Handwriting 2 Text

CA400 Technical Manual

Project Title: Handwriting to Text Converter
Student Name: Conor Hanlon
Student Number: 15445378
Supervisor: Ray Walshe

Abstract:

Handwritten notes and essays are a key part of the academic environment. While the digital world is constantly growing, a lot of students, lecturers and professors still prefer writing notes by hand. My project is centered around converting a user's handwriting to a digital format.

I plan to create and train a neural network model that will be able to identify different characters and words in images uploaded by the user and convert this into a text document. The machine learning process will use convolutional and recurrent neural networks to process the training images. The model will be deployed and utilised by a simple web application. Any person can open up the application and upload images of their handwritten notes. They will be given the option to just download the generated documented, or they can save it to their Google Drive if they are signed into their Google account. It will also be possible to share the document via email. This application will help users keep track of any handwritten notes or essays they may want to keep safe for future use.

Table of Contents

1. Introduction

1.1. Overview

Handwriting to Text Converter is a web application aimed at students who write a lot of notes and essays by hand. It allows them to create a digital copy of their work with ease by creating documents on either their Google Drive or local machine.

This technical manual illustrates all processes involved in the development of this application. It highlights all of the preparation carried out by researching papers and investigations into what technologies would be best suited to ensure a reliable end product. A long design period was necessary to develop a robust application. All diagrams from this phase such as a system architecture diagram, high-level context diagram, data flow diagram and numerous sequence diagrams are shown in this document. Core features such as the implementation of the word prediction model and image segmentation and preprocessing are outlined in detail.

With a project of such size, it is impossible to avoid multiple errors and problems. These were documented along the way, and in this manual I describe the critical thinking applied to find their resolutions. Finally, the project life cycle and results as a whole are analysed, as well as recommendations added for future work.

1.2. Glossary

- *Neural Network:* A system of hardware and/or software patterned after the operation of neurons in the human brain.
- *Image Processing:* The analysis and manipulation of a digitized image.
- *Optical Character Recognition:* The mechanical or electronic conversion of images of typed, handwritten or printed text into machine-encoded text.
- *Convolutional Neural Network:* A class of deep, feed forward artificial neural networks, most commonly applied to analyzing visual imagery.
- *Recurrent Neural Network:* A class of artificial neural network where connections between nodes form a directed graph along a sequence.
- *Connectionist Temporal Classification:* A type of neural network output and associated scoring function, for training recurrent neural networks.
- *Activation Function:* The activation function of a node defines the output of that node, or "neuron," given an input or set of inputs.
- *Loss Function:* Measures the consistency between the predicted value and the actual label.
- *Training Process:* The process of minimising the loss function in order to make accurate predictions.
- *GPU:* Graphics processing unit, used to handle computation in machine learning.
- *Docker container:* A lightweight, standalone, executable package of software that includes everything needed to run an application.
- *API:* Set of functions that allow the creation of applications which access the features or data of another application.
- *Data Augmentation:* Increasing the number of data points in a dataset.

2. Research

At the beginning of my project, my main objective was to gather as much information as possible regarding previous optical character recognition techniques. This was an area that was new to me, meaning that I needed a comprehensive insight to ensure I did not stray away from the objective. We looked at the basics of OCR in our *Search Technologies* module, which gave me a solid basis for which path I should follow.

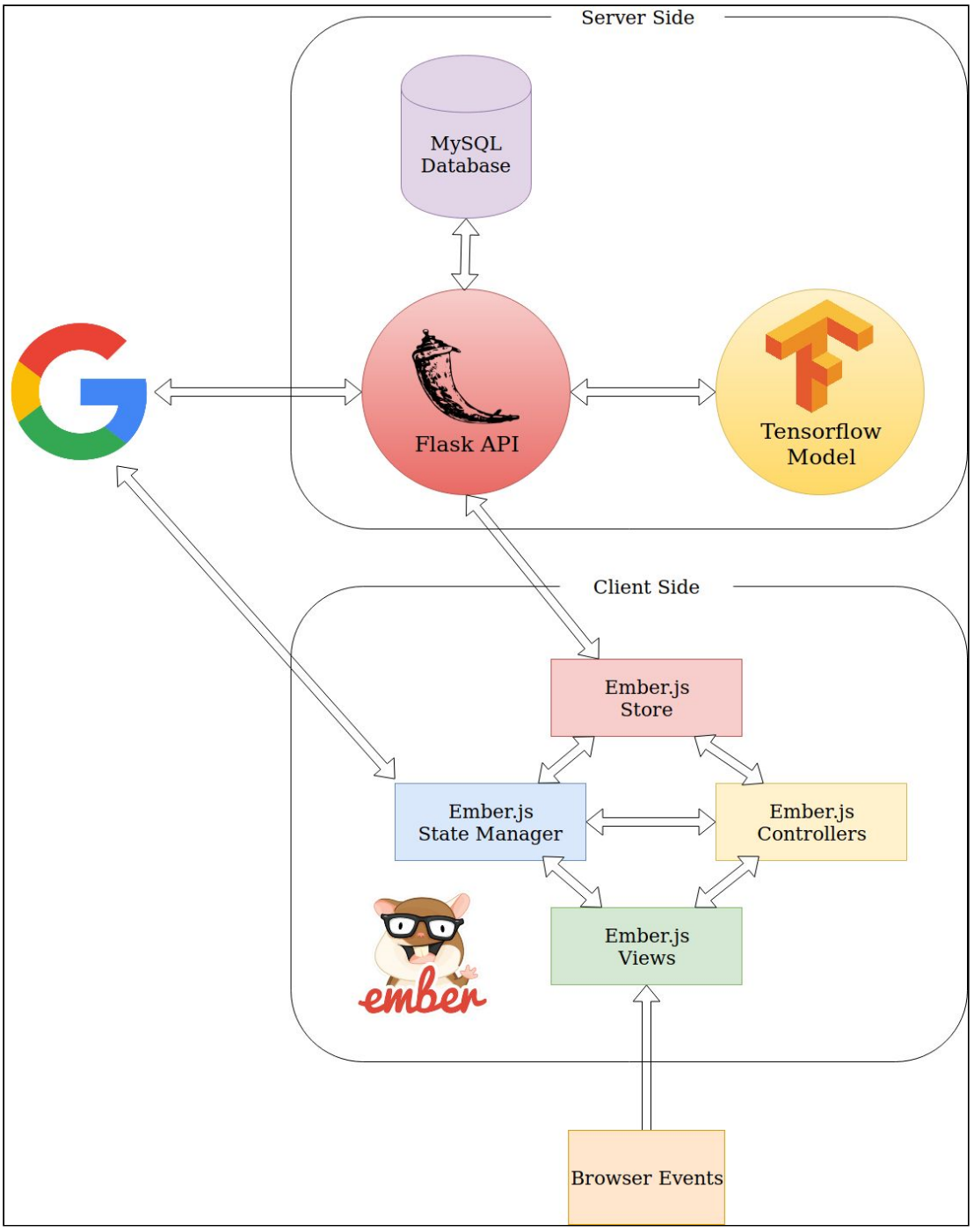
With further research, I found numerous research papers dedicated to optical character recognition using neural networks. I adapted a commonly proposed approach which combines multiple types of network. The first step is to implement convolutional layers which extract features from the images of the words. The output from this is then passed to a deep recurrent neural network. The features are propagated through the LSTM (Long-Short Term Memory) layers, which look at the current and previous contexts of the image sequences to generate a prediction matrix. Finally, a CTC (Connectionist Temporal Classification) decoder is used to make a prediction for the given word. I used the word beam search algorithm for decoding, which is suited for neural networks.

While at this point I had a much clearer vision of my implementation and the structure of my model, I had little experience with coding a neural network. I purchased the book *Hands-On Machine Learning with Scikit-Learn and Tensorflow* by O'Reilly publishers. I learned a vast amount about using Tensorflow to create my model, as well as preprocessing my training and validation datasets. I decided to use the IAM handwriting dataset, which is one of the largest collections of handwritten images available. It has roughly 115,000 images of individual words and is contributed to by 657 different writers.

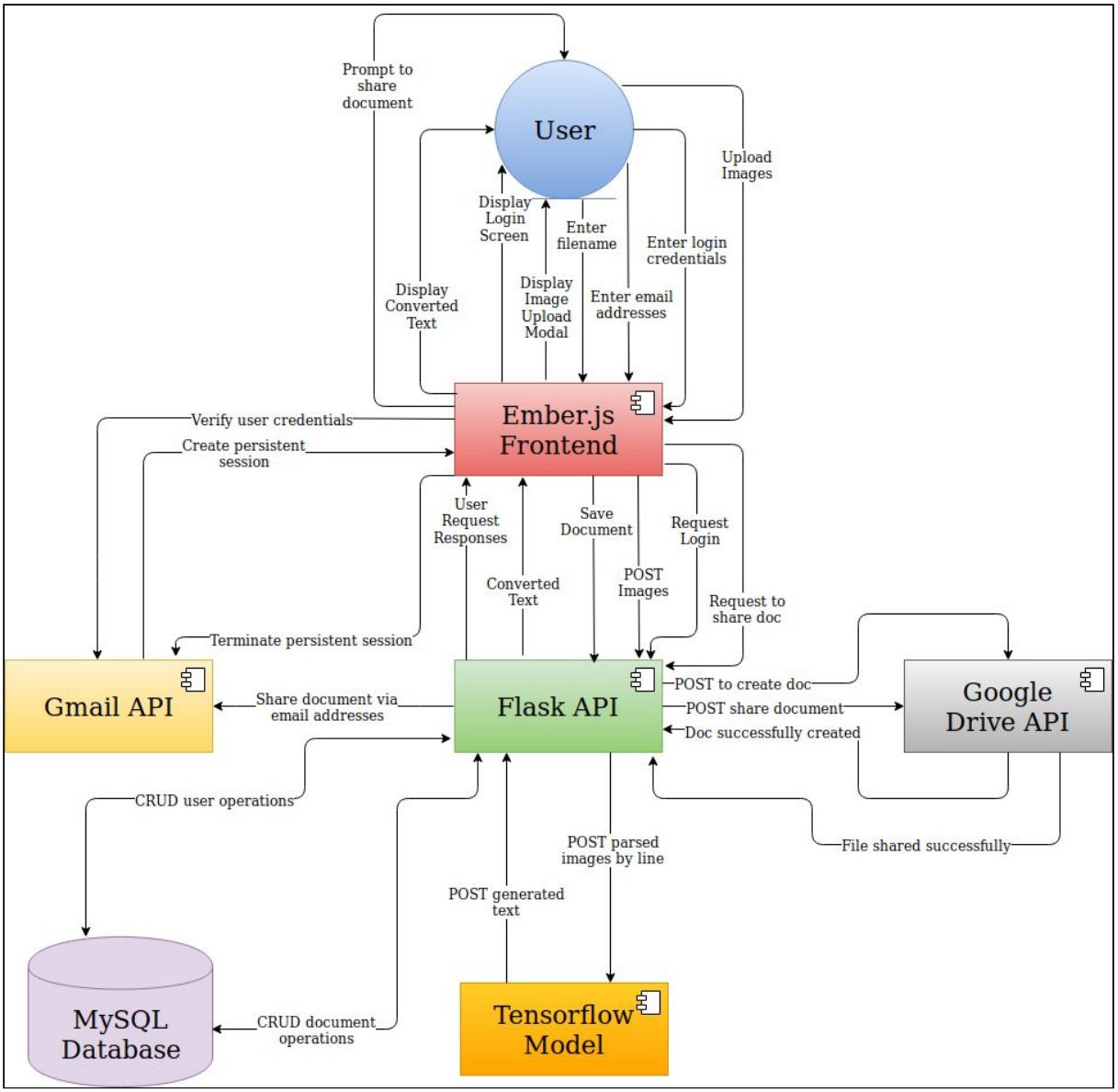
The image processing element was another section that I needed to research for this project. OpenCV is an open-source library that can be used in Python for computer vision. I tried many different techniques during the development process, for functions such as removing any shadows in the image and detecting the text borders around words using their contours. I also used OpenCV for augmentation of my dataset. These implementations will be explained in further detail later in the manual.

3. System Design

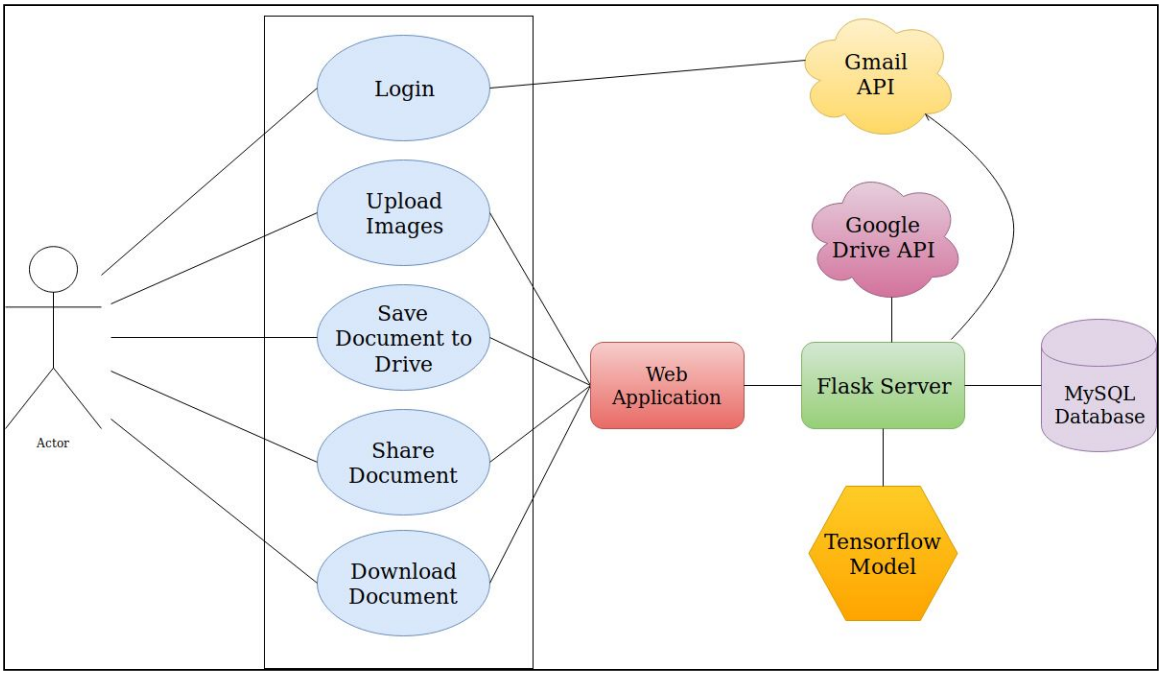
3.1. System Architecture



3.2. Context Diagram

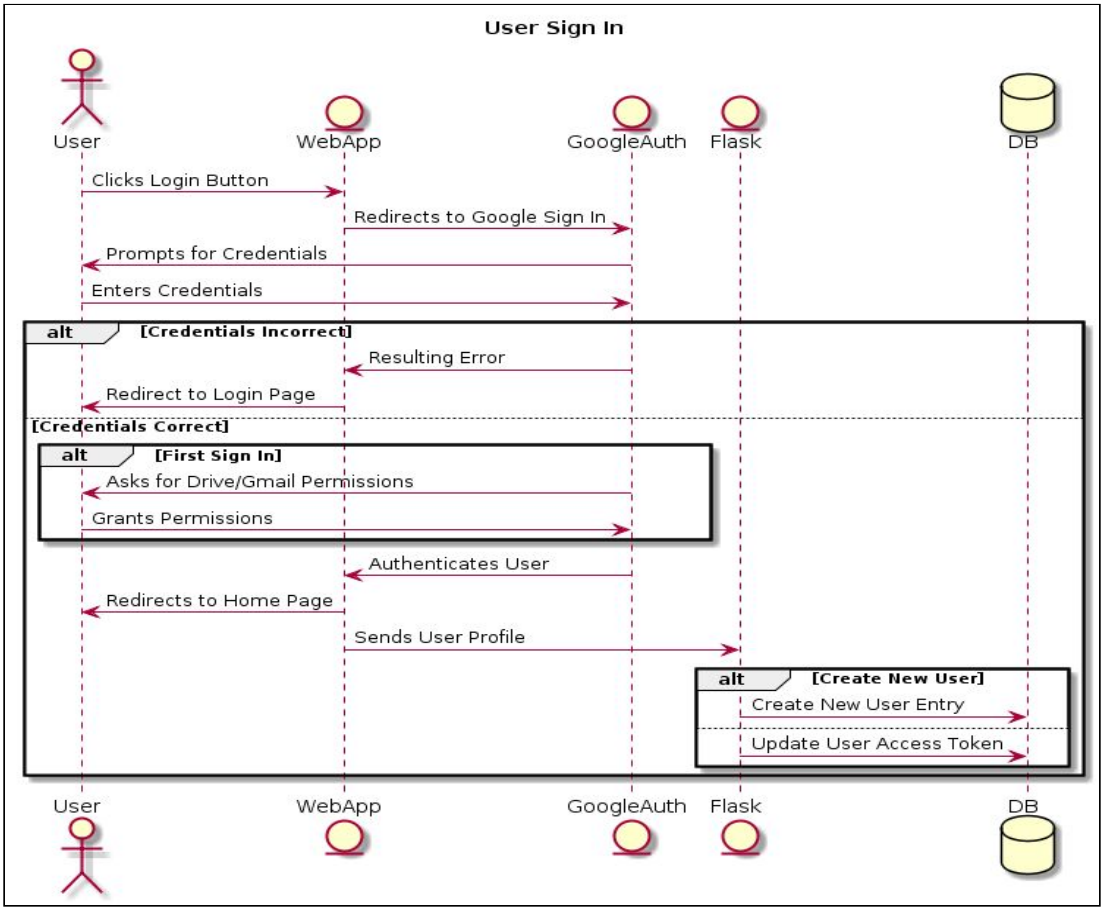


3.3. Use Case Diagram

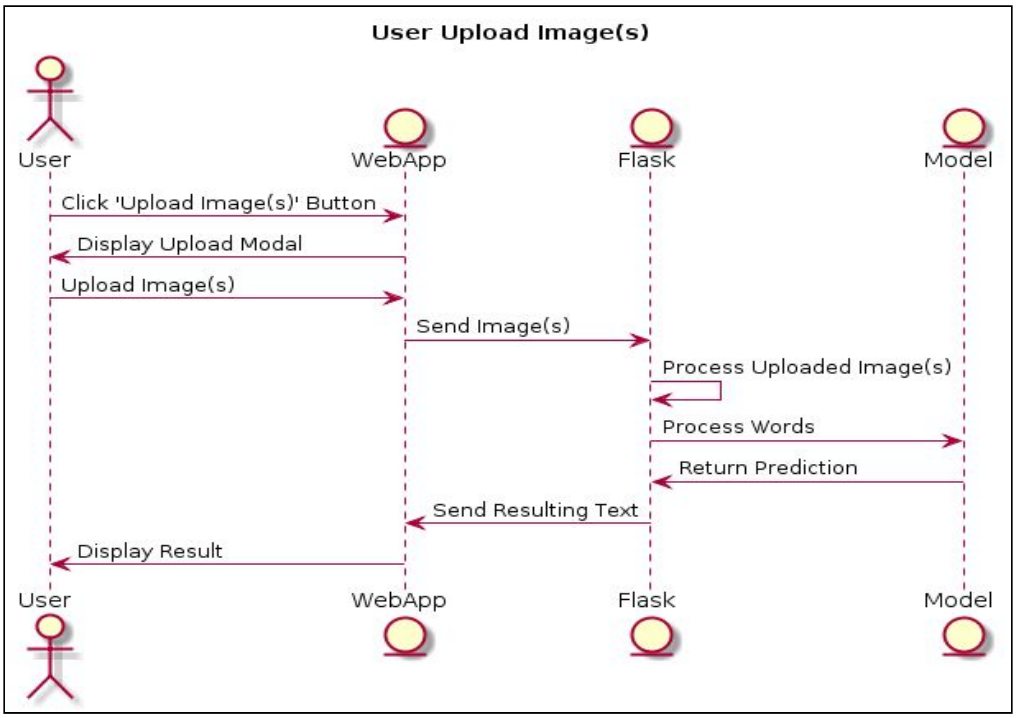


3.4. Sequence Diagrams

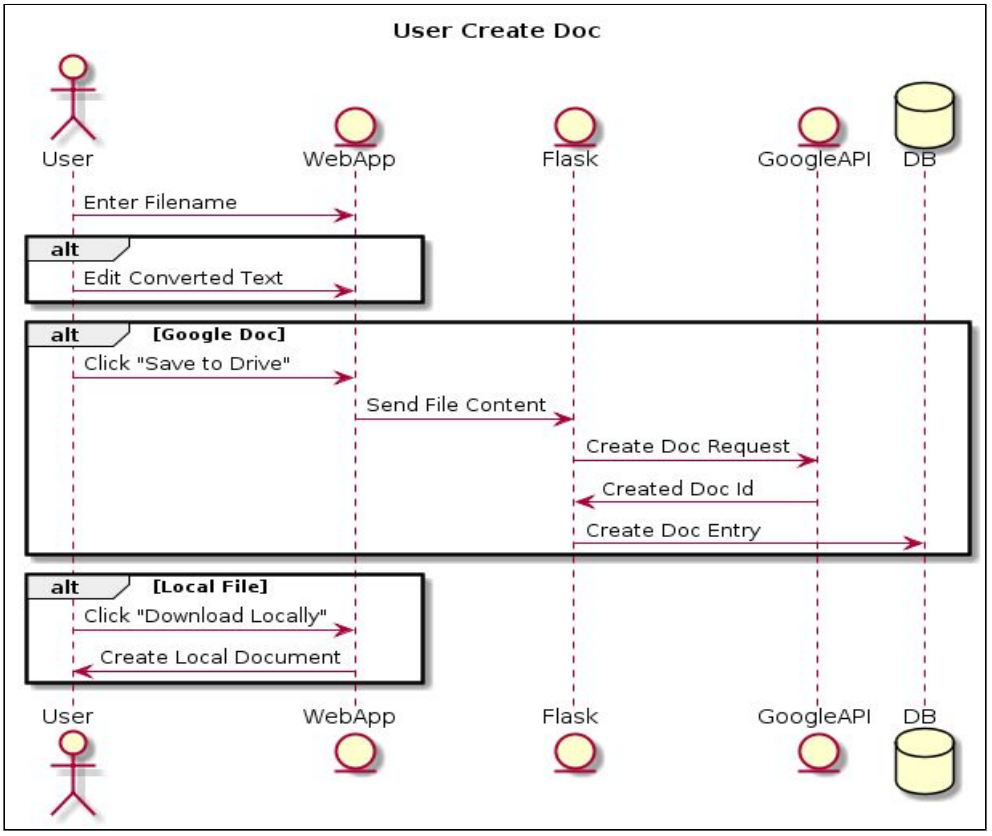
3.4.1. User Login



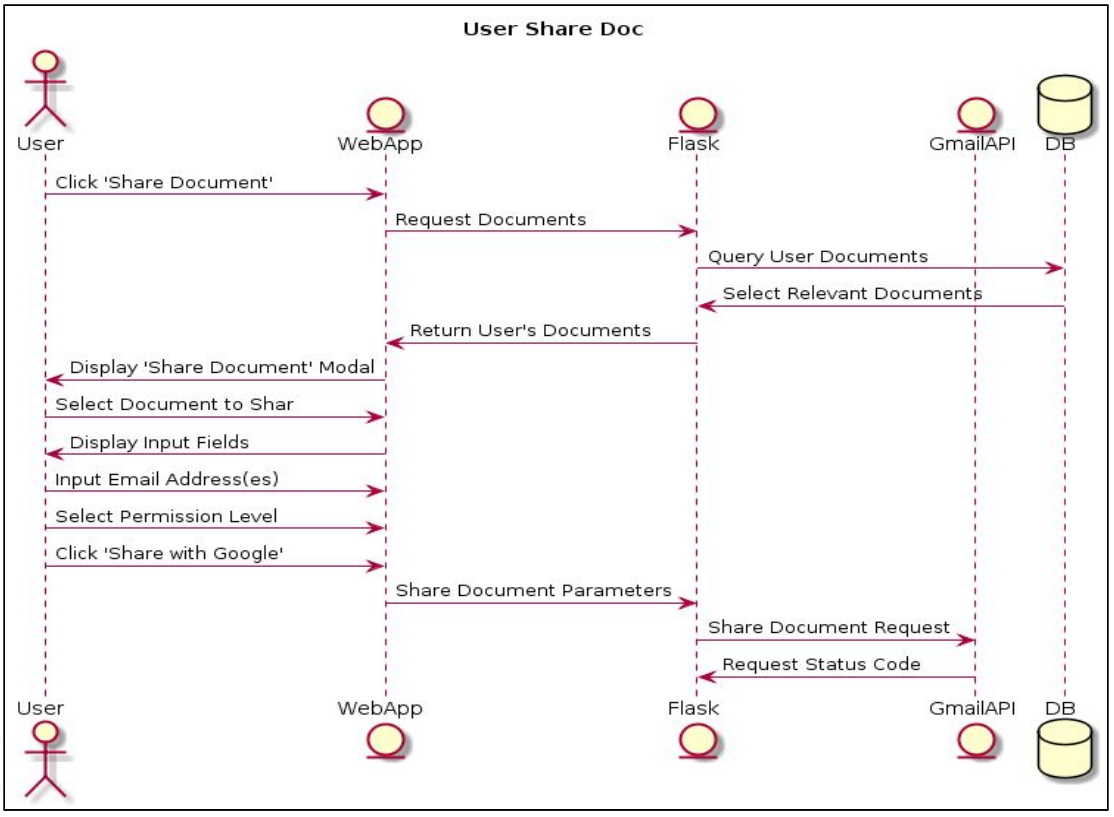
3.4.2. User Uploads Image(s)



3.4.3. User Creates Document

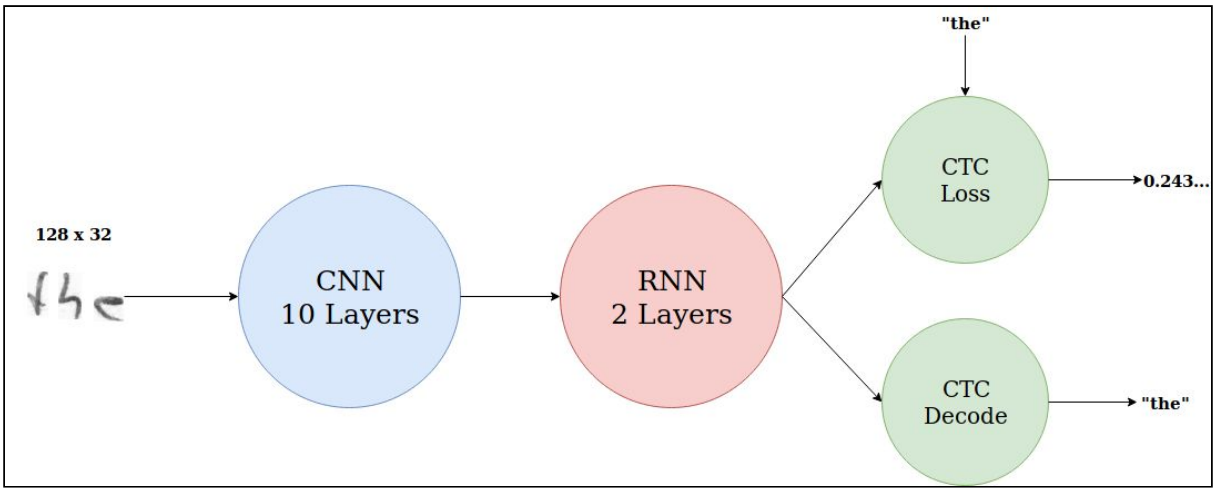


3.4.4. User Shares Document



3.5. Recognition Model Design

Type	Type	Output Size
Input	Gray-Scale Images	128 x 32 x 1
Convolution	kernel: 5x5	128 x 32 x 32
Convolution + Pooling	kernel: 5x5, pool: 2x2	64 x 16 x 32
Convolution	kernel: 5x5	64 x 16 x 64
Convolution + Pooling	kernel: 5x5, pool: 2x2	32 x 8 x 64
Convolution	kernel: 3x3	32 x 8 x 128
Convolution + Pooling	kernel: 3x3, pool: 1x2	32 x 4 x 128
Convolution	kernel: 3x3	32 x 4 x 128
Convolution + Pooling	kernel: 3x3, pool: 1x2	32 x 2 x 128
Convolution	kernel: 3x3	32 x 2 x 256
Convolution + Pooling	kernel: 3x3, pool: 1x2	32 x 1 x 256
LSTM	Bidirectional, 256 Hidden Cells	32 x 256
LSTM	Bidirectional, 256 Hidden Cells	32 x 256
Project	Project onto 80 Characters	32 x 80
CTC	Decode / Loss	<= 32



4. Implementation

4.1. Login (EmberJS, Google Auth API, Google+ API, Flask, MySQL)

The first step in the application is for the user to log in via their Gmail account. For this functionality, I chose to implement a single sign-on approach. This involved redirecting to a Google accounts URL away from the application, which handles the validation of credentials. Once the user has been authenticated, Google redirects to the callback route of the Ember.js application. It passes an access token as a parameter in the URL. This token allows us to make a request to the Google+ API, which returns information about the user for use in the application.

```
let responseUrl = window.location.href;
let access_token = responseUrl.split('&')[0].split('=')[1];
let                                queryUrl                                =
`https://www.googleapis.com/plus/v1/people/me?access_token=${access_token}`;
this.get('ajax').request(queryUrl, {
 method: 'GET',
 success: (userInfo) => {
   this.createCookie(userInfo, access_token);
   console.log(userInfo);
   later(() => {
     this.transitionTo('home');
   }, 1000);
 }
```

As shown in the snippet above, the information is stored in a cookie for use by the application. Later in the process, these values are also sent to the Flask API to store as an entry in the user database table.

```
@app_blueprint.route('/create_user', methods=['POST'])
@app.route('/create_user', methods=['POST'])
@cross_origin()
def create_user():
   if request.form is not None or len(request.form) == 0:
       id = str(request.form['id'])
       user = User.query.filter_by(id=id).first()
       if user is not None:
           access_token = str(request.form['accessToken'])
           if access_token != user.access_token:
               user.access_token = access_token
               db.session.commit()
                   return json.dumps({'status': '201', 'val': 'Access
token updated.'})
           else:
                   return json.dumps({'status': '201', 'val': 'Access
token up to date.'})
       else:
           forename = str(request.form['forename'])
           surname = str(request.form['surname'])
           email = str(request.form['email'])
           image_url = str(request.form['imageUrl'])
           access_token = str(request.form['accessToken'])
                user = User(id, forename, surname, email, image_url,
access_token)
           db.session.add(user)
           db.session.commit()
                return json.dumps({'status': '201', 'val': 'New user
created.'})
   else:
       return json.dumps({'status': '500', 'val': 'Error'})
```

The purpose of this is for authentication when requests are later made to different Google APIs.

4.2. Image Upload (Ember.js)

The uploading of images by users is handled in the Javascript. It stores single or multiple images in a FormData object, which is then sent via POST request to the Flask API. This is the easiest format to send and handle on the server side.

```
addImages: function(e) {
 let files = e.target.files;
 for (let i = 0; i < files.length; i++) {
```

```
    this.fd.append(`image[${this.imageCount}]`, files[i]);
    this.set('imageCount', this.imageCount + 1);
  }
  this.set('hasImages', true);
},
```

4.3. Word Segmentation (Python, OpenCV)

Once the Flask API has received the user's uploaded images, it can then begin the image processing phase. The first step is to remove any shadows from the image that could hinder the system from identifying individual words.

```
def remove_shadow(image):
    rgb_planes = cv2.split(image)
    result_planes = []
    for plane in rgb_planes:
        dilated_img = cv2.dilate(plane, np.ones((5, 5), np.uint8))
        bg_img = cv2.medianBlur(dilated_img, 21)
        diff_img = 255 - cv2.absdiff(plane, bg_img)
        result_planes.append(diff_img)
    return cv2.merge(result_planes)
```

The image is divided into separate colour planes, to remove the darker shadow. Transformations and dilations are then applied to each plane, which removes the darkness from the relevant planes. Finally, they are merged together to reconstruct the original image.

Once the shadow has been removed, the program proceeds to identify the bounding rectangles around each individual word.

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
th, threshed = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY_INV)
kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (6, 3))
dilated = cv2.dilate(threshed, kernel, iterations=2)
components = cv2.findContours(dilated, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[0]
```

The image is converted to grayscale, making the handwriting more identifier. The image is thresholded, which helps classify different pixels, and dilated using a kernel. Once these transformations have been applied, we can use OpenCV to find the letter and word contours on the page. Using the (x, y) coordinates and the returned width and height values, these words are saved as individual images to be passed to the Tensorflow model for prediction.

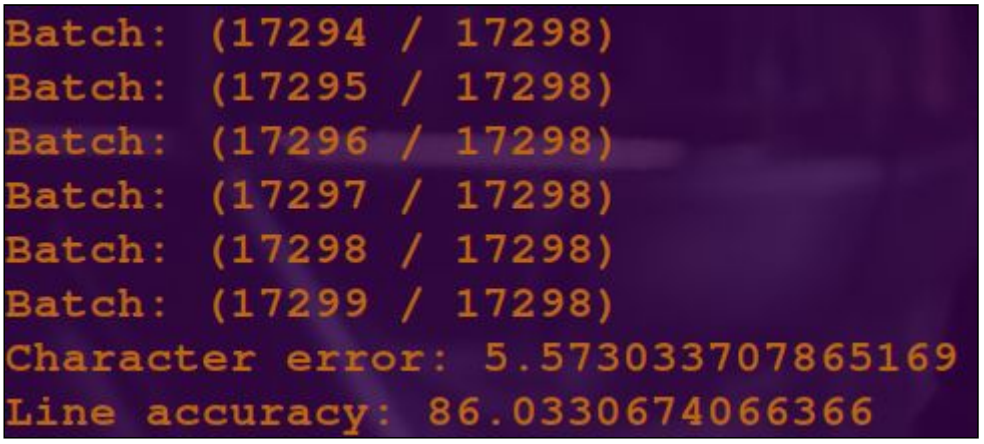
4.4. Neural Network (Python, Tensorflow, OpenCV)

As mentioned in my research section above, my findings led me to take the approach of combining multiple types of neural network to perform the optical character recognition. Before the images could be passed through the model, they had processed to ensure they fit the requirements. The first convolutional layer expects an grayscale image of width 128px by height 32px.

```
kernel = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
convolution = tf.nn.conv2d(self.cnn, kernel, padding='SAME',
strides=(1, 1, 1, 1))
convolution_normalised = tf.layers.batch_normalization(convolution,
training=self.is_train)
relu = tf.nn.relu(convolution_normalised)
```

The images are scaled to a normalized region of that shape, to ensure the letters are not distorted in any way. The images are passed through ten CNN layers to extract all the relevant features. The result of this is a 32x256 matrix with contents of the hidden cells. A stacked recurrent neural network consisting of two LSTM layers then propagates the information back and forth to generate the prediction matrix. Once the matrix is made, the model decodes the output using the CTC decoder. This returns both a prediction for the word contained in the image and the current loss function of the neural network. The loss is used to calculate the accuracy of the model, which is key in optimising its performance.

Using this approach, I reached an accuracy of approximately 86%. Each previously segmented word image is predicted with the model, and the resulting text is sent back to the web application to be displayed.



4.5. Creating Google Document (Ember.js, Flask, Python, MySQL, Google Docs API)

When the user has made any changes they would like to their converted text and entered a filename, they may choose to create a Google Document to be saved to their Drive. By clicking the “Save to Drive” button, a POST request is triggered in the web application that passes the text and filename, as well as the user ID from the cookie. The ID is used to query the MySQL database and find the user’s access token. The token is necessary to make a request to the Google Docs API. I created a function to carry out the request, which sets the necessary headers and constructs the service.

```
def create_doc(access_token, content, filename):
   user_creds = Credentials(token=access_token, token_uri=token_uri,
                                                        client_id=client_id,
client_secret=client_secret)
   service = build('docs', 'v1', credentials=user_creds)
   requests = [
       {
           'insertText': {
               'location': {
                   'index': 1,
               },
               'text': content
           }
       }
   ]
                                                       doc                =
service.documents().create(body={'title':filename}).execute()
   document_id = doc['documentId']
               service.documents().batchUpdate(documentId=document_id,
body={'requests': requests}).execute()
   return document_id
```

The function makes use of the *Credentials* class and *build* function from the Google client Python library. The end results is the ID of the created document, which is then stored as an entry in the documents table in the database. The IDs are necessary for the file sharing functionality of the application.

4.6. Sharing Google Document (Ember.js, Python, Flask, Google Drive API, MySQL)

In the web application, the user has a choice of their created documents available to share. This list is populated from the database when the page is first loaded. When they select a document, they are prompted to input the

email addresses of those they want to share the document with. They must also select the level of permission that will be granted to their chosen friends. The addresses are entered using a power-select tag, which stores the strings in the component.

```
{{#power-select-multiple
 selected=selected
 id="email-input"
 onchange=(action (mut selected))
 onkeydown=(action "createOnEnter")
 as |number|}}
 {{number}}
{{/power-select-multiple}}
```

```
createOnEnter: function(select, e) {
 if (e.keyCode === 13 && select.isOpen &&
   !select.highlighted && !isBlank(select.searchText)) {

   let selected = this.get('selected');
   if (!selected.includes(select.searchText)) {
     select.actions.choose(select.searchText);
   }
 }
}
```

The values are then sent in a POST request to the Flask API. I have another function which handles sharing the Google Document, similar to the function for creating a document.

4.7. Database (Python, Flask, MySQL)

To store user and document information, I set up a MySQL database. I used SQLAlchemy, a commonly used Python library for Flask, to create the tables and insert, read and delete entries. Tables are created using a class inherited from the *SQLAlchemy.Model* template.

```
class CreatedDocument(db.Model):
   id = db.Column(db.String(64), primary_key=True, unique=True)
        uid   =   db.Column(db.String(32),   db.ForeignKey('user.id'),
nullable=False)
   title = db.Column(db.String(64), nullable=False)
   created_on = db.Column(db.DateTime, default=db.func.now())

   def __init__(self, id, uid, title):
```

```
        self.id = id
        self.uid = uid
        self.title = title

    def as_dict(self):
        return {
            'id': self.id,
            'uid': self.uid,
            'title': self.title,
            'created_on': self.created_on.__str__()
        }
```

Using this library allowed for clean, straightforward interactions with the MySQL database.

4.8. Training Process (Google Cloud Platform)

Due to the size of the IAM dataset and the large number of layers in my neural network, it would be infeasible to carry out the training process on my local machine. The School of Computing allocated all final year students €100 in Google Cloud Platform credits, which I made use of. I used a virtual machine with 30 gigabytes of RAM and eight cores. I also attached a Nvidia Tesla GPU for further optimisation.

While the training machine had high specifications, Tensorflow requires you to optimise for operations intended for the GPU.

```
if gpu:
    with tf.device('/gpu:0'):
        self.prepare_cnn()
        self.prepare_rnn()
        self.prepare_ctc()
```

The vast reduction in training time allowed me to experiment with numerous different approaches and model tuning approaches.

5. Problems and Resolutions

5.1. Words vs Lines

When I originally designed my neural network model, I intended to make predictions on images of complete lines of text instead of individual words. The input size would be 512x32, and the design had 5 layers instead of the current 10.

There were a few problems with this approach. Firstly, the IAM dataset only has 13,600 images of complete handwritten lines of text, which would drastically reduce

the amount of training data available. This would make it significantly more difficult to achieve a high accuracy. As well as this, training a neural network to extract the features from a larger image would result in longer training times. While I started my implementation early in the year to stay on track, less time to experiment with ideas would hinder my overall progress.

The image processing functionality proved a challenge also. It is rare that lines of text are completely horizontal across the page. Often letters overlapped between lines, which made it difficult to identify the splitting point.

The simplest resolution to these problems was simply to make predictions on individual words instead of lines. The dataset was much larger, training times were reduced and identify the bounding boxes of a word was much easier than figuring out where the break between two lines is.

5.2. Google Sign In

When I was starting to develop my web application, one of the main challenges was implementing the Google login functionality. I was new to the Ember.js framework and unsure how to set up authenticated user sessions. I researched different sign in approaches, and decided to use single sign-on. This works by redirecting to an external link which handles the authentication. When this process is completed, an access token for the user is passed as a parameter to the we application's callback route.

While this seemed straightforward initially, I had some problems handling the routing. The home route was not blocked, so any user could access it without passing through the Google sign-in. After some research, I found a library specifically designed to handle authenticated and unauthenticated routes. It uses Ember.js mixins to determine whether or not a user session is active. The callback route was of type *OAuth2ImplicitGrantCallbackRouteMixin*, which allowed it to start these user sessions when redirected to via Google. This resolved the problem, allowing me to make use of Google APIs in the web application.

6. Conclusion

Overall, I feel that my experience with this project has taught me a lot about many different technologies. I had little knowledge of computer vision or machine learning at the beginning. The process has taught me that a neural network requires a lot of research and planning. Anyone developing a model should account for a lot of trial and error, as the likelihood of the initial design working perfectly as intended is marginal.

The application itself works to a high level, taking into account the time constraints as well as other assignments and exams. With an accuracy of roughly 86%, the model makes accurate predictions of the user's handwritten words. The user interface is simple and intuitive, while still providing helpful features to the target audience.

7. Future Work

Due to time constraints with the project, there were a few extra features I had considered but did not have time to implement. One of these was to further explore processing the images. The application could try to distinguish headings from blocks of text in the page and format the document accordingly. It would also be a great idea to identify bullet points, and add these to the Google Doc. This would be a more comprehensive approach to converting the documents, but it was just not feasible in the time frame.

A lot of students, particularly in STEM courses, draw diagrams to accompany their notes and visualise how different components work. A practical idea would be to use computer vision to convert these diagrams into a Google Drawing file. This would be a fantastic resource for keeping track of all their diagrams. However, this would almost be an entire project in itself, and I was not able to complete this alongside my handwriting to text converter.

8. Installation

This application can be installed and hosted on a Debian Linux machine.

Prerequisites:

- Python:3.5
- MySQL:5.7

Run the following commands to deploy the application:

```
git clone git@gitlab.computing.dcu.ie:hanlonc5/2019-ca400-hanlonc5.git

cd 2019-ca400-hanlonc5/src

bash deploy.sh
```

9. Appendices

1. *Tensorflow:* https://www.tensorflow.org/
2. *Flask:* http://flask.pocoo.org/
3. *Ember.js:* https://www.emberjs.com/
4. *Docker:* https://www.docker.com/
5. *Google Drive API:* https://developers.google.com/drive/
6. *Gmail API:* https://developers.google.com/gmail/api/
7. *OpenCV:* https://www.opencv.org/
8. *IAM Handwriting Database:* http://www.fki.inf.unibe.ch/databases/iam-handwriting-database
9. *An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition:* https://arxiv.org/pdf/1507.05717.pdf

10. *Handwritten Text Recognition in Historical Documents:* https://repositum.tuwien.ac.at/obvutwhs/download/pdf/2874742