Here are the results of the previously-selected image.

• • •

Conclusion

I hope this post serves the purpose of giving you a head start in understanding how to implement a TesnorFlow.js model in a React Native app, as well as a better understanding of image classification, a core use case in computer vision-based machine learning.

Since the TF.js for React Native is in alpha at the time of writing this post, we can hope to see many more advanced examples in the future to build real-time applications.

Here are some resources that I find extremely useful.

- tfjs-react-native GitHub repo that contains more examples using different pre-trained models
- Infinite Red's NSFW JS and React Native example is crisp and very helpful
- Introduction to Tensorflow.js

Here's the link to the complete code in a **GitHub repo, for your reference**.

. . .

Editor's Note: ***Heartbeat*** *is a contributor-driven online publication and community dedicated to exploring the emerging intersection of mobile app development and machine learning. We're committed to supporting and inspiring developers and engineers from all walks of life.*

Editorially independent, Heartbeat is sponsored and published by ***Fritz AI****, the machine learning platform that helps developers teach devices to see, hear, sense, and think. We pay our contributors, and we don't sell ads.*

If you'd like to contribute, head on over to our ***call for contributors****. You can also sign up to receive our weekly newsletters (****Deep Learning Weekly*** *and the* ***Fritz AI Newsletter****), join us on* ***Slack****, and follow Fritz AI on* ***Twitter*** *for all the latest in mobile machine learning.*

Thanks to Austin Kodra.

React Native Heartbeat Machine Learning JavaScript Mobile Development