Computer science coursework: Handwritten Alpha-numeric character recognition

Table of Contents

[Analysis 1](#_Toc96879670)

[Problem Identification 1](#_Toc96879671)

[My clients 2](#_Toc96879672)

[Current solutions 2](#_Toc96879673)

[Limitations: 3](#_Toc96879674)

[Diagram of current systems: 3](#_Toc96879675)

[Character detection: 3](#_Toc96879676)

[Design: 4](#_Toc96879677)

[Key points of the solution: 4](#_Toc96879678)

[Neural Network: 4](#_Toc96879679)

[Character extraction: 8](#_Toc96879680)

[Developing the coded solution: 9](#_Toc96879681)

[Neural Network: 9](#_Toc96879682)

[Dataset: 9](#_Toc96879683)

[Model design: 11](#_Toc96879684)

[Training Process: 16](#_Toc96879685)

[Character detection: 17](#_Toc96879686)

[Pre-Processing: 17](#_Toc96879687)

[User Interface: 19](#_Toc96879688)

[Evaluation: 20](#_Toc96879689)

# Analysis

## Problem Identification

Currently there is still a large amount of data stored in hard copies, like forms, charts, etc., this data is not future proof and can easily be destroyed or lost. Hard copy data is also difficult to use as the data cannot be directly used by computers for processing. A software that recognizes handwritten characters from a form or document will reduce the time and effort required to process information gathered in physical formats, the data can directly be stored in a database, or it can be processed to yield desired results.

The user would require an image of the document they wish to convert into a digital format.

The solution will recognize and extract handwritten alpha-numeric characters from an Image and parse then into a neural network that will recognize the character. The prediction from the neural network will then be output and saved into a file or database.

## My clients

Users can be of any age, the category of users is broad as the idea of the program can be applied in many situations, an average user may want to google a word or phrase, a company might want to scan their documents, or a scientist may want to analyse their data.

## Current solutions

There is code developed for the entire program and parts of the program, libraries like PyTesseract make it quite easy as the entire development of the machine learning and neural network model is skipped, this allows focus to be shifted to the UI development.

**Machine learning:**

There are also many solutions that detail how to design your own neural network, like the TensorFlow tutorial for recognizing handwritten numbers using the MNIST dataset. From this tutorial you can easily re-purpose what you have learnt to recognize both hand-written numbers and letters, taking letter data from the A-Z Kaggle dataset. There are also many people that post their own solutions creating models with different combinations of layers and techniques to speed up the process of identifying the characters.

The simplest libraries for creating, and designing neural networks are PyTorch and TensorFlow. They both offer different benefits.

TensorFlow:

* Easy to write, it is quite beginner friendly as it uses the Keras API which has very ‘obvious’ implementations, the different layers have very clear-cut application and the method to define it is quite straight forward. The API also allows for higher level methods and practices, where they allow the creating of custom-made layers if necessary. Classes and functional methods can also be used to implement models.
* There is a large amount of functionality present in the library, it has pre-installed datasets for testing and experimentation, it has a large variety of optimizers and activation functions that can readily be used by the developer.
* Extensive documentation
* It is developed by Google
* Only CUDA support

<https://www.tensorflow.org/about>

PyTorch:

* Classes are heavily used, where the programmer defines the model in a more direct manner compared to TensorFlow. Here the programmer has more control over the fine details of the neural network.
* Extensive documentation
* Open Source
* Allows AMD GPUs to be used on Linux

<https://pytorch.org/features/>

**Character detection:**

The main library to use for character detection is Open-CV, there are alternatives however the library is used as standard across computer vision, e.g., it is utilized in PyTesseract which is developed by google.

The documentation and the resources for the library are very extensive as they detail how to use each part of the library using though examples.

<https://opencv.org/about/>

**UI:**

The UI is being developed using the Tkinter library, this library is very suited for simple window-based applications.

## Limitations:

There are many limitations for this project. Firstly, the neural network may make the wrong prediction, this is a worry as the dataset that I am using may not be very comprehensive in the different variations of handwriting styles. Secondly, the neural network cannot recognize any symbols other than the 26 letters of the English alphabet and the digits from 0-9, this means the recognition is extremely limited and cannot be used for mathematical problems or languages that use a different lettering system, like Mandarin or Hindi.

## Diagram of current systems:

### Character detection:

Diagram

Description automatically generated

Neural Network:

Diagram

Description automatically generated

# Design:

## Key points of the solution:

### Neural Network:

**Dataset:**

MNIST, KAGGLE A-Z datasets are the datasets I decided to use. They are datasets that are derived from the NIST special database 19 which is officially backed by the US government. They have variations of all 36 characters, 26 English alphabet and 10 digits, they are in a 28x28 pixel format and have already had the canny filter applied, colour values are 1 or 0, reducing the need for postprocessing. The MNIST dataset comes packaged in with the TensorFlow library and the Kaggle A-Z dataset needs to be installed from the Kaggle website.

**MNIST:** 60 000 images

<http://yann.lecun.com/exdb/mnist/>

**Kaggle:** 372 451 images

<https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format>

**NIST 19:** <https://www.nist.gov/srd/nist-special-database-19>

**Layer composition:**

There are many unique designs/selections of layers for a neural network, the 2 simplest implementations for this project are, a mix of Convolutional layers and Dense layers or just Dense layers.

The most basic, and my original, implementation for this project is using purely Dense layers. Here the neural network will take the colour values of each of the pixels in a 28x28 pixel image after it has been pre-processed, I.e., 784 inputs being either 1 or 0. These inputs will then be transformed, they will be multiplied by a weight and summed with a bias, this is the Dense layer. The Dense layers weight and bias are determined by an activation function, this function usually dictates the method in which each value is used. The initial model I started with consisted of a ‘Flatten’ layer, which takes the 28x28 image and makes it into an array of dimensions 784x1, and 2 Dense layers, the activation for this was a Sigmoid function. The example below shows the bare bones model with a ReLu activation function

**Basic Dense NN in tensorflow:**

Source Code Example:

import tensorflow as tf

model = tf.keras.Sequential([

    tf.keras.layers.Flatten(input\_shape=(28, 28)),

    tf.keras.layers.Dense(128, activation='relu'),

    tf.keras.layers.Dense(10)

])

Output:

Text

Description automatically generated

This method is very inefficient as the neural network activates each pixel leading to increased computing times. Dense layers are also not specifically designed to deal with images, they are more general purpose and tend to be used at the end to make the prediction while specialized layers condense and refine the data to reduce processing time and computer resources. Therefor the Convolutional layer is used.

Convolutional Layers are designed to be used on images and computer vision, the number of weights per layer is a lot smaller, which helps a lot with high-dimensional inputs such as image data.

**Basic Convolutional NN:**

Source code:

CNN = model = tf.keras.Sequential([

        # Convolutional layers

        tf.keras.layers.Conv2D(32, (5, 5), input\_shape=(28, 28, 1), padding='same', activation='relu'),

        tf.keras.layers.Conv2D(32, (5, 5), input\_shape=(28, 28, 1), padding='same', activation='relu'),

        tf.keras.layers.MaxPooling2D((2, 2)),

        tf.keras.layers.Conv2D(64, (3, 3), input\_shape=(28, 28, 1), padding='same', activation='relu'),

        tf.keras.layers.Conv2D(64, (3, 3), input\_shape=(28, 28, 1), padding='same', activation='relu'),

        tf.keras.layers.MaxPooling2D((2, 2)),

        # flattening the array

        tf.keras.layers.Flatten(),

        # applying the FC layers

        tf.keras.layers.Dense(64, activation='relu'),

        tf.keras.layers.Dense(26, activation='softmax')

    ])

Output: Text

Description automatically generated

<https://towardsdatascience.com/ultimate-guide-to-input-shape-and-model-complexity-in-neural-networks-ae665c728f4b>

<https://towardsdatascience.com/convolutional-layers-vs-fully-connected-layers-364f05ab460b>

## Character extraction:

**Pre-Processing:**

The images need to be pre-processed before they can be fed into the Neural Network, this means applying filters.

Gray Scale – the colours are from 1 to 0 decimals included

Binarization - Changing all colour values in the image from 3 channel RGB to 1 or 0 where 1 is black and 0 is white, this is done using the threshold function from OpenCV

Canny - this filter is applied for edge detection and is the main way to extract the edges from the image to use in the Neural network

“Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various [computer vision](https://en.wikipedia.org/wiki/Computer_vision) systems. Canny has found that the requirements for the application of [edge detection](https://en.wikipedia.org/wiki/Edge_detection) on diverse vision systems are similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible
2. The edge point detected from the operator should accurately localize on the centre of the edge.
3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges.” *- Wikipedia*

<https://en.wikipedia.org/wiki/Canny_edge_detector>

**Finding Contours:**

OpenCV “findContours” is used to locate the contours from the Canny image, The function retrieves contours from the binary image using the algorithm the user specifies, e.g., RETR\_EXTERNAL

**Sorting and resizing contours:**

Out of all the contours that are found I need to sort the contours out, I need to select the contours that I can pass through my neural network, this is done by restricting the types of bounding boxes that can be created from all the contours found.

Lastly, I will resize the contours so that they conform to the dimensions that my neural network can handle, 28x28, using some image manipulation.

# Developing the coded solution:

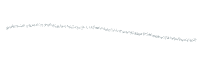
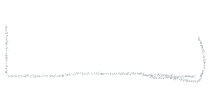
## Neural Network:

### Dataset:

The dataset needed to be retrieved, this is the MNIST dataset and the KAGGLE A-Z dataset, the MNIST dataset is already prebuilt into TensorFlow and the KAGGLE dataset can easily be downloaded from the KAGGLE website.

**MNIST dataset:** mnist = tf.keras.datasets.mnist

**KAGGLE dataset** – This dataset is not in the right/most efficient format, it can be installed as a .csv file form the Kaggle site, this however then needs to be turned into a NumPy array as the tensors in TensorFlow are not compatible with .csv formats. Doing this every time we need to retrain the model is very inefficient, therefor I found a small program that converts .csv into an array and edited it to be able to store these arrays as .npy files. I decided to include the MNIST dataset into the .npy files needed for the Kaggle dataset as that made everything easier and more manageable.

Text

Description automatically generated



**RED** – Loading dataset**|CURLY** – Resizing or combining the datasets**|GRAY** – Creating/Saving .npy file

The last thing I did for Datasets was creating a function that can create a dataset split between training data and test data.

Text

Description automatically generated

Before splitting the data, I am shuffling the data as it is in order, where all the A’s are at the front followed by the B’s, etc, this is done with the permutation function from NumPy.

Here I am using the train\_test\_split from the sk.learn library, another machine learning library, with the default split ratio being 3:1 for training data

There were multiple iterations of this, and was quite messy and inefficient at first, this is because I had not split the overarching problem of pre-processing the dataset into smaller parts making me try to complete all the data manipulation at once making it especially hard to follow and fix. Text

Description automatically generated

Text

Description automatically generated

I also made a Unit test for the dataset, this allowed me to check if the dataset was properly split into the test and train components

Graphical user interface, text

Description automatically generated

### Model design:

The model is the machine learning framework, it is the method in which the program will recognize what character is displayed, it is the core component of the project.

When stating with the project I had no experience in machine learning and as such there are multiple models of varying effectiveness employing different techniques.

**Model1 (Linear style):**

Text

Description automatically generated

This is the same as the example from the design section of this document, this is the most basic model, and it only predicts for numbers ranging from 0 to 9. It is comprised of a Flatten layer, which transforms a 2-dimensional array into a single dimension array, and 2 Dense, fully connected, Neural layers.

Image e.g. (with one output)



This model gave me fairly accurate predictions on the character being predicted. Results being: #insert results

After applying this model to the numeric dataset, I decided to implement the same model for the Alphabetic characters, this also yielded comparable results in terms of accuracy

#Insert results

Text

Description automatically generated

Next, I started to look at machine learning algorithms and techniques that are specifically used for image processing. Convolutional layers are not densely connected, not all input nodes affect all output nodes. This gives convolutional layers more flexibility in learning. Moreover, the number of weights per layer is a lot smaller, which helps a lot with high-dimensional inputs such as image data.

When a model uses Convolutional layers, it requires some specific supporting layers and features.

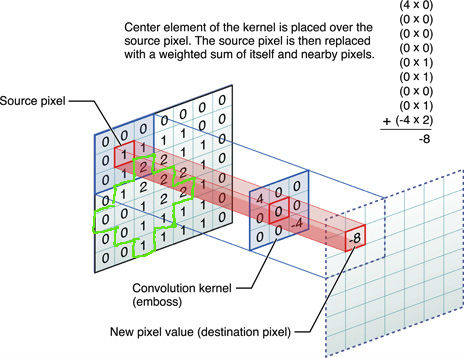
**Model 2 (Convolutional) (current model):**

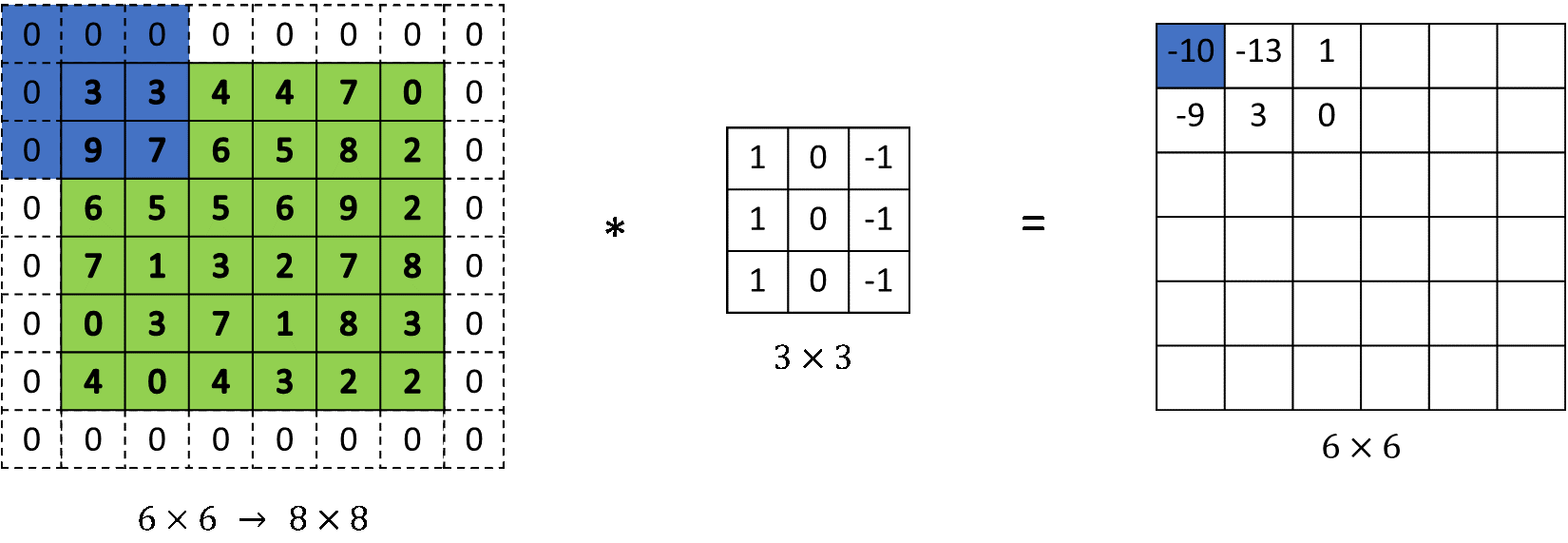
The kernel, window, iterates over the image until each pixel has been covered. A dot product between the pixels included in the window and the window are taken, the result of this dot product is known as a ‘feature’. A ‘feature map’ is a collection of ‘features’. Several types of kernels have different values in the matrix.

Convolutional layers also have a certain padding added to the edges of the image passed through, this is because when the kernel is centred on a corner pixel 1 side or more of the kernel is going to be out of the image, this can cause errors.

First – CNN

Second – Padding





Max-Pooling:

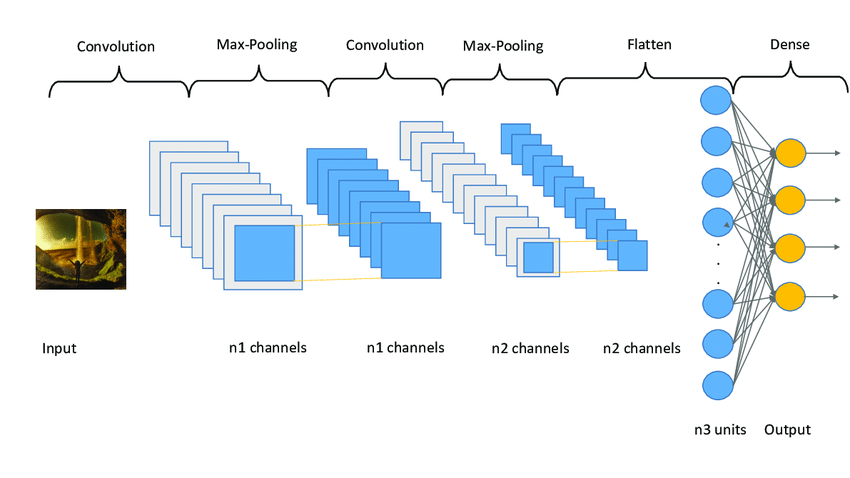
After a convolution has been applied to the data and a feature map has been created the feature map is passed through a Max-Pooling layer.

The Max-Pooling layer takes the largest value in each patch of the feature map; this is to reduce the total number of samples but have the most comprehensive overview of all the key features in the image.



Dense layers:

After the convolutions have been applied the final feature map after Max-Pooling is sent through the Dense, fully connected, layers so that the predictions can be made. Here I used the same framework as before just changing the number of resulting predictions from 10 to 36(0-9, A-Z).



Text

Description automatically generated

Text

Description automatically generated

<https://towardsdatascience.com/convolutional-layers-vs-fully-connected-layers-364f05ab460b>

### Training Process:

**Previous method:**

**Text

Description automatically generated**

**Text

Description automatically generated**

Nr of lines: 108

**Parser:**

A screenshot of a computer

Description automatically generated with medium confidence

**Storing weights and biases:**

Text

Description automatically generated

**Instantiating model, dataset, and class names:**

Text

Description automatically generated

**Training loop:**

Text

Description automatically generated

Nr of lines: 70

When run:





### Character detection:

The program needs to be able to find the characters it will convert from text format into string data. For this I need to create or find a sub-program that will efficiently detect all the handwritten characters in an image.

### Pre-Processing:

Before the characters can be extracted from the image, I first need to apply filters to make the contours and shapes easier to detect. Specifically, I need to apply filters on the image so that it can match up to the images used for training the neural network.



First, I Gray scale the image this is to make it so that the pixel values are in a range between 1 and 0, meaning they can be float values, this will also help with thresholding the image. After I threshold the image, this means that the pixel values above a threshold will be set to 255, black, and if they are below, set to 0, white. Next, I apply a gaussian blur this will help the program detect the edges of the characters in the image. Lastly, I apply a Canny filter to clearly outline any edges clearly.

Original: Canny:

A picture containing text, building, window

Description automatically generatedGraphical user interface

Description automatically generated with low confidence

Feature extraction:

Now that the image has been prepared the I use the findContours function built into the OpenCV library, this will find all the connected contours, the white lines, in the canny image. It will find them and store them in an array.

Then these contours will be filtered based on contour width and height as the letters will be smaller contour size, this is done by creating “Bounding boxes” around the contours.

A for loop iterates through the elements in the array called ‘cnts’. Each element in the array is a contour, and then the contour has a bounding box created for it, this will specify the width, height, x, and y coordinates. After the bounding boxes are separated based on their width and height. If they are within a certain threshold they will deemed as a ‘character’. Then the character will be extracted from the image

Text

Description automatically generated

Text

Description automatically generated

### User Interface:

#Work in progress

Graphical user interface, text

Description automatically generated

# Evaluation:

The requirements of the success criteria have been partially met. The machine learning component of the program has been fully developed and should be fully functional, it allows for the program to take arguments from the command line and then execute the functions properly. It loads the models previous weights, and it can also override the previous training weights if the user wishes (by default it loads previous weights).

Character detection is different, the character detection was the most difficult. Finding and extraction characters in the image requires that the program be able to distinguish between what a character is and what is not a character, this is the problem, my program is able to find all the edges and extract all the edges however it is difficult for it to then lift only the character out of the image. This issue is far less prominent in font-based text, where the program is easily able to extract letters that are written in a digital font, however when this font character is passed though the neural network it causes major errors, the output is often that all characters are F or Y.

The user interface is the #Work in progress