A Progressive Web Application for Bird Classification

Final Project Report

TU858

BSc (Hons) in Computer Science

Robin Huumonen

Michael Collins

School of Computing

Technological University Dublin

26.4.2021

Abstract

This project report details the process of designing and developing a progressive web application (PWA) for bird species classification. The goal of this project is to provide full offline classification capabilities once the classifier has been stored in client-side storage.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

Robin Huumonen

7.3.2021

Acknowledgements

I would like to thank Drs Michael Collins and Emma Murphy for their guidance in project management, academic writing and studies in general.

Table of Contents

1. Introduction

1.1. Overview and Background

Progressive web applications have been here for a while now, whether we have noticed them or not. Companies such as Twitter, Starbucks, Uber and Washington Post have PWAs accompanying native apps or have completely switched over. One of the major benefits of PWAs is that mobile users can add them to their home screen among native apps. The difference is that PWAs are still web applications, but with their features they can be developed to resemble native apps. With different caching strategies enabled by the use of a Service Worker, PWAs can be developed to have different levels of offline capabilities (Tamire, 2019). This feature is used to make this project a classification tool to be used in places where there might not be reliable internet connection.

According to John White's sourced blog post, bird watching, also known as birding, is one of the fastest growing hobbies in North America. In fact, Canadians spent more time birding than gardening (White, 2019). There are many available groups, clubs, books and apps for the hobbyists. Many apps provide help to locate and identify bird species. Some apps can classify birds with automatic image or sound recognition, others can help to recognize by asking features of the observed bird and narrowing down one's options (Scott, 2021).

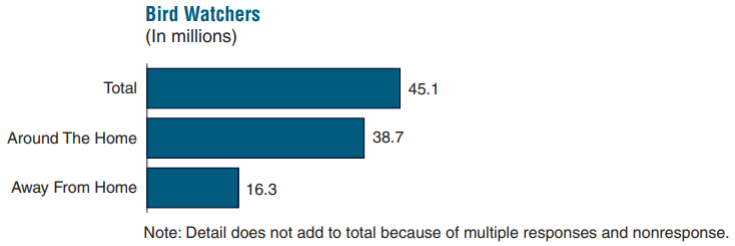


Figure 1.1: Bird watching participants in the USA (U.S. Census Bureau, 2016).

Classifications will be done with Google's open source TensorFlow (TF), a platform for building and running deep learning models. Models are trained with machine learning algorithms; they encapsulate all the information required to make predictions of the data that is run through it (Brownlee, 2019b). TF also provides a JavaScript (JS) version that can be used in a web browser or Node.js environment. This project will run a TF.js

model in a web browser to make client-side offline classification possible. On-device computation is great for data privacy and accessibility (Smilkov et al., 2019)

1.2. Project Objectives

The objective of this project is to develop and deploy a PWA that is capable of on-device classification of bird species. The aim is to train an accurate multinomial classification model with the data set selected in the research process. As for the web development of this project, the goal is to include enough PWA features such that the application will be identified as progressive by web browsers and Lighthouse audit.

1.3. Structure of the Document

Research	In this chapter following technologies are selected and introduced: React, Express and Keras. Furthermore, MobileNetV2 convolutional neural network (CNN) architecture is introduced by explaining tensors, 2d convolution, depthwise separable convolution, inverted residual and linear bottleneck layer.
Design	Design chapter shows the design of the following: • Modified Kanban project management methodology • Web stack and offline-first caching strategy • Low fidelity prototype and screenshots of the final design • Use cases in a model
Development	Development chapter explains the development process and shows snippets of the main components of the following parts of the architecture: • React front end with styled-components and conditional rendering • Express back end • Classifier trained in Google Colab
System Validation	The System is validated with test cases that test the system as a whole and the prediction accuracy of the classifier with different species. The application is audited with Lighthouse tool against PWA checks. Finally, links to two demonstration videos are given and screenshots of them presented.
Project Development	This chapter explains what was changed and omitted during the project's development and reasons for them.
Conclusion	This chapter gives suggestions to future development possibilities.
Appendix	This chapter has the file that renders all the content in the front end and most of the Keras training script.

2. Research

2.1. Overview of Technologies

Research for developing a PWA started with choosing a front end framework. React was chosen, because starting development environment using integrated tool chain Create React App with built-in PWA template is a great starting point for developing a PWA with offline-first caching (Timothy, 2020). It initiates one of the core features of PWAs, a Service Worker, that is a script that runs in the background in a separate thread. It enables different caching strategies by intercepting request-response flow (Tamire, 2019). Also, a web app manifest file will be written. It is a file that gives information about the PWA so that it can be 'installed' on a user's device. A typical manifest file includes the app name, icon and the URL that should be opened when the icon is clicked (Beaufort, 2021).

For the back end Express framework for Node.js was chosen, because it is a minimalist framework, and the author is familiar with it. Server's role in this project is minimal. It will only serve the website's code and the TF.js model. Nothing will be uploaded to the server. The web page assets, such as JS and HTML, will be stored to a browser's cache using Cache API. The classification model will be stored with IndexedDB API, an API for NoSQL database in a browser. The image that is about to be classified remains on-device and classification is performed in-browser. The final version of the application will be deployed on Heroku, that is a cloud platform as a service. Deploying means that the application will be usable through the Internet.

After PWA development research TF.js was researched, mainly from (Smilkov et al., 2019). TF.js is an open source library for building and running machine learning algorithms in a web browser or Node.js environment. This project will run a TF.js model in a web browser to allow offline classification. TF.js has a model converter that can convert pre-trained TensorFlow or Keras models into TF.js web format. The model for the project was chosen to be developed with Keras, which is a more user-friendly and easy to use high level Python API for TensorFlow (Gadicherla, 2020).

Data set was searched from Kaggle, a community for people sharing data sets and code to perform machine learning. The selected data set is of 250 bird species, with 35

215 training images, 1250 test and validation images (5 per species). Training images' distribution of images per species is uneven, but at least 100 training images per species is assured, and the creator of the data set states that he has been able to develop an accurate classifier despite this. Another imbalance of this data set is with the sexes. 80 % of the images are from male birds. Male birds typically are more brightly colored than their female counterparts, thus they may look very different. The creator warns that due to this a classifier trained with this data set may not perform as well on female birds. (Piosenka, 2021).

Finally different CNN architectures were researched. Google's MobileNetV2 was chosen because it is designed to be very small and have low latency, especially with mobile and embedded vision applications in mind. MobileNets are nearly as accurate as the well-known VGG16, and the first version is 32 times smaller and 27 times less compute intensive (A. G. Howard et al., 2017). Newer MobileNet retains the same accuracy while being faster and having less parameters, which leads to smaller size.

2.2. MobileNetV2

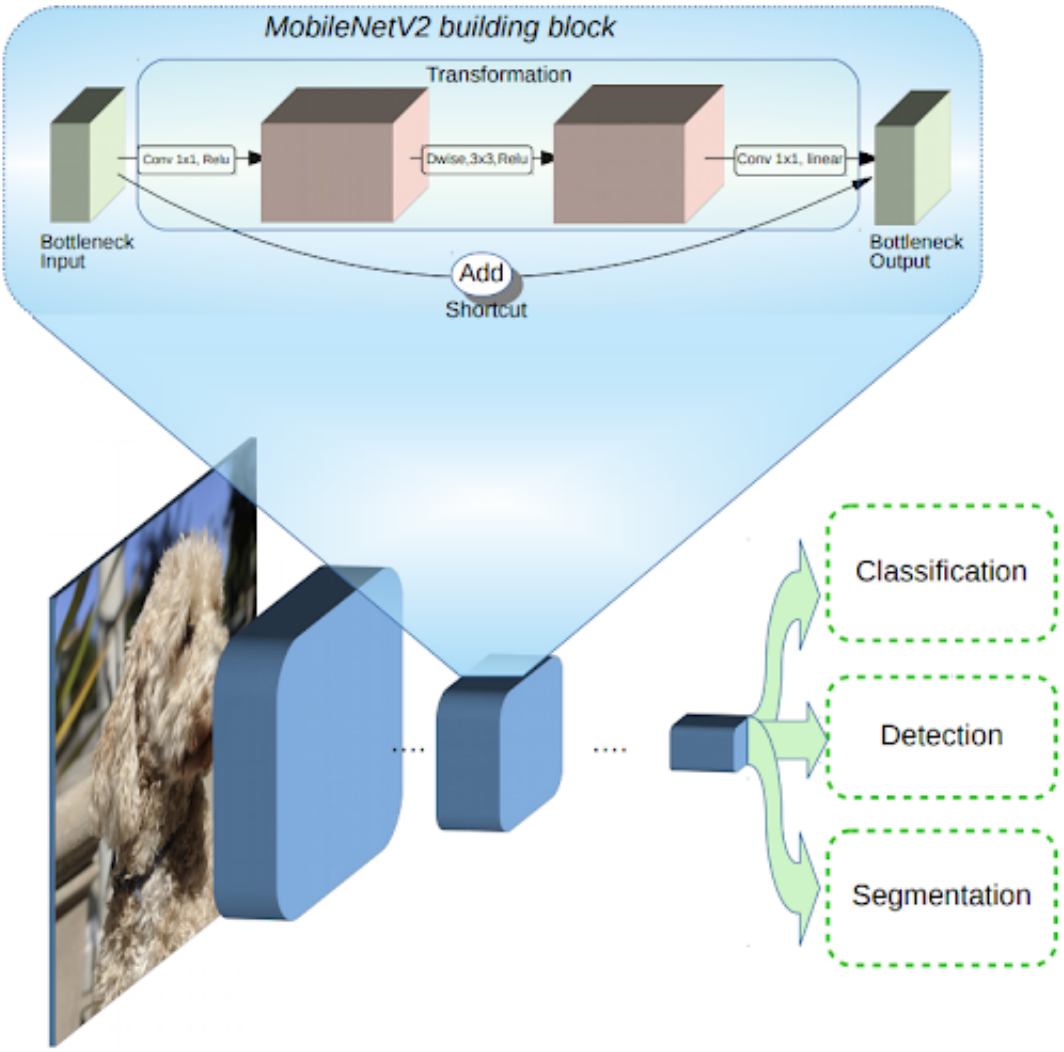
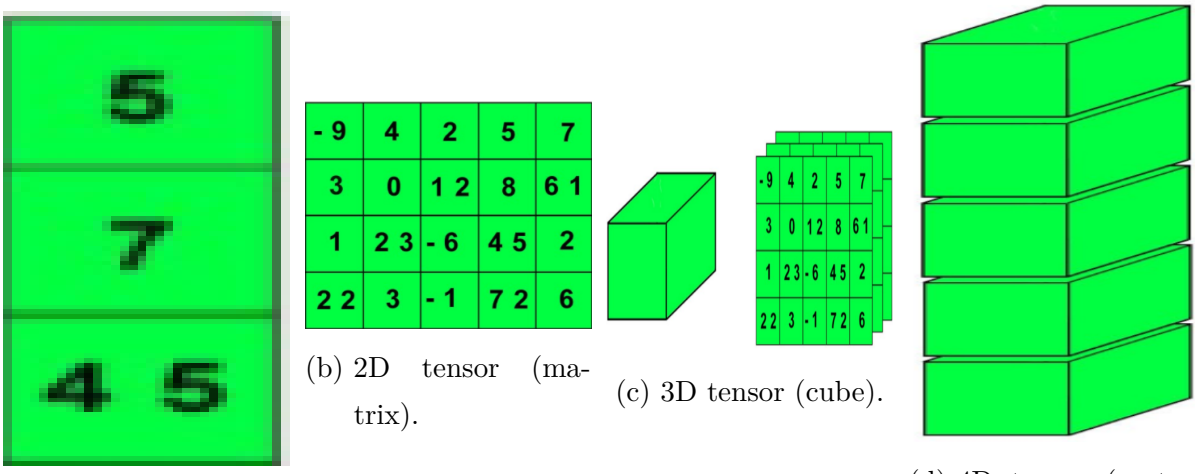


Figure 2.1: Overview of MobilenetV2 architecture (M. S. A. Howard, 2018).

To get the idea of MobileNetV2 architecture (Figure 2.1) following concepts will be introduced: tensor, 2d convolution, depthwise separable convolution, inverted residual and linear bottleneck layer. Where the last two were introduced in MobileNetV2 as its building block.



(a) 1D tensor (vector).

(b) 2D tensor (matrix).

(c) 3D tensor (cube).

(d) 4D tensor (vector of cubes).

Figure 2.2: Tensors of 1, 2, 3 and 4 dimensions (Jeffries, 2019).

Simple way to think about tensors is that they are data structures for any kind of data. Tensors can be represented in programming with arrays of N dimensions. A color image is represented in a 3D tensor (Figure 2.2c). Matrices (Figure 2.2b) representing the image are of the height and width of the image in pixels, i.e., each element has a corresponding pixel. A pixel is the smallest unit of an image. The numerical value in an element is the color intensity of its corresponding pixel. Color image has 3 matrices, one for each color channel: red, green and blue (Bortolossi, n.d.). A set of images is represented as a 4D tensor, i.e., an array of images (Figure 2.2d). This is the data type fed into the CNN (Jeffries, 2019).

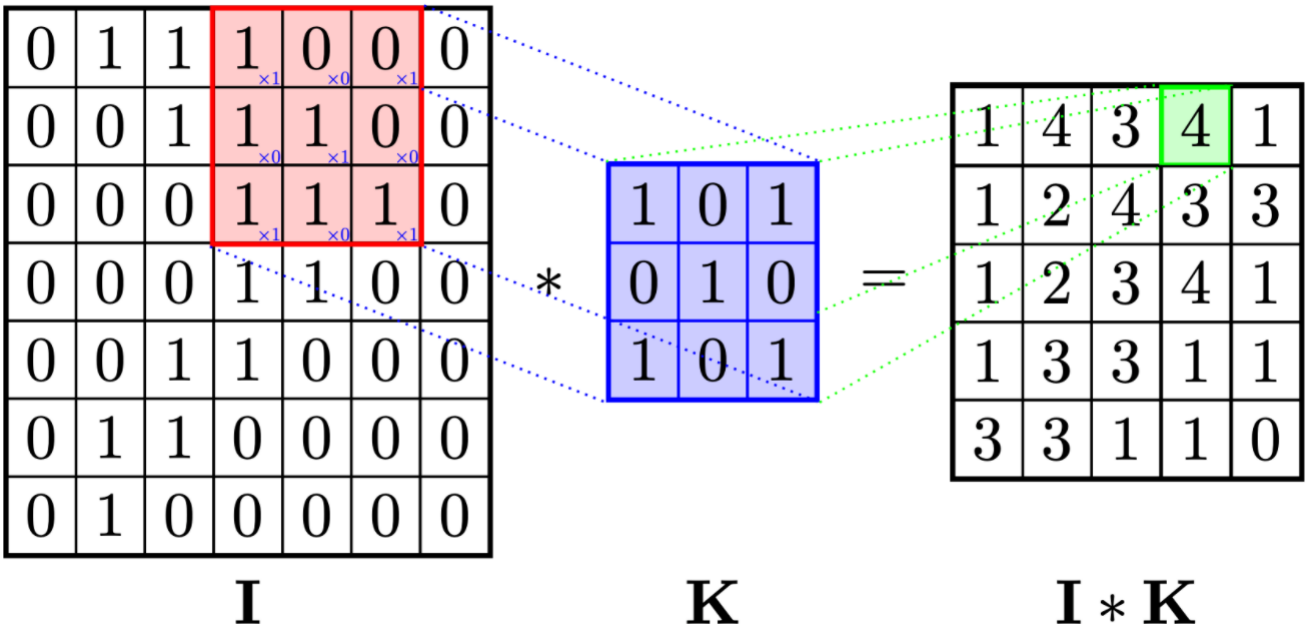
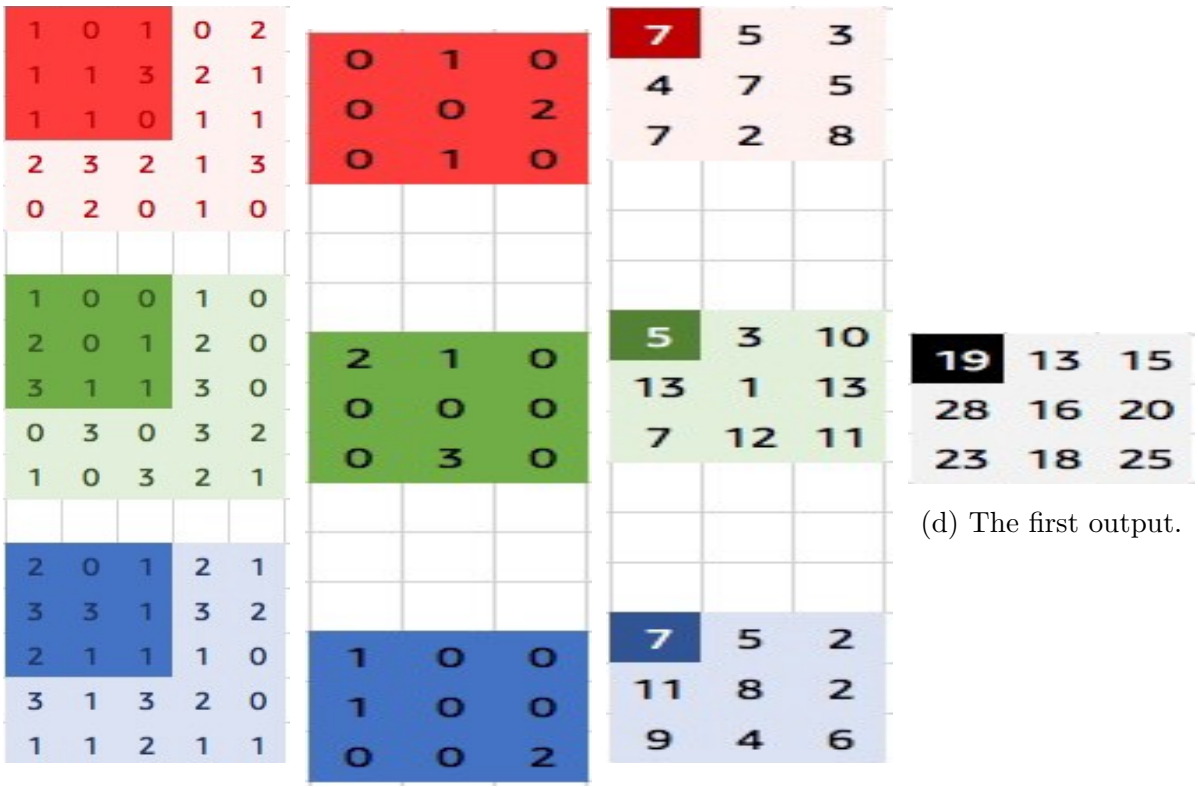


Figure 2.3: Matrix convolution (*Visualizing matrix convolution,* 2019).

Convolution is used in deep learning to extract features from an input image. It is element-wise multiplication and addition operation of tensors to produce an output tensor (Figure 2.3). The width and height of the multiplicand and the multiplier are usually different, this means that the multiplier must be moved through the multiplicand to go through all of its elements.



(a) RGB input channels.

(b) RGB kernels.

(c) Intermediate outputs.

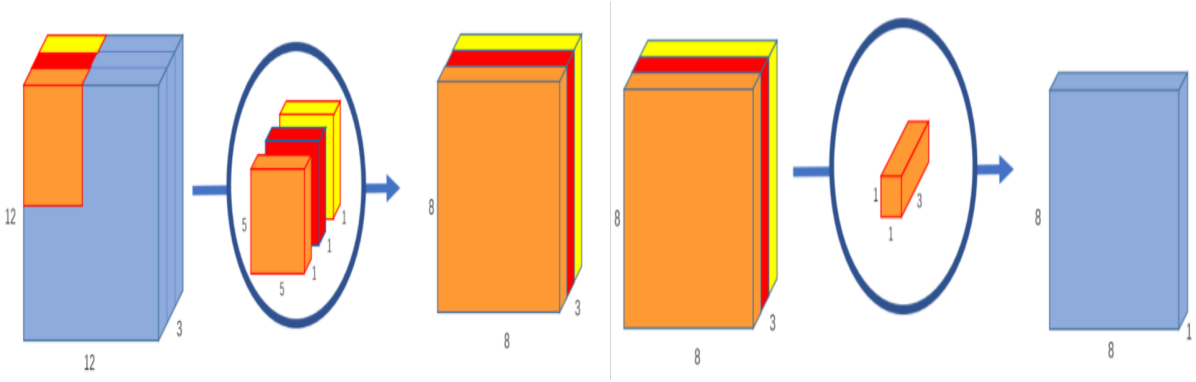
(d) The first output.

Figure 2.4: Illustration of multichannel 2D convolution (Lane, 2018).

In deep learning parlance, 2D and 3D tensors of weights are called kernel and filter. Weights are the parameters that the CNN is optimizing to extract enough information of the input to make predictions about it. 2D convolution means that the kernel is moved through the input in 2 directions, thus the depth (number of matrices) of the filter and input must match (Figure 2.4), otherwise the kernel must be moved in 3 directions, when filter depth $<$ input depth, to go through all of the input's elements. The output of a multichannel 2D convolution operation for an RGB image is a matrix representing a greyscale image (Figure 2.4d). It is the sum of the intermediate outputs (Figure 2.4c). (Bai, 2019).

If one wants to increase the number of channels in the output image, one needs to increase the number of filters. This can become computationally heavy. Figure 2.4 shows the calculation of the first element of the output image using one 3-kernel filter. The MobileNetV2 initial fully convolution layer has 32 filters (Sandler et al., 2018).

MobileNets started to use depthwise separable convolution to reduce computation (A. G. Howard et al., 2017). Depthwise separable convolution separates normal convolution into 2 parts: a depthwise convolution and a pointwise convolution (Wang, 2018).



(a) Depthwise convolution. (b) Pointwise convolution.

Figure 2.5: Depthwise separable convolution (Wang, 2018).

Depthwise convolution (Figure 2.5a) is similar to what was showed in multichannel convolution (Figure 2.4), but the output will have the same number of channels as the input and filter. Convolution is done for each filter's kernel with one and only one of the image's channels. Pointwise convolution (Figure 2.5b) is done with the output of the depthwise. Its kernel size is 1×1, and filter depth = input depth. The output of this operation can be elongated by increasing the filters (Figure 2.6). In normal convolution the input image is transformed as many times as there are filters, whereas in separable convolution the image is transformed once in the depthwise step and lengthened with the pointwise step. (Wang, 2018).

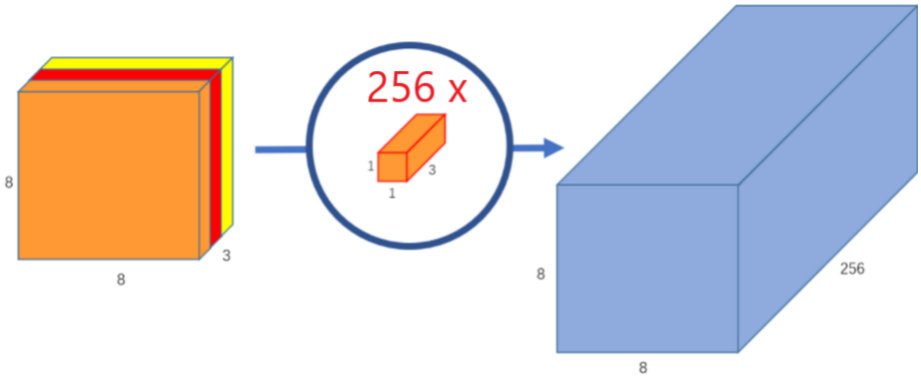


Figure 2.6: Pointwise convolution with 256 filters (Wang, 2018).

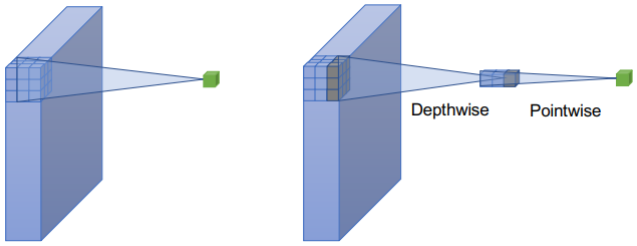


Figure 2.7: Normal vs. depthwise separable convolution (Guo et al., 2019).

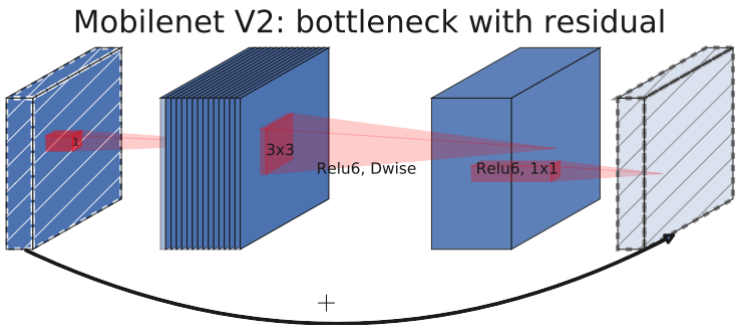


Figure 2.8: Inverted residual block (A. Howard et al., 2019).

MobileNetV2's building block (Figure 2.8, cf. Figure 2.1) follows a narrow layer ⇒ wide ⇒ narrow approach, where narrow refers to pointwise convolution and wide to depthwise. A skip connection connects the first and the last layers, the linear bottlenecks (Pröve, 2018). This allows the network to access earlier activations that were not modified in the block. Activations are the values of activation functions, which decide whether to fire a

neuron. The block uses rectified linear unit (ReLU) types as activation functions. It is simply $y = \max(0, x)$, meaning it will output the input directly if it is positive, otherwise zero (Liu, 2017). ReLU6 is defined as $y = \min(\max(0, x), 6)$, i.e., the maximum output is 6. The bottleneck layer is a layer with fewer neurons than the layer following or preceding it. These layers are used to reduce the number of feature maps, i.e., channels in the CNN, meaning the tensorial feature representations are compressed (Slater, 2017). In research for MobileNetV2 introduction paper their experimental evidence suggested that inserting linear bottleneck layers into the convolutional blocks "prevents nonlinearities from destroying too much information" (Pröve, 2018).

3. Design

3.1. Methodology

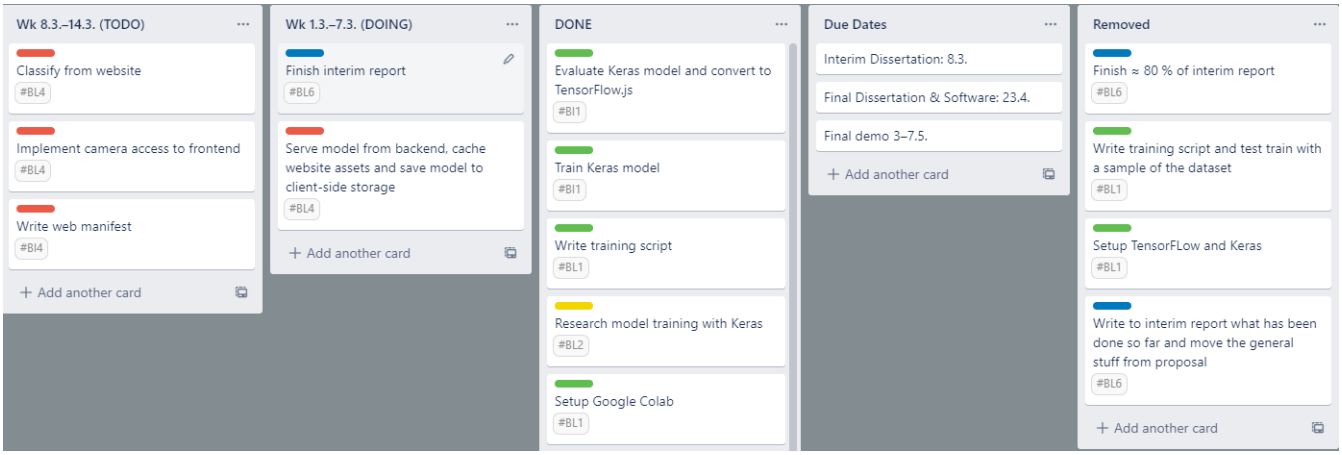


Figure 3.1: Modified Kanban board.

This project was done using author's own version of Kanban methodology. Kanban is a simple agile methodology, which has a board with a three-step workflow: To Do, Doing and Done (RADIGAN, n.d.). High-level tasks are told as stories in Trello. Stories were divided into subtasks, that can be in a To Do, Doing, Done or Removed state. Git was used for version control and GitHub for remote repositories. There is one individual repository for the frontend, backend and classifier. These are available from: frontend, backend and classifier repositories.

3.2. Architecture

Figure 3.2: Web stack.

Figure 3.2 shows the web technologies used in this project that were introduced in the Research chapter.

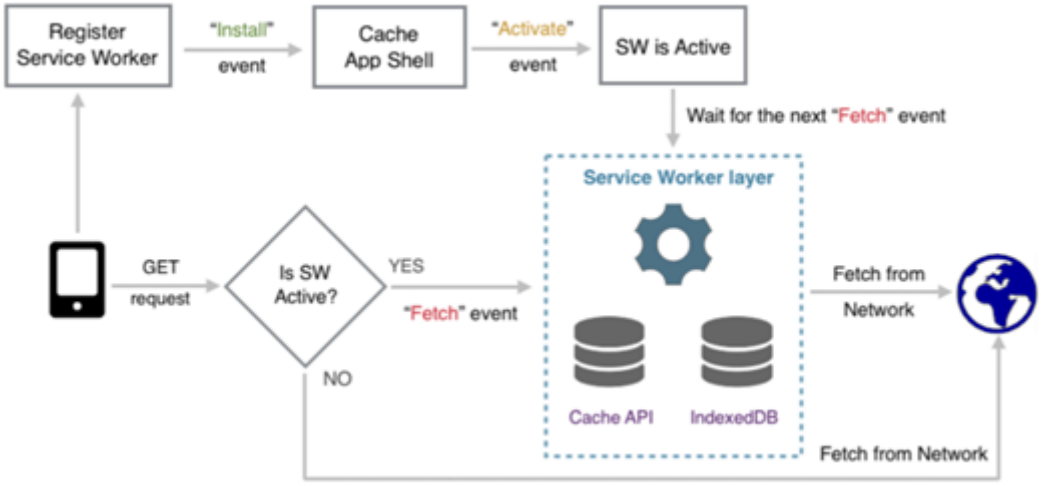


Figure 3.3: Service Worker's flow for "cache then network" strategy (Idakiev, 2016).

This project uses cache then network, or offline-first, caching strategy (Figure 3.3). All static site assets, such as JS, HTML, CSS and image files, that are part of the bundled web application, are cached. Thus, the application loads faster on subsequent visits, and will work regardless of the network state (Timothy, 2020). An offline-first Service Worker checks whether requested files are in the cache or client-side database first before requesting them from the server. With Cache API one can cache static assets, and with IndexedDB, one can store almost any kind of data.

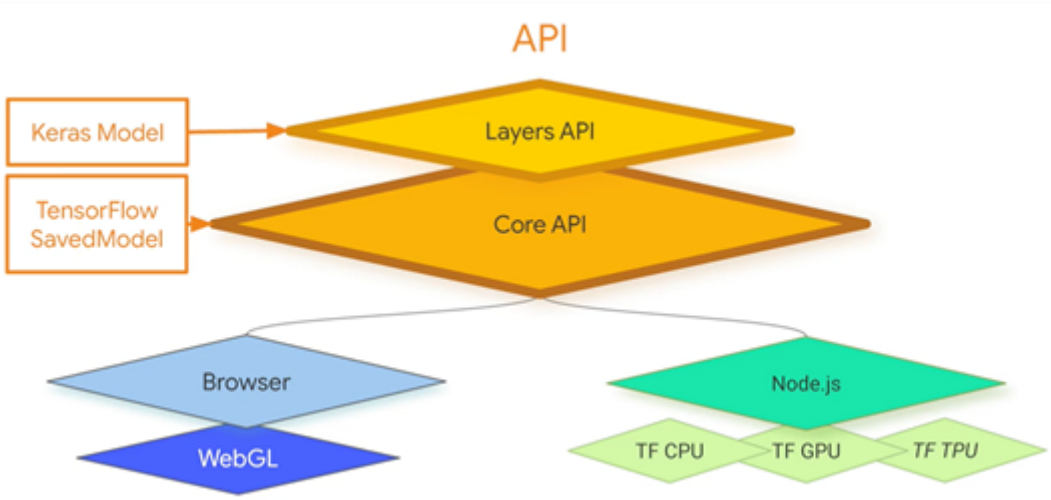


Figure 3.4: TF.js architecture (Tirth1306, 2020).

In this project TF.js will be used in a browser (Figure 3.4). When TF.js is run in a browser, it utilizes the GPU of the device via Web Graphics Library API. TF.js offers a layers API for Keras users, that is developed to be as similar as possible to Keras (TensorFlow, n.d.).

3.3. UI

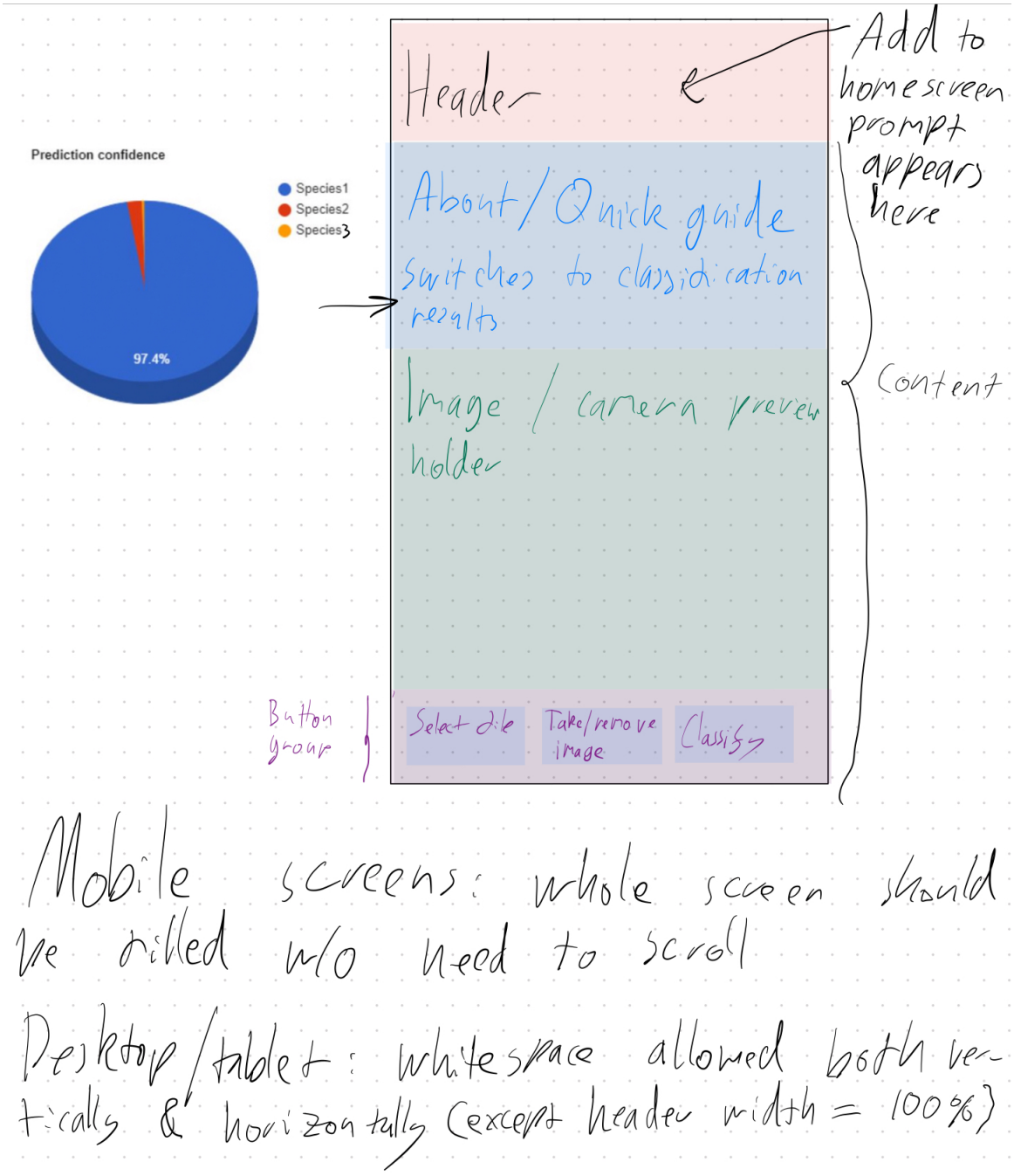
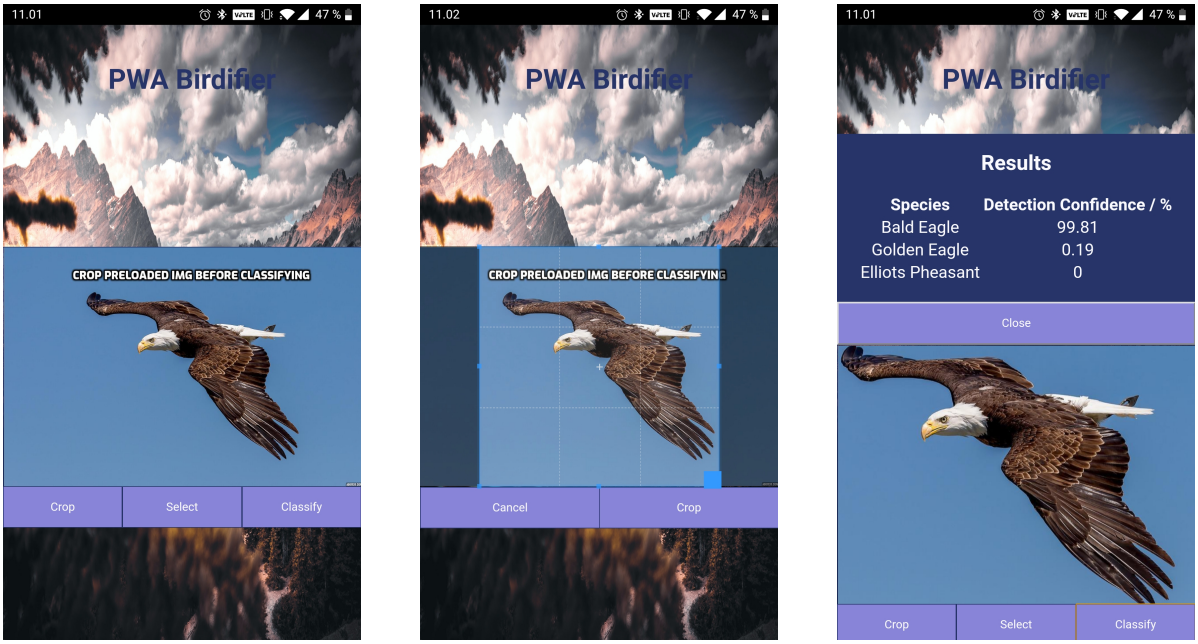


Figure 3.5: Low fidelity prototype.

The UI's low fidelity prototype (Figure 3.5) was designed mobile-first. The React front end is a single page application (SPA), meaning the current page is updated dynamically

piece-by-piece instead of updating the whole page. About nor quick guide portions were not implemented due to the application's simplicity.



(a) Home. (b) Image cropper. (c) Classification results.

Figure 3.6: UI screens.

Figure 3.6 shows the final design of the UI.

3.4. Features

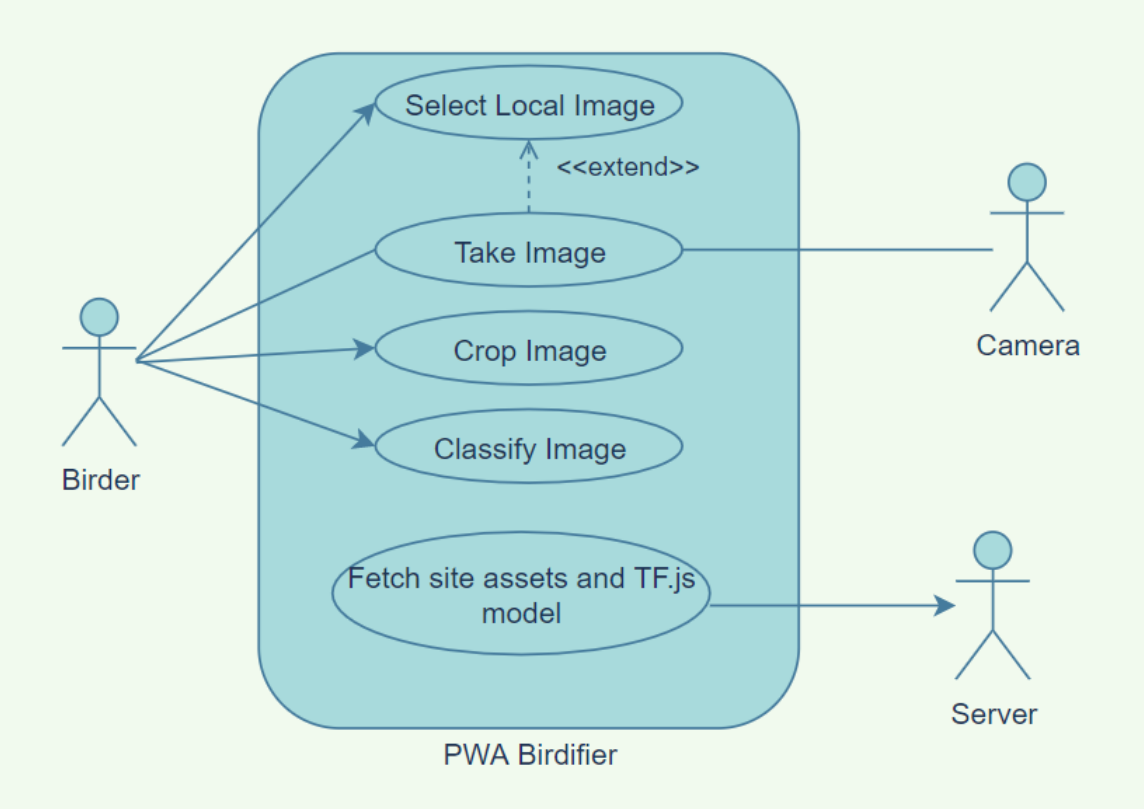


Figure 3.7: Use cases.

The application holds one image in cache that can be classified. The classifiable image can be the preloaded and cropped one, selected local image or just taken image. Server is thought as an external actor (Figure 3.7), because after the initial visit the website assets and TF.js model are stored into the system, making it an independent entity. Taking an image extends selecting a local image, because selecting the image was done with `<input type="file">`, which offers an option to take an image with a mobile device instead of selecting one.

4. Development

4.1. Front End

```
function App() {
  useEffect(() => {
    document.title = "PWA Birdifier";
  }, []);

  return (
    <PageWrap>
    <Header/>
    <Content/>
    </PageWrap>
  );
}
```

Figure 4.1: Snippet of front end's App.js.

Front end development environment was booted up with *npx create-react-app myapp –template cra-template-pw* (Timothy, 2020) to get a Service Worker. As shown in the Design chapter the page structure is simple. React components are used to split the UI into independent pieces of code. In Figure 4.1 `App()` is the root component that imports all the other components. `<PageWrap>` is just a div element with a custom style.

```
import styled from "styled-components";

export const ContentWrap = styled.div`
  display: flex;
  flex-direction: column;
  width: 100%;
  margin: auto;

  div {
    flex: 1;
  }

  @media(min-width: 1024px) {
    width: 40rem;
    margin:0 auto;
  }
```

Figure 4.2: Snippet of ContentStyles.js.

Most components are styled with styled-components package as shown in Figure 4.2, others are styled with inline styles, e.g., `<CustomComponent style={{ color: "red" }}/>`.

```
ReactDOM.render(
  <React.StrictMode>
  <App />
  </React.StrictMode>,
  document.getElementById("root")
);
serviceWorkerRegistration.register();
```

Figure 4.3: Snippet of front end's Index.js.

The root component (Figure 4.1) is rendered, i.e., transformed for presentation, in the index.js file (Figure 4.3), which is analogous to an index.html file. The Service Worker

is also registered with ServiceWorkerRegistration interface here.

```
return (
  <ContentWrap>
    {renderResults === true ?
      <Results />
    : null}
    {renderSpinner === true ?
      <Orbitals />
    : null}
    {cropImage === true ?
      <CropImage />
    : <div className="currentImage">
        <img src={imgSrc}></img>
      </div>}
```

Figure 4.4: Snippet of Content.js.

The content (Figure 4.4 and appendix A.1) section contains everything but the heading of the application. The component is rendered conditionally, meaning a different view is returned depending on the state of the component. Conditional rendering was done with useState hooks and ternary operators, of which syntax:

`condition ? exprIfTrue : exprIfFalse.`

Local image cropping was made with react-cropper (Kuanghuei, 2021), and code for it was adjusted from their working examples. Top three classification results are displayed in a HTML table. The `<input>` element for an image is hidden from an user, and it is used with a HTML button instead, thus it can be and was made a part of a button group.

```
async function saveModel(model) {
  if (model) {
    await model.save("indexeddb://my-model");
  }
}

async function loadModel() {
  try {
    setModel(await tf.loadLayersModel("indexeddb://my-model"));
  } catch (error) {
  } finally {
    setModel(await tf.loadLayersModel(modelDir));
    if (window.indexedDB ) {
      saveModel(model);
    }
  }
}

useEffect(() => {
loadModel();
}, []);
```

Figure 4.5: Loading and saving the classifier.

The classification model is sent from the server along with other page assets. The model is loaded using useEffect hook (Figure 4.5). The empty array argument sets the hook to be executed only during the initial render of the component.

4.2. Back End

```
const app = require("./app");
const http = require("http");
const config = require("./utils/config");
const logger = require("./utils/logger");

const server = http.createServer(app);

server.listen(config.PORT, () => {
  logger.info(`Server running on port ${config.PORT}`);
});
```

Figure 4.6: Snippet of back end's Index.js.

The Express back end (Figure 4.6) is created with `createServer` method, where `app` is an instance of Express.

```
const express = require("express");
const app = express();

app.use(express.static(path.join(__dirname, "build")));

app.get("/*", function (req, res) {
  res.sendFile(path.join(__dirname, "build", "index.html"));
});

module.exports = app;
```

Figure 4.7: Snippet of back end's App.js.

The app (Figure 4.7) serves static files from the React front end's production build, where all the JavaScript code is minified and merged into one file. All GET requests are responded with the index.html.

The whole application, which includes the React production build and the back end, was deployed on to Heroku using Heroku CLI and GIT. It is a remote repository created with `heroku create`, and the application can be pushed into it with `git push heroku master`. The back end also has a Procfile, that tells Heroku how to start the application. The Procfile has only one line: web: node index.js.

4.3. Classifier

The bird classifier was developed in a Google Colaboratory (Colab) environment. With Colab one can execute Jupyter notebooks on Google's cloud servers. Servers can be hardware accelerated with graphics or tensor processing units. This project's classifier was trained with TF GPU and Keras library. Most of the training script can be read from Appendix A.2.

Image data was read from directories using Keras's ImageDataGenerator. RGB intensity coefficients were rescaled from 0–250 to 0–1 to make them easier to process for the model. MobileNetV2 was used as the model, which was described in the Research chapter. The classifier's training process was started with pre-trained ImageNet weights. Starting with weights already trained to some task and using the general knowledge that the CNN has learned to a new task is called transfer learning (Brownlee, 2019a). ImageNet is a huge image dataset, thus models trained with it knows how to recognize many features. The last layer in the model has as many neurons as there are classes (250), and it uses Softmax activation, because the classifier trained for multi-class classification. Softmax, or normalized exponential function, returns the probability distribution of a list of potential outcomes (Uniqtech, 2018).

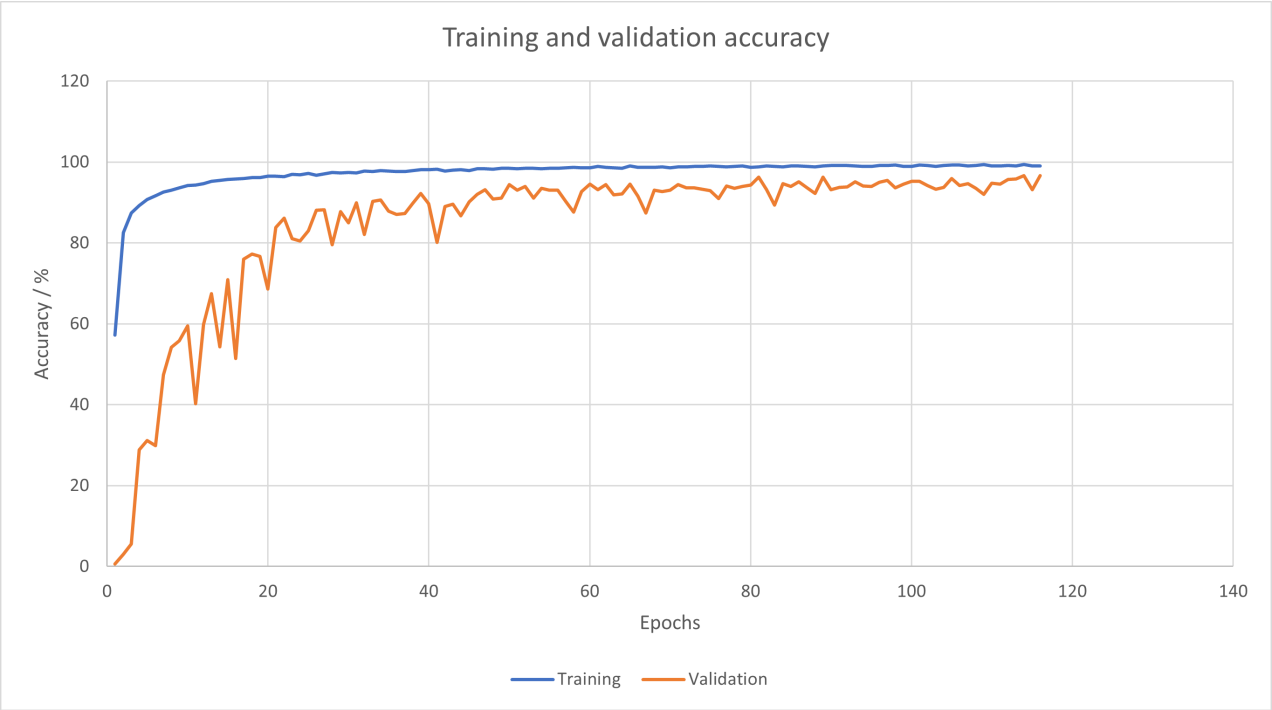


Figure 4.1: Accuracy.

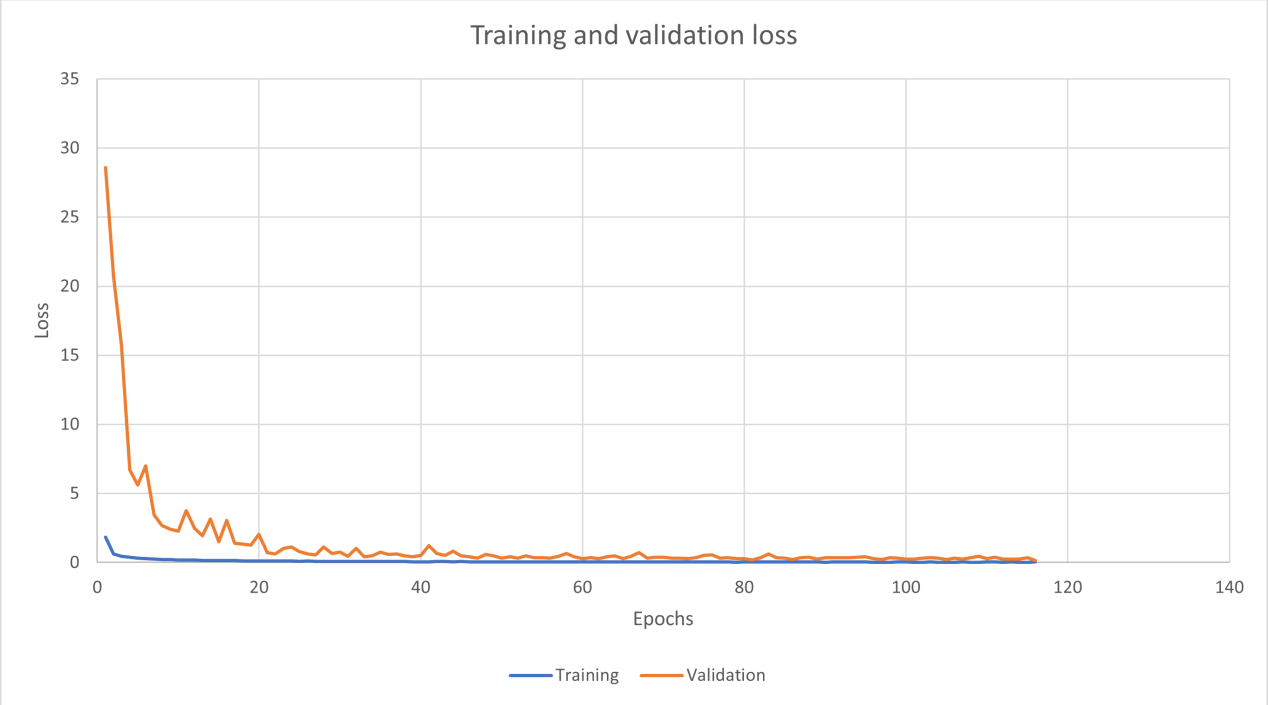


Figure 4.2: Loss.

During training validation loss was monitored with EarlyStopping callback. Validation data is used to fine-tune the model with hyperparameters to data it has not used in training, because any good model will fit to the training data almost perfectly, thus it is overfitted to the data, meaning it will predict poorly on data it has not seen. Hyperparameters are used to control the training process, whereas weights are derived via training. Loss was monitored because it measures how bad the model's predictions are. Model training history was saved into a file with CSVLogger callback and plotted with Excel (Figures 4.1 and 4.2). (Claesen and Moor, 2015).

Patience in the EarlyStopping callback was set to 40 epochs, which is large for a training process like this, but there was no hurry to train the model in the cloud. One epoch means that the entire data set goes through the model once. The training was stopped by the EarlyStopping callback at the 156th epoch and best weights were restored from the 116th epoch. The model can overfit to the validation data also, although it is rare. This is why the final evaluation was done with data the model had never seen in the training nor validation process with `model.evaluate()`. This method evaluated the accuracy to 97.5 % and the loss to 0.1386. For comparison, the validation accuracy was 96.6 % and the loss 0.153 at the 116th epoch.

5. System Validation

5.1. Manual Tests

Test Case Name: End-to-End and Offline Availability
Purpose: Confirm That the System Works for All Use Cases
Procedure Steps: 1. Visit the website and add the app to the home screen. 2. Open the app and go offline. 3. Crop and classify the preloaded bald eagle. 4. Press Select and use camera as input. 5. Crop and classify photo from camera. 6. Press Select and use file browser as input. 7. Crop and classify photo from file browser.
Expected Results: All Steps Worked As Intended

Table 5.1: Test case for the whole system that goes through all the use cases.

The application was developed mobile first, thus use case tests (Table 5.1) were performed on an Android 10 and iPadOS 14 devices using their default browsers. All the procedure steps were successful on both test devices. Windows 10 PC does not suggest the option to use a camera input to `<input type="file">`.

Test Case Name: Random Bird Classifications
Purpose: Confirm That the System Classifies Randomly Selected Images With a High Detection Confidence
Procedure Steps: 1. Select 3 random species from the classes the model was trained on. 2. Select random images for each species (10 % of the training images used for that species ≈ 15 samples). 3. Crop, classify and log results of the selected images. 4. Calculate arithmetic mean and median detection confidences.
Expected Results: Mean Results Have High Detection Confidence

Table 5.2: Test case for classifier accuracy evaluation with random species.

The classifier was evaluated during its *development*. Manual test case was developed for classifying images that were cropped manually (Table 5.2), since the cropping of an image has a significant impact on detection confidence, and no programmatic cropping was implemented.

(a) Gray catbird. (b) Red-faced warbler. (c) White-necked raven.

Figure 5.1: Three randomly selected species (Andrew C, 2014; Demesa, 2015; Krishnappa, 2012).

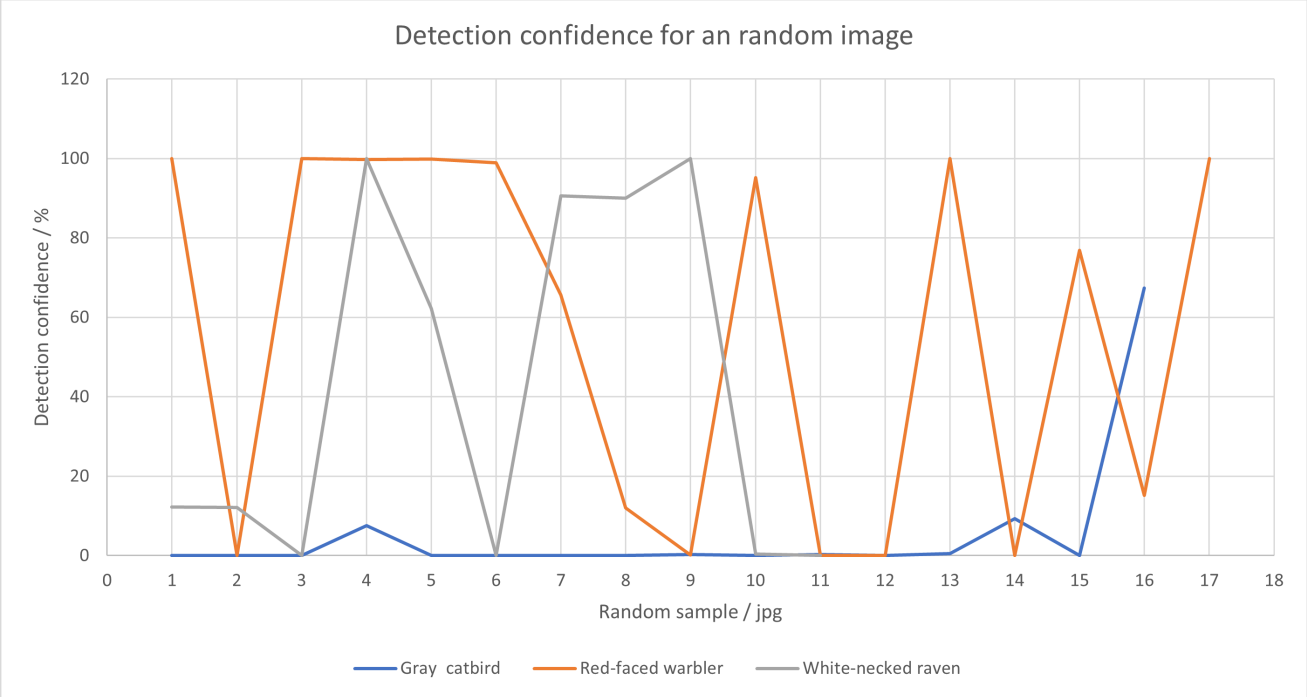


Figure 5.2: Detection confidence per randomly selected image.

	Gray catbird	Red-faced warbler	White-necked raven
Median / %	0.07	76.87	37.24
Arith. mean / %	6.1	56.68	46.75

Table 5.3: Test case for classifier detection confidence evaluation with random species.

The species and images were selected with the help of Google's random number generator. Some of the images were poor quality and the bird could be backwards. The classifier performed poorly with Gray catbird (Figure 5.1a). It classified the species correctly with a decent detection confidence only once out of sixteen samples. The classifier seems to perform better with species that have clear distinctive features, especially color. This is why the test was replicated with handpicked species and test images.

(a) Anna's hummingbird. (b) Fire-tailed myzorni. (c) King vulture.

Figure 5.3: Three handpicked species (Matsubara, 2021; Bhatia, 2015; Neles, 2010).

Figure 5.4: Detection confidence per handpicked image.

	Anna's hummingbird	Fire-tailed myzornis	King vulture
Median / %	86.68	100	99.85
Arith. mean / %	58.98	93.3	65.17

Table 5.4: Test results for classifier detection confidence evaluation with handpicked species.

As expected, the classifier performed significantly better compared to the previous test. The amount of training data per species was not probably sufficient to extract enough features from a feature poor species, such as the Gray catbird.

5.2. Classifier Test with Birdlike Images

(a) Bald eagle mask. (b) Big Bird from Muppets. (c) Yoda from Star Wars.

Figure 5.5: Three birdlike images (Tickle, 2014; The White House, 2013; Wikipedia, 2021b).

This test was done to see how the classifier performs with birdlike images (Figure 5.5).

	Bald eagle mask	Big Bird	Yoda
W/o cropping	1. Golden pheasant 60.1 % 2. Bald eagle 39.69 % 3. Maleo 0.16 %	1. Golden pheasant 46.84 % 2. Eurasian golden oriole 26.7 % 3. King vulture 13.15 %	1. Harpy eagle 54.71 % 2. Long-eared owl 26.73 % 3. Downy woodpecker 17.73 %
With cropping	1. Bald eagle 99.57 % 2. Snowly owl 0.35 % 3. Gila woodpecker 0.02 %	1. Eurasian golden oriole 99.8 % 2. Golden pipit 0.18 % 3. American goldfinch 0.01 %	1. Long-eared owl 96.53 % 2. Harpy eagle 3.18 % 3. Tit mouse 0.15 %

Table 5.5: Top 3 results for classifier detection confidence evaluation with birdlike input.

Everything but the heads were cropped off from images of Yoda and man in a bald eagle mask. The Big Bird was extracted as a whole. One can notice that almost all the predictions for Big Bird have word ”gold” in their names.

(a) Bald eagle. (b) Eurasian golden oriole. (c) Long-eared owl

Figure 5.6: Example images for top predictions with cropped images from Table 5.5 (Krishnappa; 2010; Wikipedia, 2021a; Urbonas, 2007).

5.3. Lighthouse audit

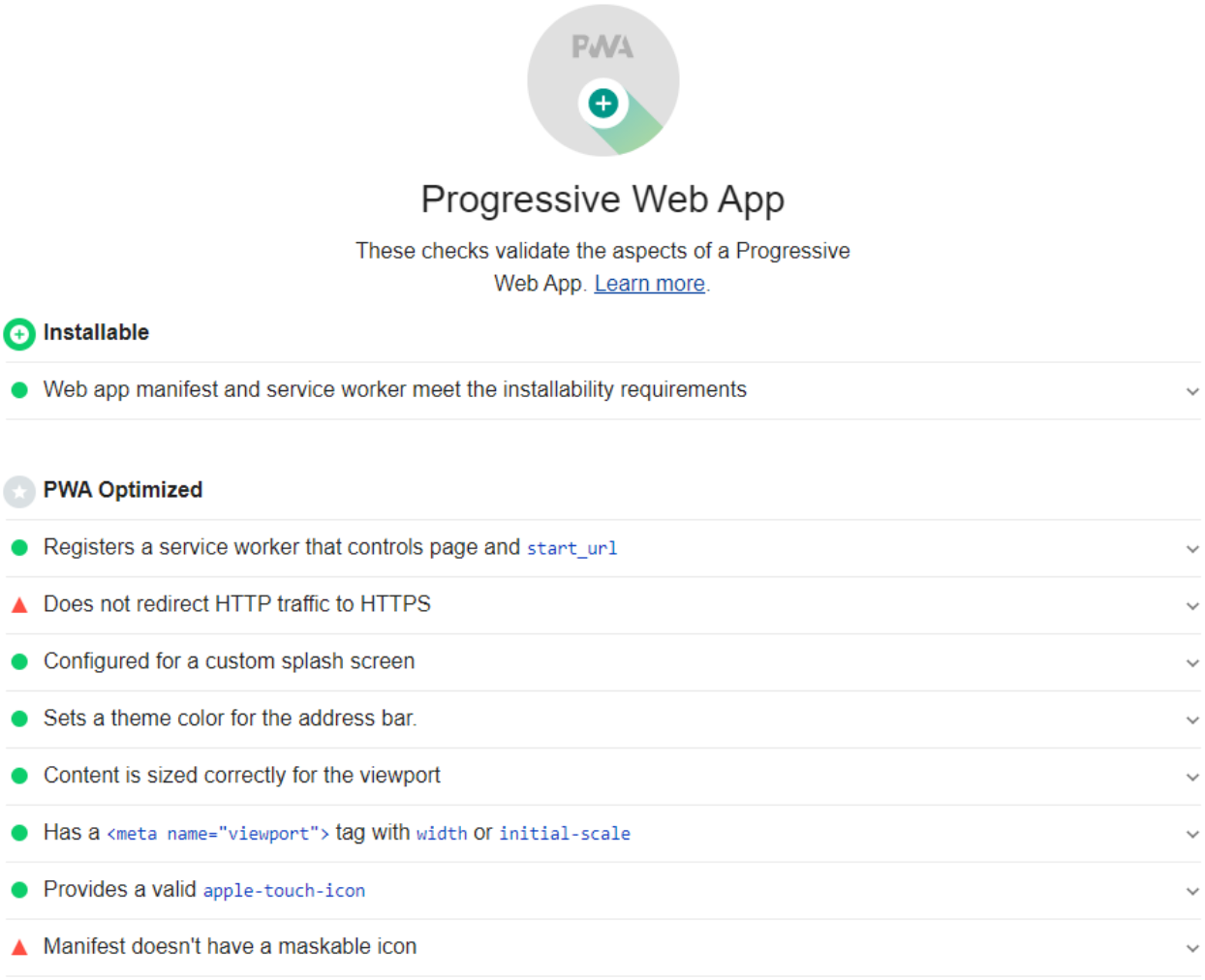


Figure 5.7: Lighthouse PWA checks.

Runtime Settings

URL	https://pwa-birdifier.herokuapp.com/
Fetch Time	Apr 10, 2021, 3:52 PM GMT+3
Device	Emulated Moto G4
Network throttling	150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)
CPU throttling	4x slowdown (Simulated)
Channel	devtools
User agent (host)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.114 Safari/537.36
User agent (network)	Mozilla/5.0 (Linux; Android 7.0; Moto G (4)) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4143.7 Mobile Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	1534

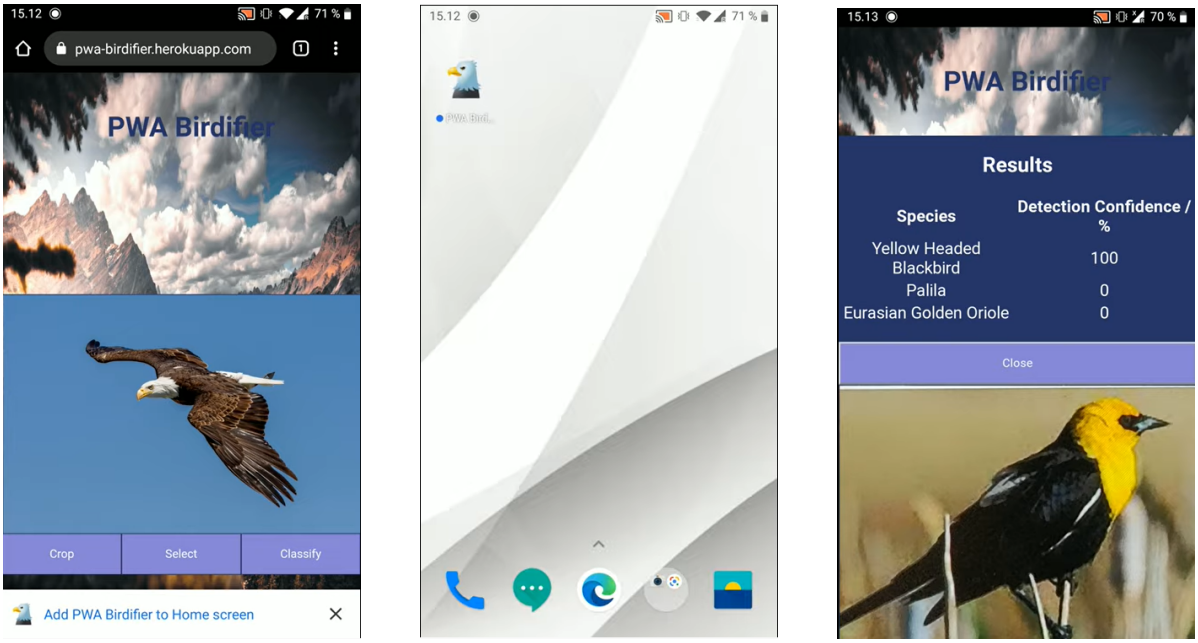
Generated by **Lighthouse** 7.0.0 | File an issue

Figure 5.8: Lighthouse audit runtime settings

The application was audited using Google's Lighthouse tool to confirm it has sufficient PWA features to be classified as one (Figures 5.7 and 5.8).

5.4. Demonstration

Two demonstration videos were made for the project (1. demo, 2. demo). First demo shows most of the steps from the test case shown in Table 5.1. Second demo shows the application in a real-world use case when Mallard ducks were encountered. Rest of this chapter is used for screenshots from both demos.

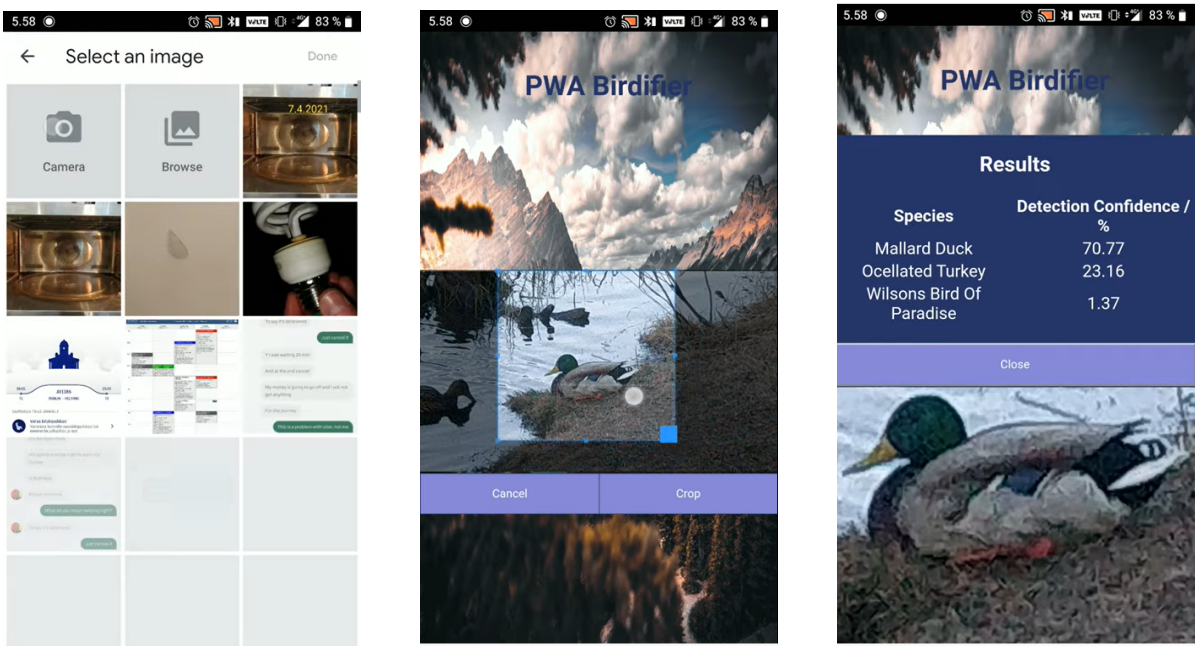


(a) Suggest to add the app to home screen.

(b) The app added to home screen.

(c) A bird classified without internet access.

Figure 5.9: Screenshots from the first demo demonstrating the application's installation and offline usage.



(a) Selecting camera for `<input type="file">`.

(b) Using react-cropper to crop the image.

(c) Mallard duck correctly classified.

Figure 5.10: Screenshots from the second demo demonstrating the application in a real-world situation.

6. Project Development

6.1. Changes From Proposal and During Development

Initially all project tasks were on a single ToDo list in the Kanban board, but were later divided into weekly sprints (Figure 3.1). This helped the author to estimate how much time should be allocated to the project and to show the progression trajectory to the supervisor. Pie chart to show classification results was left out because the author got annoyed with the library used to implement it, and it did not provide enough additional value compared to a HTML table to justify spending more time on it. Unit testing was left out of the web application validation since all the components were tested manually multiple times during the development; the unit tests should have been done before writing the units. Web camera use was not implemented to a PC, because it would have required an additional library, thus the application would have been heavier to cache, and it would not improve real-world use significantly. The classifier was originally planned to be trained on author's PC, but after a quick test and memory allocation error, Google Colab was chosen as the training environment.

7. Conclusion

Developing a PWA is a smooth process with React, no changes were needed to the Service Worker provided with the cra-template-pwa. If this project was continued, one might want to serve the classifier from a database instead of being included in the web application, therefore offering updated classifiers would be smoother. The data set used had 250 classes when the classifier was developed, and at the time of writing the set has 260 classes. It would be easy to use the current model and add more classes to it with transfer learning. Consequently, the user should be notified when a new version of the classifier would be available. Also, one might serve classifiers regionally, because a class for African crowned crane is probably not needed in Ireland. It would also be useful to provide an example image of the top prediction to help users to detect false classifications faster. Users could be allowed to view classification confidences from all the classes somehow, currently only the three highest are presented.

References

Andrew C (May 17, 2014). *File:Gray Catbird (Dumetella carolinensis) (14209547674).jpg*. URL: https://commons.wikimedia.org/wiki/File:Gray_Catbird_(Dumetella_carolinensis)_(14209547674).jpg (visited on 04/12/2021).

Bai, Kunlun (Feb. 11, 2019). *A Comprehensive Introduction to Different Types of Convolutions in Deep Learning*. URL: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215 (visited on 03/06/2021).

Beaufort, Pete LePage; François (Feb. 1, 2021). *Add a web app manifest*. URL: https://web.dev/add-manifest/ (visited on 01/31/2021).

Bhatia, Garima (June 1, 2015). *GarimaBhatia Fire-tailed Myzornis*. URL: https://commons.wikimedia.org/wiki/File:GarimaBhatia_Fire-tailed_Myzornis_IMG_1977.jpg (visited on 04/14/2021).

Bortolossi, Dirce Uesu Pesco; Humberto Jos´e (n.d.). *Matrices and Digital Images*. Tech. rep. Institute of Mathematics and Statistics Fluminense Federal University. URL: http://dmuw.zum.de/images/6/6d/Matrix-en.pdf (visited on 03/06/2021).

Brownlee, Jason (Sept. 16, 2019a). *A Gentle Introduction to Transfer Learning for Deep Learning*. URL: https://machinelearningmastery.com/transfer-learning-for-deep-learning/ (visited on 03/07/2021).

– (Aug. 19, 2019b). *Difference Between Algorithm and Model in Machine Learning*. URL: https://machinelearningmastery.com/difference-between-algorithm-and-model-in-machine-learning/ (visited on 02/19/2021).

Claesen, Marc and Bart De Moor (2015). "Hyperparameter Search in Machine Learning". In: *CoRR* abs/1502.02127. arXiv: 1502.02127. URL: http://arxiv.org/abs/1502.02127.

Demesa, Amado (Apr. 3, 2015). URL: https://www.flickr.com/photos/sabandijah/16854176538 (visited on 04/12/2021).

Gadicherla, Sameer (Jan. 23, 2020). *Tensorflow Vs. Keras: Comparison by Building a Model for Image Classification*. URL: https://hackernoon.com/tensorflow-vs-keras-comparison-by-building-a-model-for-image-classification-f007f336c519 (visited on 01/31/2021).

Guo, Yunhui et al. (2019). "Depthwise Convolution is All You Need for Learning Multiple Visual Domains". In: *CoRR* abs/1902.00927. arXiv: 1902.00927. URL: http://arxiv.org/abs/1902.00927.

Howard, Andrew et al. (2019). "Searching for MobileNetV3". In: *CoRR* abs/1905.02244. arXiv: 1905.02244. URL: http://arxiv.org/abs/1905.02244.

Howard, Andrew G. et al. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv: 1704.04861 [cs.CV].

Howard, Mark Sandler; Andrew (Apr. 3, 2018). *MobileNetV2: The Next Generation of On-Device Computer Vision Networks*. URL: https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html (visited on 03/01/2021).

Idakiev, Ilia (Oct. 24, 2016). *Angular Offline Progressive Web Apps With NodeJS*. URL: https://www.slideshare.net/IliaIdakiev/angular-offline-progressive-web-apps-with-nodejs (visited on 01/31/2021).

Jeffries, Daniel (Sept. 27, 2019). *Learning AI if You Suck at Math — P4 — Tensors Illustrated (with Cats!)* URL: https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32 (visited on 03/06/2021).

Krishnappa, Yathin S (Dec. 18, 2012). *File:2009-white-necked-raven.jpg*. URL: https://commons.wikimedia.org/wiki/File:2009-white-necked-raven.jpg (visited on 04/14/2021).

Krishnappa; Yathin S (July 21, 2010). *Bald-eagle-kodiak*. URL: https://commons.wikimedia.org/wiki/File:2010-bald-eagle-kodiak.jpg (visited on 04/19/2021).

Kuanghuei, Fong (Mar. 4, 2021). *react-cropper*. URL: https://github.com/react-cropper/react-cropper (visited on 03/07/2021).

Lane, Thom (Oct. 18, 2018). *Multi-Channel Convolutions explained with... MS Excel!* URL: https://medium.com/apache-mxnet/multi-channel-convolutions-explained-with-ms-excel-9bbf8eb77108 (visited on 03/06/2021).

Liu, Danqing (Nov. 30, 2017). *A Practical Guide to ReLU*. URL: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7 (visited on 03/06/2021).

Matsubara, Becky (Jan. 19, 2021). *Anna's Hummingbird*. URL: https://commons.wikimedia.org/wiki/File:Anna's_Hummingbird_(39089535924).jpg (visited on 04/12/2021).

Neles, Erwin (Sept. 21, 2010). *BIRD KING VULTURE SURINAM AMAZONE SOUTH-AMERICA*. URL: https://commons.wikimedia.org/wiki/File:BIRD_KING_VULTURE_SURINAM_AMAZONE_SOUTH-AMERICA_(32202834653).jpg (visited on 04/14/2021).

Piosenka, Gerald (2021). *https://www.kaggle.com/gpiosenka/100-bird-species*. (Visited on 01/31/2021).

Pröve, Paul-Louis (Apr. 11, 2018). *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. URL: https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5 (visited on 03/06/2021).

RADIGAN, DAN (n.d.). *How the kanban methodology applies to software development*. URL: https://www.atlassian.com/agile/kanban (visited on 01/31/2021).

Sandler, Mark et al. (2018). "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation". In: *CoRR* abs/1801.04381. arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

Scott (2021). *The 13 Best Bird Watching Apps (2021)*. URL: https://birdwatchinghq.com/birdingapps/#recognition (visited on 02/27/2021).

Slater, Neil (Dec. 31, 2017). *What is the concept of Tensorflow Bottlenecks?* URL: https://ai.stackexchange.com/a/4887 (visited on 03/06/2021).

Smilkov, Daniel et al. (2019). "TensorFlow.js: Machine Learning for the Web and Beyond". In: Palo Alto, CA, USA. URL: https://arxiv.org/abs/1901.05350.

Tamire, Workneh (Dec. 16, 2019). "Evaluation of Progressive Web Application to develop an Offline-First Task Management App". en. G2 Pro gradu, diplomityö, p. 74. URL: http://urn.fi/URN:NBN:fi:aalto-201912226568.

TensorFlow (n.d.). *TensorFlow.js layers API for Keras users*. URL: https://www.tensorflow.org/js/guide/layers_for_keras_users (visited on 01/31/2021).

The White House (Feb. 13, 2013). *Big Bird and Michelle Obama*. URL: https://commons.wikimedia.org/wiki/File:Big_Bird_and_Michelle_Obama_(8555066920).jpg (visited on 04/19/2021).

Tickle, Glen (Sept. 30, 2014). *Realistic-Looking Eagle Mask and Talon Gloves by Archie McPhee*. URL: https://laughingsquid.com/realistic-looking-eagle-mask-and-talon-gloves-by-archie-mcphee/.

Timothy (Sept. 9, 2020). *Making a Progressive Web App*. URL: https://create-react-app.dev/docs/making-a-progressive-web-app/ (visited on 03/01/2021).

Tirth1306 (Sept. 3, 2020). *Browser-based Models with TensorFlow.js*. URL: https://medium.com/analytics-vidhya/browser-based-models-with-tensorflow-js-afcd10be3615.

U.S. Census Bureau (2016). *National Survey of Fishing, Hunting, and Wildlife-Associated Recreation*. Research rep. URL: https://www.census.gov/programs-surveys/fhwar.html (visited on 02/27/2021).

Uniqtech (Jan. 30, 2018). *Understand the Softmax Function in Minutes*. URL: https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d (visited on 03/07/2021).

Urbonas, Mindaugas (Sept. 6, 2007). *Long-eared Owl*. URL: https://commons.wikimedia.org/wiki/File:Long-eared_Owl-Mindaugas_Urbonas-1.jpg (visited on 04/19/2021).

Visualizing matrix convolution, (Dec. 28, 2019). URL: https://tex.stackexchange.com/a/522122 (visited on 03/06/2021).

Wang, Chi-Feng (Aug. 14, 2018). *A Basic Introduction to Separable Convolutions*. URL: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728 (visited on 03/06/2021).

White, John (Aug. 15, 2019). *How popular is birdwatching?* URL: https://chirpbirding.com/blog/81/how-popular-is-birdwatching/ (visited on 02/27/2021).

Wikipedia (Mar. 14, 2021a). *Eurasian golden oriole*. URL: https://en.wikipedia.org/wiki/Eurasian_golden_oriole (visited on 04/19/2021).

– (Apr. 14, 2021b). *Yoda*. URL: https://fi.wikipedia.org/wiki/Yoda (visited on 04/19/2021).

A. Appendix

A.1. Content.js

```
import React, { useState, useEffect, useRef  } from "react";
import * as tf from "@tensorflow/tfjs";
import { ContentWrap } from "./ContentStyles";
import preloadedImgSrc from "../resources/bald-eagle-preloaded-v3.jpg";
import Results from "./Results";
import CropImage from "./CropImage";
import { classes } from "../resources/classes";
import Orbitals from "@bit/joshk.react-spinners-css.orbitals";

function Content() {
  const fileInput = useRef(null);
  const [classifications, setClassifications] = useState([
    { name:  "Bald Eagle", value: 90 },
    { name:  "Golden Eagle", value: 7 },
    { name:  "Turkey Vulture", value: 3 },
  ]);
  const [renderResults, setRenderResults] = useState(false);
  const [renderSpinner, setRenderSpinner] = useState(false);
  const [cropImage, setCropImage] = useState(false);
  const [imgSrc, setImgSrc] = useState(preloadedImgSrc);
  const [model, setModel] = useState(null);
  const modelDir = "/tfjs_files/model.json";
  const pixelHeight = 224, pixelWidth = 224;
  const classesSpecies = Object.keys(classes);
  const noModel = "No model available. Check connection or wait it to load";

  async function saveModel(model) {
    if (model) {
      await model.save("indexeddb://my-model");
    }
  }

  async function loadModel() {
    try {
      setModel(await tf.loadLayersModel("indexeddb://my-model"));
```

```
  } catch (error) {
  //
  } finally {
    setModel(await tf.loadLayersModel(modelDir));
    if (window.indexedDB ) {
      saveModel(model);
    }
  }
}

useEffect(() => {
  loadModel();
}, []);

const OrbitalsStyle = {
  margin: "auto",
  width: "50%",
};

function rescaleImage(image) {
  return image.expandDims(0).toFloat().div(127).sub(1);
}

function setThreeBestPredictions(probabilities) {
  const mapProbabilitiesToSpecies = [];
  for (let i = 0; i < probabilities.length; i++) {
    mapProbabilitiesToSpecies.push({
      species: classesSpecies[i],
      prediction: probabilities[i]
    });
  }

  const highest = probabilities.indexOf(Math.max(...probabilities));
  probabilities[highest] = 0;
  const secondHighest = probabilities.indexOf(Math.max(...probabilities));
  probabilities[secondHighest] = 0;
  const thirdHighest = probabilities.indexOf(Math.max(...probabilities));

  const newClassifications = [
    {
```

```
    name: mapProbabilitiesToSpecies[highest].species,
    value: mapProbabilitiesToSpecies[highest].prediction * 100
    },
    {
    name: mapProbabilitiesToSpecies[secondHighest].species,
    value: mapProbabilitiesToSpecies[secondHighest].prediction * 100
    },
    {
    name: mapProbabilitiesToSpecies[thirdHighest].species,
    value: mapProbabilitiesToSpecies[thirdHighest].prediction * 100
    },
  ];
  setClassifications(newClassifications);
};

const classify = async() => {
  if (imgSrc === preloadedImgSrc) {
    alert("Crop preloaded image before classifying");
    return;
  }
  if (model) {
    setRenderSpinner(true);
    const image = new Image(pixelHeight, pixelWidth);
    image.src = imgSrc;
    try {
      const img = tf.browser.fromPixels(image);
      const batchedImage = rescaleImage(img);
      const predict = model.predict(batchedImage);
      const probabilities = await predict.data();
      setThreeBestPredictions(probabilities);
      setRenderSpinner(false);
      setRenderResults(true);
    } catch (e) {
      setRenderSpinner(false);
      console.log(e);
      alert("Something went wrong with classification. Try again!");
    }
  } else {
    alert(noModel);
  }
```

```
};

const cropOnClick = () => {
  if (renderResults) {
    setRenderResults(false);
  }
  return setCropImage(true);
};

const fileInputOnChange = (e) => {
  e.preventDefault();
  let files;

  if (e.dataTransfer) {
    files = e.dataTransfer.files;
  } else if (e.target) {
    files = e.target.files;
  }

  const reader = new FileReader();
  reader.onload = () => {
    setImgSrc(reader.result);
  };

  try {
    reader.readAsDataURL(files[0]);
  } catch (err) {
    console.log(err);
    alert("Something went wrong with image selection. Try again!");
  }
};

return (
  <ContentWrap>

    {renderResults === true ?
      <Results
        classifications={classifications}
        setRenderResults={setRenderResults}
      />
```

```
  : null
}

{renderSpinner === true ?
  <Orbitals color="#8884d8" style={OrbitalsStyle}/>
  : null
}

{cropImage === true ?
  <CropImage
    setCropImage={setCropImage}
    imgSrc={imgSrc}
    setImgSrc={setImgSrc}
  />
  : <div className="currentImage">
      <img src={imgSrc}></img>
    </div>
}

{cropImage === false ?
  <div>
    <div className="btn-group">
      <button onClick={() => cropOnClick()}>Crop</button>
      <button onClick={() => {
        if (renderResults)
          setRenderResults(false);
        fileInput.current.click();
      } }>Select</button>

      {model === null ?
        <button onClick={() => alert(noModel)}>No Model</button>
        : <button onClick={() => classify()}>Classify</button>
      }
    </div>
    <input
      style={{ display: "none" }}
      ref={fileInput}
      type="file"
      onChange={fileInputOnChange}
      accept="image/*"
```

```
          />
        </div>
        : null
      }

    </ContentWrap>
  );
}

export default Content;
```

A.2. Meat of the Notebook Used to Train the Model

[5]:
```
rescaled_ImageDataGenerator = ImageDataGenerator(rescale=1.0/255) #rescale=1./
 ↪255 scales RGB coefficients from 0-255 to 0-1
```

[6]:
```
pixel_height = pixel_width = 224
batch_size = 64
classes_amount = 250
color_channels = 3
training_samples = 35215
validation_samples = 1250
```

Data downloaded with Kaggle API from Gerald Piosenka

[7]:
```
train_generator = rescaled_ImageDataGenerator.flow_from_directory(
    base_dir + train,
    target_size=(pixel_height,pixel_width),
    batch_size=batch_size,
    class_mode='categorical'
)
validation_generator = rescaled_ImageDataGenerator.flow_from_directory(
    base_dir + valid,
    target_size=(pixel_height,pixel_width),
    batch_size=batch_size,
    class_mode='categorical'
)
test_generator = rescaled_ImageDataGenerator.flow_from_directory(
    base_dir + test,
    target_size=(pixel_height,pixel_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
```

```
Found 35215 images belonging to 250 classes.
Found 1250 images belonging to 250 classes.
Found 1250 images belonging to 250 classes.
```

[8]:
```
#Creates MobileNetV2 model with preloaded weights. Src: https://medium.com/
 ↪hackernoon/tf-serving-keras-mobilenetv2-632b8d92983c
def create_model():
  input_tensor = Input(shape=(pixel_height, pixel_width, 3))

  base_model = tf.keras.applications.MobileNetV2(
  include_top=False,
  weights='imagenet',
  input_tensor=input_tensor,
  input_shape=(pixel_height, pixel_width, 3),
  pooling='avg'
)

  for layer in base_model.layers:
```

```
    layer.trainable = True  # trainable has to be false in order to freeze␣
 ↪the layers

  op = Dense(256, activation='relu')(base_model.output)
  op = Dropout(.25)(op)

  output_tensor = Dense(classes_amount, activation='softmax')(op)

  model = Model(inputs=input_tensor, outputs=output_tensor)

  return model
```

[10]:
```
def evaluate_model(model):
 model.evaluate(test_generator, steps=len(test_generator))

  print("Test loss: {0:.4f}".format(test_loss))
  print("Test acc: {0:.4f} % ".format(test_acc * 100.0))
```

[12]:
```
model = create_model()
model.compile(optimizer = 'adam',loss = 'categorical_crossentropy', metrics =␣
 ↪['accuracy'])
print(model.summary())
```

```
=================================================================================
(Architechture summary removed)
Total params: 2,650,170
Trainable params: 2,616,058
Non-trainable params: 34,112
=================================================================================
```

[13]:
```
#Save model at some interval during the trainig process
checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1,␣
 ↪save_best_only=True, mode='min')

#Use early stopping callback to avoid over fitting
stop_early = EarlyStopping(
    monitor='val_loss',
    min_delta=1e-3,
    patience=40,
    verbose=1,
    mode='auto',
    restore_best_weights = True
)

#CSVLogger logs epoch, acc, loss, val_acc, val_loss
logs = CSVLogger('/content/drive/MyDrive/Colab Notebooks/Logs/logs-v2.csv',␣
 ↪separator=',', append=False)

callbacks = [checkpoint, stop_early, logs]
```

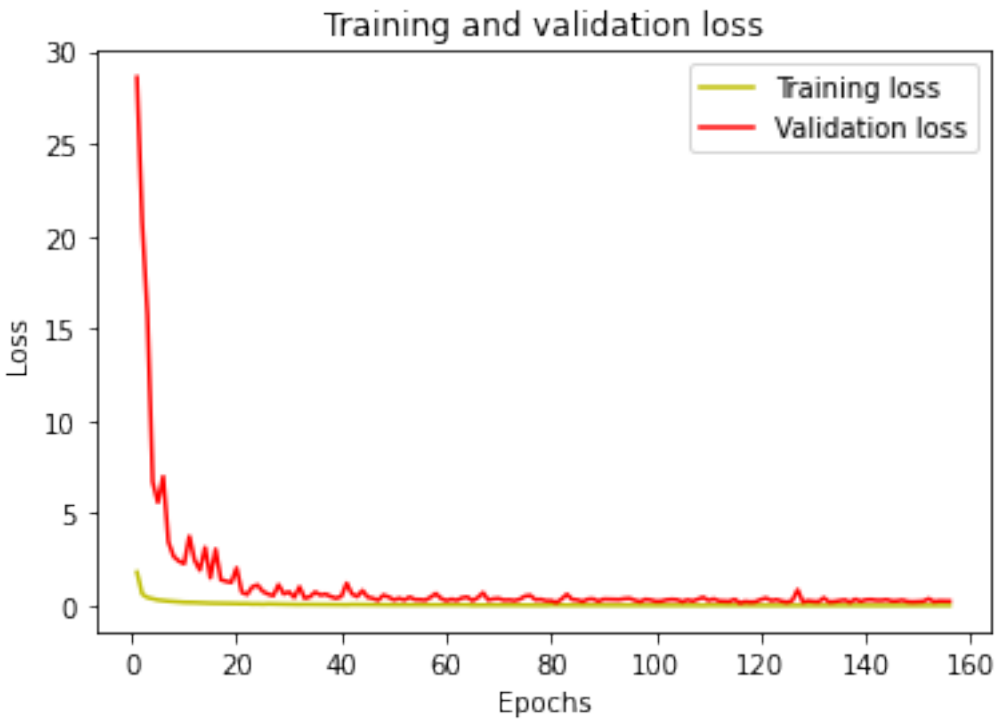
[ ]:
```
history = model.fit(
    train_generator,
```

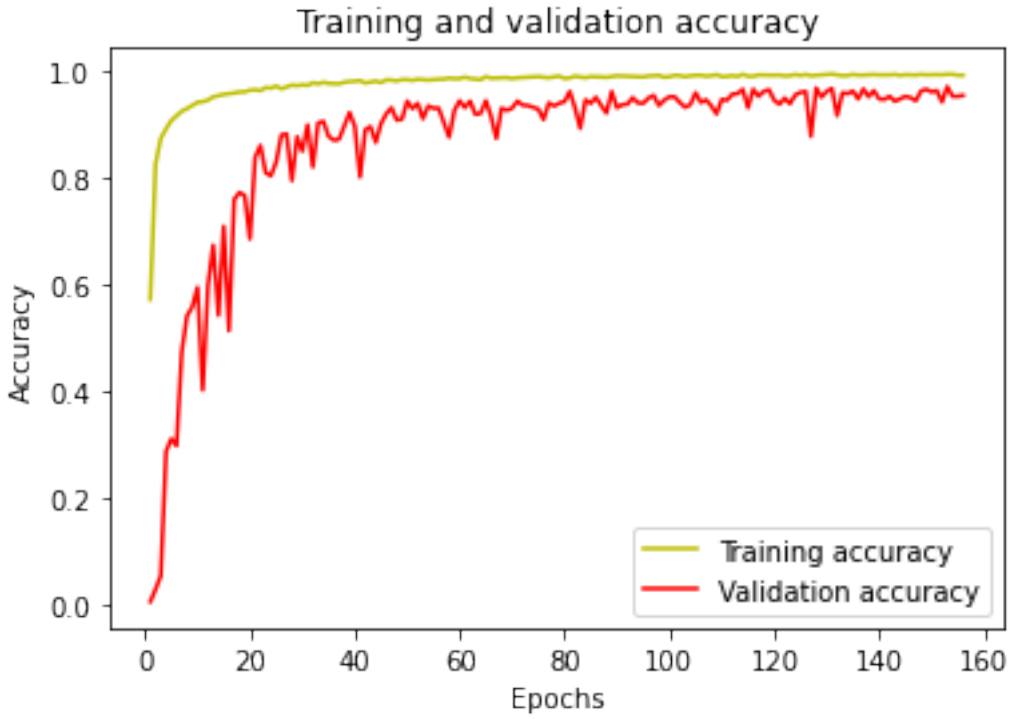
```
    steps_per_epoch = training_samples // batch_size,
    validation_data = validation_generator,
    validation_steps = validation_samples // batch_size,
    epochs = 280,
    verbose=0,
    callbacks=callbacks
)

model.save(model_name)
```

[15]:
```
evaluate_model(model)
```

```
20/20 [==============================] - 4s 182ms/step - loss: 0.1386 -
accuracy: 0.9752
Test loss: 0.1386
Test acc: 97.5200 %
```

[16]:
```
plot_model(history)
```




Training and validation accuracy
Accuracy
Epochs
Training accuracy
Validation accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
120
140
160