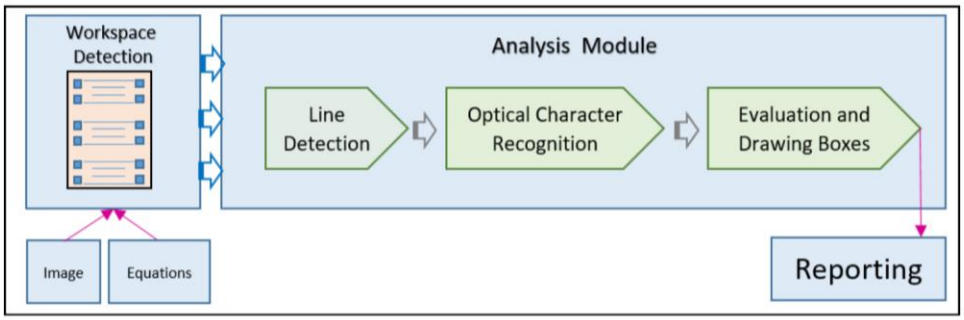
Project name

Evaluating mathematical problems automatically using image processing and deep learning algorithms.

Project Scope

The objective of this project is to develop a computer vision algorithm along with solution package for recognizing, digitizing and validating a mathematical equation written by freehand on a paper. The following are the modules the project has been divided into to achieve the end objective

1. Workspace Detection module - responsible for detecting multiple workspaces in a given sheet of paper using pre-defined markers.
2. Analysis Module - Analysis module is responsible for detecting and localizing characters in any given single workspace, and mathematically analyzing them and drawing red, green lines depending upon their correctness where green represents correct and red represents wrong answers.



Technical Scope

This section gives a detailed description about the data set used for analytics and the various models and techniques used.

Datasets

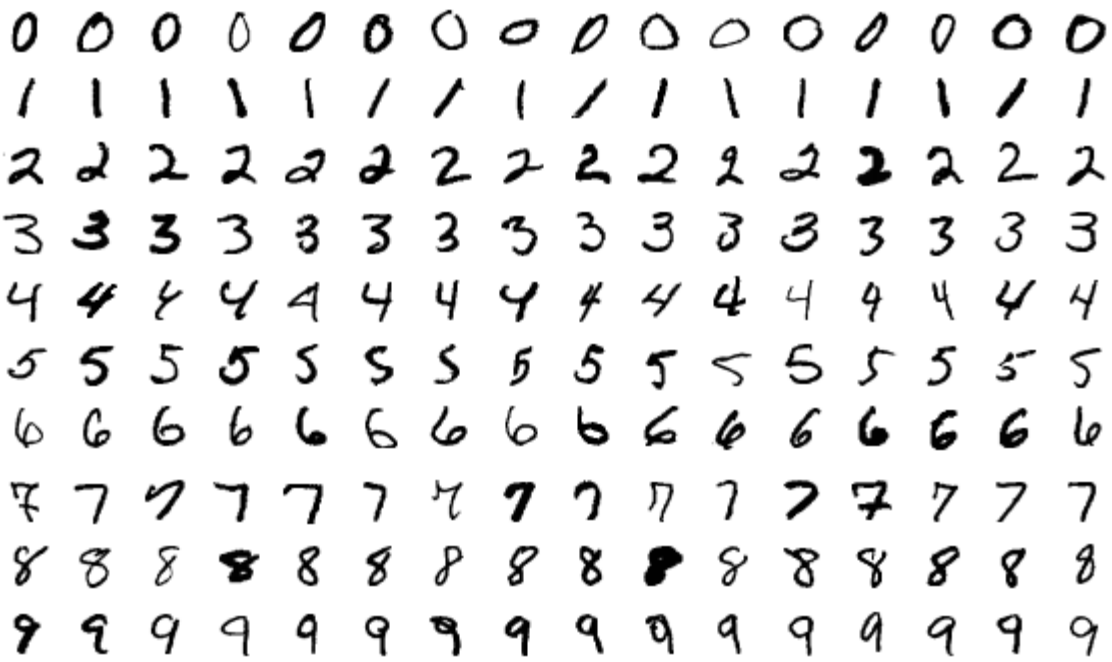
Two open source datasets such as MNIST and Kaggle's mathematical symbols are used for optical character recognition.

• MNIST - Samples provided from MNIST (Modified National Institute of Standards and Technology) dataset from kaggle, includes handwritten digits total of 45,000 images consisting of around 35000 examples in training set and around 10,000 examples in testing set.

• The `MNIST` database (National Institute of Standards and Technology) is a large database of `handwritten digits that is commonly used for training various image processing systems.` • The database is also widely used for training and testing in the field of machine learning. It was created by `re-mixing` the samples from NIST's original datasets.

• Kaggle's Handwritten Mathematical symbols - This datasets include 82 symbols but only a few symbols such as "+", "-", "*", "(", ")" are selected. Each symbol contains at most 4000 examples Images has to be processed in the same way as MNIST before training.

- SAMPLE IMAGE:



Training and Validation Split

The dataset is split into training and validation subsets in the ratio of 85:15 using tensorflow's image data generator.

Workspace Detection Module

Steps involved in workspace detection

• Finding closed rectangular boxes

• Sorting the boxes (Top-to-Bottom) based on the coordinates

• Choosing the desired boxes based on the area.

Analysis Module

Line Detection - The approach for line detection assumes that there is a sufficient gap between lines and there is some intersection between exponential characters and line. The binary images of the detected workspaces are compressed in a single array to take forward derivative thereby detecting the coordinates of each line.

Line Detection using OCRopus - OCRopus is a collection of document analysis programs, it performs the following steps :

• Binarization - Steps involved in binarization are estimating skew angle, estimating thresholds and rescaling

• Page-Layout Analysis - Performs text line finding, this identifies the tops and bottoms of text lines by computing gradients and performs some adaptive thresholding, those components are then used as seeds for the text-line recognition.

• Text-Line Recognition - Text line Recognition uses CLSTM. CLSTM is an implementation of the LSTM recurrent neural network model in C++, using Eigen library for numerical computations. After text line recognition each lines in the image are then extracted and saved as separate images with labels in '.txt' format.

• OCR

Data preparation for DCCNN:

Image pre-processing is performed prior to prediction for the extracted characters from scanned worksheet to match the training dataset.

Training:

• Activation Function Softmax is used in the final layer, all other layers contain ReLu activation function

• Optimizers:

```
o AdaDelta optimizer is an adaptive learning rate method which requires no manual tuning of a learning rate
performed well compared to other optimizers. The parameters used are,
   Learning Rate = 1.0  and Rho = 0.95 (decay factor)
```

• Batch Normalization is used to overcome vanishing gradient problem

• Dropout of 50 % is used before the final dense layer

• Model is trained for only 10 epochs with accuracy up to 97.5%

• Augmentation only slightly improved accuracy in this case (Random rotations, width shift, height shift)

Accuracy

- At the end of 10th epoch the validation accuracy was around `97.59%` , offering a minimum `loss of 0.070`

Tesseract

Tesseract 4 adds a new neural net (LSTM) based OCR engine which is focused on line recognition. It also has a unique configuration option for detecting equation region in the document

Flask

Flask is a lightweight WSGI web application framework, designed to make getting started quick and easy, with the ability to scale up to complex applications.

Handling FILE UPLOADS with flask

- Create a basic form that accepts the images to process and evaluate the sum. Here is a simple HTML page with a form that accepts a file:

```
<!doctype html>
      <html>
      <head>
          <title>File Upload</title>
      </head>
      <body>
          <h1>File Upload</h1>
          <form method="POST" action="" enctype="multipart/form-data">
          <p><input type="file" name="file"></p>
          <p><input type="submit" value="Submit"></p>
          </form>
      </body>
   </html>
```

The method attribute of the

element can be GET or POST. With GET, the data is submitted in the query string of the request URL, while with POST it goes in the request body. When files are being included in the form, you must use POST, as it would be impossible to submit file data in the query string.
The enctype attribute in the element is normally not included with forms that don't have files. This attribute defines how the browser should format the data before it is submitted to the server. The HTML specification defines three possible values for it:

```
*  application/x-www-form-urlencoded: This is the default, and the best format for any *  forms except those that contain file fields.
*  multipart/form-data: This format is required when at least one of the fields in the form is a file field.
*  text/plain: This format has no practical use, so you should ignore it
```

Accepting File Submissions with Flask

For regular forms, Flask provides access to submitted form fields in the request.form dictionary. File fields, however, are included in the request.files dictionary. The request.form and request.files dictionaries are really "multi-dicts", a specialized dictionary implementation that supports duplicate keys. This is necessary because forms can include multiple fields with the same name, as is often the case with groups of check boxes. This also happens with file fields that allow multiple files.

Ignoring important aspects such as validation and security for the moment, the short Flask application shown below accepts a file uploaded with the form shown in the previous section, and writes the submitted file to the current directory:

```
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/', methods=['POST'])
def upload_file():
    uploaded_file = request.files['file']
    if uploaded_file.filename != '':
        uploaded_file.save(uploaded_file.filename)
    return redirect(url_for('index'))
```

The `upload_file()` function is provided with `@app.route` so that it is invoked when the browser sends a POST request. The `uploaded_file` variable holds the submitted file object. This is an instance of `class FileStorage`, which Flask imports from `Werkzeug`.

The filename attribute in the `FileStorage` provides the filename submitted by the the user who wants the image to be evaluated. If the user submits the form without selecting a file in the file field, then the filename is going to be an empty string, so it is important to always check the filename to determine if a file is available or not.
When Flask receives a file submission it does not automatically write it to disk. This is actually a good thing, because it gives the application the opportunity to review and validate the file submission, as you will see later. The actual file data can be accessed from the stream attribute. If the application just wants to save the file to disk, then it can call the save() method, passing the desired path as an argument. If the file's `save()` method is not called, then the file is discarded.

It is important to look into the following aspects while uploading the files:

- Securing File Uploads
- Limiting the size of the files
- Validating the filenames
- Validating file contents

Dealing with uploaded files

We can either make it the uploads as public or for more secure purposes we can make it private:

- Consuming public uploads
- Consuming private uploads

Dropzone JS

Use `dropzone JS` to upload files as the library is flexible and is more customizable. There are other javascript libraries to facilitate uploading of files, as they all follow the HTTP standard, which means that your Flask server is going to work well with all of them. Feel free to look at Dropzone.js documentation
Here is the complete and updated version of main.py designed to work with dropzone.js:

```
import imghdr
import os
from flask import Flask, render_template, request, redirect, url_for, abort, \
    send_from_directory
from werkzeug.utils import secure_filename
from model import execute
from os import listdir
from os.path import isfile, join

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 2 * 1024 * 1024
app.config['UPLOAD_EXTENSIONS'] = ['.jpg', '.png', '.gif', '.JPG']
app.config['UPLOAD_PATH'] = 'uploads'

def validate_image(stream):
    header = stream.read(512)
    stream.seek(0)
    format = imghdr.what(None, header)
    if not format:
        return None
    return '.' + (format if format != 'jpeg' else 'jpg')

@app.errorhandler(413)
def too_large(e):
    return "File is too large", 413

@app.route('/')
def index():
    files = os.listdir(app.config['UPLOAD_PATH'])
```

```
    return render_template('index.html', files=files)

@app.route('/', methods=['POST'])
def upload_files():
    uploaded_file = request.files['file']
    filename = secure_filename(uploaded_file.filename)
    if filename != '':
        file_ext = os.path.splitext(filename)[1]
        if file_ext not in app.config['UPLOAD_EXTENSIONS'] or \
                file_ext != validate_image(uploaded_file.stream):
            abort(400)
        uploaded_file.save(os.path.join(app.config['UPLOAD_PATH'], filename))
    return redirect(url_for('index'))

@app.route('/send', methods=['POST'])
def send_images():
    onlyfiles = [f for f in listdir('./uploads/') if isfile(join('./uploads/', f))]
    return execute(onlyfiles)

@app.route('/uploads/<filename>')
def upload(filename):
    return send_from_directory(app.config['UPLOAD_PATH'], filename)

if __name__ == '__main__':
    app.run(host='127.0.0.1', debug=True)
```