AARHUS UNIVERSITY

Exam Portfolio for Visual Analytics

Link to GitHub repository: https://github.com/bokajgd/Visual-Analytics-Exam

May 27th 2021

Jakob Grøhn Damgaard, 201808996
Visual Analytics – Teacher: Ross Deans Kristensen-McLachlan.

Table of Contents

Introduction

This document constitutes my collection of assignments from Spring 2021 course *Visual Analytics*, part of the bachelor's elective in Cultural Data Science. These assignments all revolve around conducting computational analysis of visual (image) data with varying purposes in Python (Van Rossum & Drake, 2009). Four assignments were assigned by the teacher throughout the semester while the last one comprises a self-assignment project. All source scripts for each assignment are coded according to principles of object-oriented programming. Hence, each script consists of a class that houses different functions and is initialized when the script is executed from a command line.

General Instructions

Link to overall repository: https://github.com/bokajgd/Visual-Analytics-Exam

This section provides a detailed guide for locally downloading the code from GitHub, initialising a virtual Python environment, and installing the necessary requirements. In order to maximise user-friendliness and ease the processes mentioned above, all projects have been collated into one single GitHub repository. Therefore, all code can be fetched into one local folder and as there are no overlapping dependencies, only a single virtual environment is needed. Please note, a local installation of Python 3.8.5 or higher is necessary to run the scripts.
To locally download the code, please open a terminal window, redirect the directory to the desired location on your machine and clone the repository using the following command:

```
git clone https://github.com/bokajgd/Visual-Analytics-Exam
```

Then, proceed to execute the Bash script provided in the repository for initialising a suitable virtual environment:

```
./create_venv.sh
```

This command may take a few minutes to finalise since multiple packages and libraries must be collected and updated. When it has run, your folder should have the following structure (folder depth of 2):

```
.
├── A2-Histogram-Comparisons
│   ├── README.md
│   ├── data
│   ├── output
│   └── src
├── A3-Edge-Detection
│   ├── README.md
│   ├── data
│   ├── output
│   └── src
├── A4-Image-Classification
│   ├── README.md
│   ├── graphics
│   ├── output
│   ├── src
│   ├── test_images
│   └── user_manual
├── README.md
├── SA-Unbuilding-LEGO
│   ├── README.md
│   ├── data
│   ├── output
│   └── src
├── Visual_Analytics_Exam.pdf
├── create_venv.sh
└── requirements.txt
```

You can verify this structure by running the following command:

```
tree -I vis_analytics_venv -L 2
```

If everything checks out, you should be ready to execute the code scripts located in the respective assignment folders. This is the base folder from which you are advised to navigate back to whenever you are finished running and assessing the scripts in one of the assignment folders. Always remember to activate the virtual environment before executing scripts in the terminal command line. This command can only be executed when you are located in the main folder.

```
source vis_analytics_venv/bin/activate
```

The same goes for deactivating the environment when use is ceases:

```
deactivate
```

All code has been tested in Python 3.8.5 on a 2020 MacBook Pro 13'', 2 GHz Quad-Core Intel Core i5, 16 GB Ram running macOS Big Sur (11.2.6). Thus, following the instructions should allow for smooth execution on MacOS and Linux machines. If you are running the code in any other operating system, please adapt commands to that syntax.

More detailed instructions regarding the execution of the individual scripts can be found in the sections on the respective assignments.

A2 – Histogram Comparisons

Link to the assignment folder (embedded in the overall repository) can be found here: https://github.com/bokajgd/Visual-Analytics-Exam/tree/main/A2-Histogram-Comparisons

Description

Colours constitute key markers for the visual system when faced with tasks such as object recognition and memory consolidation (Wichmann and Sharpe, 2002). Furthermore, colours can be numerically represented in e.g., a 3D RGB colour space. Thus, by being both relevant neural features and computationally manageable, colours are an obvious feature to focus on when performing a simple analysis on visual data. As an example, image similarity can be analysed by calculating the differences in the colour composition between two different images. To conduct such an analysis, the various colour nuances present in an must be quantified using a 3D colour histogram. Such a histogram represents the distribution of colour present in an image.

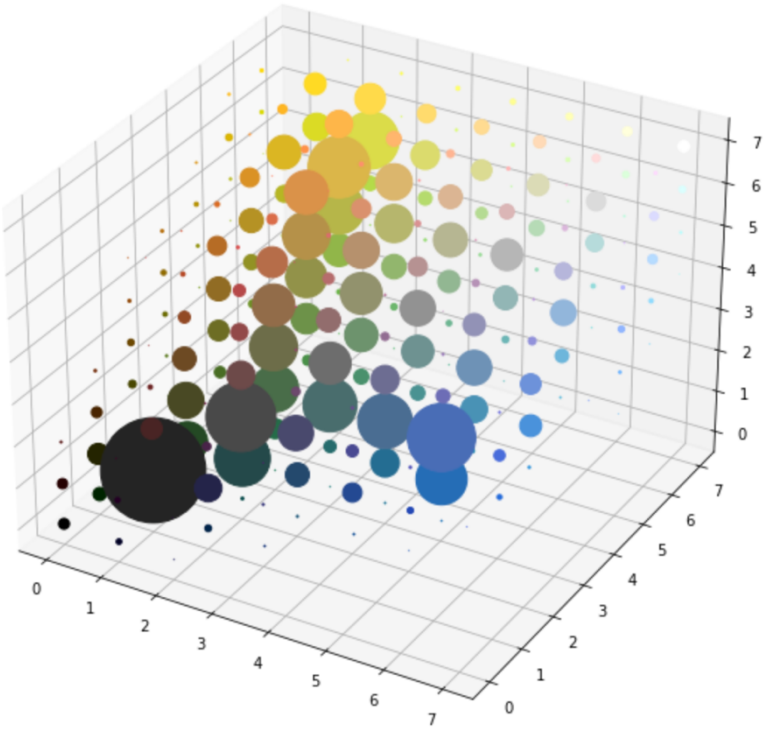

Figure 1 - 3D visualisation of a 3D colour histogram produced by the notebook from class 3 (https://github.com/CDS-AU-DK/cds-visual/blob/main/notebooks/session3.ipynb)

For this assignment, we were provided with a data set consisting of images of 17 different common British flowers. We were asked to compare the 3D colour histogram of a self-chosen target image with each of the other images in the corpus one-by-on using chi square distance as a similarity measure. More specifically, the task was to produce a script that, for a given input image, outputs a single *.csv* file containing a column for the filenames of the compared images and a column with the corresponding distance scores. Lastly, the script should print out the filename of the image found to be most similar to the input target image.

The data set contains 80 images for each category. Note that due to storage constraint, only 250 images have been uploaded to GitHub. You are free to download the full dataset locally using this link.

Usage

If not already open, open a terminal window and redirect to the home folder of the cloned repository (see *General Instruction*). Remember to activate the virtual environment. Then, jump into the folder called *A2-Histogram Comparisons* using the following command:

```
cd A2-Histogram-Comparisons
```

Now, it should be possible to run the following command in to get an understanding of how the script is executed and which arguments should be provided:

```
# Add -h to view how which arguments should be passed
python3 src/A2-Histogram-Comparison.py -h

usage: A2-Histogram-Comparison.py [-h] [-ti --target_image]

[INFO] Image similarity using color histograms

optional arguments:
  -h, --help          show this help message and exit
  -ti --target_image  [DESCRIPTION] Name of the target image
                      [TYPE]        str
                      [DEFAULT]     image_0001
                      [EXAMPLE]     -ti image_0001
```

It should now be clear that the script can be executed using the following command.

```
python3 src/A2-Histogram-Comparison.py -ti image_0001
```

If there is no input to the *-ti* argument, *image_0001.jpg* is used as the default target image. Feel free to choose your own target image - note that it must be located in the data folder. As the script is already specialised to this specific assignment and this specific dataset, I've simplified the command line argument as much as possible to increase user-friendliness.

Therefore, please note that one only has to input the image name of the desired target image and no file path or *.jpg* suffix is needed. This also means that the script, in its current state, only supports *.jpg* files as input.

Structure

The structure of the assignment folder can be viewed using the following command:

```
tree -L 2
```

This should yield the following graph:

```
.
├── README.md
├── data
│   └── jpg
├── output
│   └── chi_sqr_comparisons_image_0001.csv
└── src
    └── A3-Histogram-Comparison.py
```

The following table explains the directory structure in more detail:

Folder	Description
`data`	A folder containing the data set used for the analysis. In this folder, the subfolder *jpg* holds 1382 *.jpg* files along with a *.txt* file listing all the filenames.
`src`	A folder containing the *.py* script (*A2-Histogram-Comparison.py*) created to solve the assignment.
`output`	A folder containing the output produced by the Python script. The script yields a *.csv* file with the file name *chi_sqr_comparisons_**<image name>**.csv*

Methods

As stated, the script is coded using the principles of object-oriented programming. The main class of the script includes an __init__ method which holds the set of statements used for solving the desired tasks. This collection of statements is executed when the class object is created. Furthermore, the class holds a series of utility functions which are called when needed in the __init__. The class object is created - and thus, the tasks are performed - whenever the main function is executed. This happens every time the module is executed as a command to the Python interpreter.

The script takes the name of a target image as an input variable, reads in this image, and creates a 3D colour histogram with 8 bins in each dimension. It then obtains the paths for all

the image file located in the *data* folder. Looping through all the images in the list paths one-by-one (and skipping a file if the path is identical to that of the target image), the script proceeds to generate a 3D colour histogram for each respective comparison image before calculating the chi square distance between it and the target image histogram. This value along with the given file name is then appended as a row to a Pandas (McKinney, 2010) data frame, which is exported as *.csv* file to the *output* folder when the loop has finished. Finally, the script prints the name and chi square value of the image with the highest similarity. Note that to make images directly comparable and to account for outliers, varying light intensity, image sizes etc., the histograms are normalized using min-max normalisation before similarity is calculated.

The main library utilised for this assignment is OpenCV (Bradski, 2000).

Discussion of Results

The script runs swiftly and outputs a tidy data frame as expected. When running the script with *image_0001* set as the target image, the most similar image is found to be *image_0597* which has a chi square distance of 1242.

```
python src/A2-Histogram-Comparison.py -ti image_0001
The most similar image is image_0597.jpg
This image has a chi square distance of 1242
```

Figure 2 - When image_0001 is set as the target image, image_0597 is found to be the most similar image in the dataset when analysing 3D colour histograms

From this example run, it is apparent that the algorithm to a certain degree captures similarity between images; the target image and the image with the lowest chi square value both depict yellow flowers photographed with a mixed green/brown background. However, color histograms only capture the proportion of the number of colours in an image and do not account for the spatial location of the colors. The target image comprises several flowers and the flower visible in *image*_0597 - though yellow - is clearly not of the same species. Hence, the method will often be insufficient to solve a more complex task like object recognition and serves better when used in combination with other visual analysis approaches.

A3 – Object Segmentation using Edge Detection

Link to the assignment folder (subfolder of the overall repository) can be found here: https://github.com/bokajgd/Visual-Analytics-Exam/tree/main/A3-Edge-Detection

Description

Edge detection refers to the notion of identifying boundary points in an image where the light or colour intensity changes drastically (Davis, 1975). For this assignment, we aim to utilize edge detection algorithms to perform object segmentation/detection on an image. The end goal, thus, is to identify and draw contours around perceptually salient segments of an image. Once all separate objects in an image have been identified, they can e.g., be classified using an object recognition algorithm.

More specifically, we were provided with a large image of a piece of text engraved onto the Jefferson memorial. The image can be found by clicking this link. The purpose is to detect and draw contours around each specific letter and language-like object (e.g. punctuation) in the image. The tasks were as follows (taken directly from assignment description):

- Draw a green rectangular box to show a region of interest (ROI) around the main body of text in the middle of the image. Save this as ***image_with_ROI.jpg***.
- Crop the original image to create a new image containing only the ROI in the rectangle. Save this as ***image_cropped.jpg***.
- Using this cropped image, use Canny edge detection to 'find' every letter in the image
- Draw a green contour around each letter in the cropped image. Save this as ***image_letters.jpg***

Usage

If not already open, open a terminal window and redirect to the home folder of the cloned repository (see *General Instruction*). Remember to activate the virtual environment. Then, jump into the folder called *A3-Edge-Detection* using the following command:

```
cd A3-Edge-Detection
```

Now, it should be possible to run the following command in to get an understanding of how the script is executed and which arguments should be provided:

```
A3-Edge-Detection % python3 src/A3-Edge-Detection.py -h
usage: A3-Edge-Detection.py [-h] [-inp --input]

[INFO] Image similarity using color histograms

optional arguments:
  -h, --help    show this help message and exit
  -inp --input  [DESCRIPTION] Name of the file of the input image
                [TYPE]        str
                [DEFAULT]     text_image.jpeg
                [EXAMPLE]     -inp text_image.jpeg
```

It should now be clear that the script can be executed using the following command.

```
# With input image specified
python3 src/A3-Edge-Detection.py -inp text_image.jpeg

# Without specification of input image
python3 src/A3-Edge-Detection.py
Input image file name is not specified.
Setting it to 'text_image.jpeg'.
```

As can be seen, if there is no input to the *-inp* argument, *text_image.jpeg* is used as the default input image. It is possible to input a self-chosen image (takes any file type supported by *cv2.open()* function) that has been placed in the *data* folder. However, as the script is already specialised to the specific image provided for the assignment results (pre-defined ROI coordinates), it will likely produce futile results.

Structure

The structure of the assignment folder can be viewed using the following command:

```
tree -L 2
```

This should yield the following graph:

```
.
├── README.md
├── data
│   └── text_image.jpeg
├── output
│   ├── image_with_ROI.png
│   ├── image_with_contours.png
│   └── only_ROI.png
└── src
    └── A3-Edge-Detection.py
```

The following table explains the directory structure in more detail:

Folder	Description
`data`	A folder containing the data used for the analysis. In this folder, the image provided for analysis, *text_image.jpeg* is located.
`src`	A folder containing the *.py* script (*A3-Edge-Detection.py*) created to solve the assignment.
`output`	A folder containing the output produced by the Python script. The script generates three images: - *image_with_ROI.png*: An image with a green border showing the pre-defined ROI - *image_with_contours.png*: An image with green contours drawn around all letters detected in the image - *only_ROI.png*: An image of only the ROI cropped out of the original image

Methods

Akin to the script in assignment 2, the script is coded using the principles of object-oriented programming. See the first paragraph of the previous methods section for a quick outline of the general script architecture.

All image processing in the script is performed using OpenCV (Bradski, 2000). After loading in the input image, a version with a green border drawn around the engravings is generated and saved in the output folder. This area is then cropped out of the original image which is also saved. In order to draw contours around the letters, the cropped image is converted into a greyscale color space. A Gaussian blur filter with a kernel size of 7x7 is then applied in order reduce noise in the image. If too much smoothing is induced, the desired edges will become undetectable. Next, the image is transformed using simple binary thresholding with a static threshold set at 115 (yields best result). This converts any pixels with a brightness value below 115 to 0 and pixels with a brightness value above 115 to 255. Lastly, canny edge detection is applied with a high threshold value of 30, a low threshold value of 150 and the kernel size set to the default 3x3.

This enables drawing green contours around the letters. Only extreme outer contours are retrieved and a contour approximation method (*cv2.CHAIN_APPROX_SIMPLE)* is applied to compress the contours. The final image is the saved to the output folder.

Discussion of Results

The below image displays the result of the full letter detection process (other output images can be found in output folder). The process appears to have been rather successful and most letters are correctly identified and highlighted with coherent contours. It is, however, not a flawless result and it seems to have struggled with excluding the inner regions of letters such as *D* and *O*. This problem could perhaps be alleviated by adjusting the kernel sizes of the blurring filter or during the canny edge detection. Furthermore, a few cracks in the wall has also been identified.

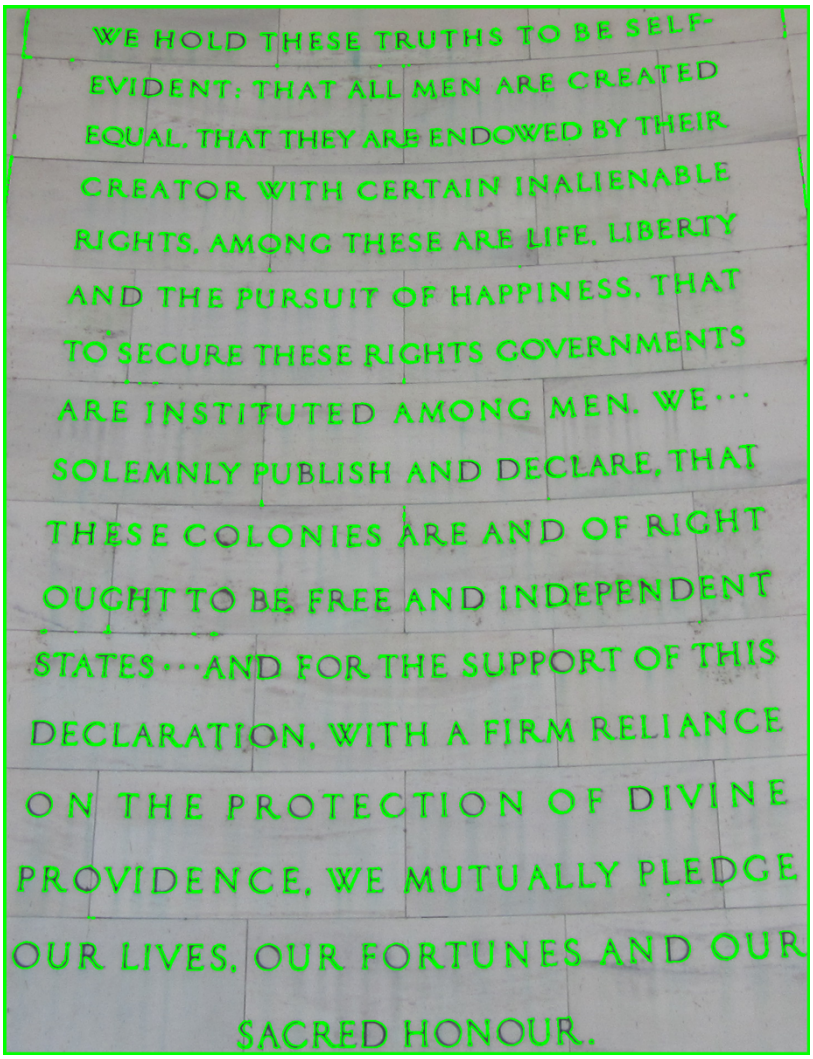

Figure 3 - Image of the cropped-out ROI with contours draw around the detected letters

In general, the script could be favourably updated to be usable on any input image by adding the ability to determine the ROI coordinates in the command line. Furthermore, it would also be relevant to improve the script by giving the user control over various parameters such as thresholding algorithms, kernel sizes and thresholds through command line arguments. This would enable the user to flexible adjust the script to varying image compositions and object detection tasks. It could e.g., be interesting to see if the result could be improved through the use of adaptive thresholding.

A4 – MNIST Image Classification GUI

Link to the assignment folder (subfolder of the overall repository) can be found: https://github.com/bokajgd/Visual-Analytics-Exam/tree/main/A4-Image-Classification

Description

Image classification is a corner stone in the field of computer vision and visual analytics. The MNIST database is a dataset consisting of 70.000 images (28x28 pixels) of handwritten digits that is commonly used as example-data for training and testing machine learning image classification algorithms (LeCun et al., 2010). For this assignment we were tasked to create two command-line tools able to perform digit classification on the MNIST data. One script should train and test a simple logistic regression classifier and the second script should train and test a deep neural network model.

As I already had experience with coding the required tasks, I decided to create an interactive tool that could give a user more intuitive, hands-on insight into the workings of both algorithm types and their outputs. This project culminated in an interactive graphical user-interface coded in Python using *tkinter* which allows the user to play around with hyperparameters and train and test various logistic regression and convolutional neural network-based digit classifiers. Furthermore, the app allows the user to upload an image from their computer and let their self-designed neural network predict which number it believes to be depicted in the image. Follow the user-guide in the following section to gain a deeper understanding of the functions embedded in the GUI.

Initiate GUI

If not already open, open a terminal window and redirect to the home folder of the cloned repository (see *General Instruction*). Remember to activate the virtual environment. Then, jump into the folder called *A4-Image-Classification* using the following command:

```
cd A4-Image-Classification
```

The GUI can now be. initiated by running the following command. This will open a new window on your computer.

```
python3 src/digit_classification_GUI.py
```

See user-manual for instructions on how navigate the GUI and use its functions.

Structure

The structure of the assignment folder can be viewed using the following command:

```
tree -L 2
```

This should yield the following graph:

```
.
├── README.md
├── graphics
│   ├── bgHome.gif
│   └── bgStandard.gif
├── output
│   ├── 2-dense-8,4-nodes-CNN-viz.png
│   ├── 2-dense-8,4-nodes-CNN.model
│   ├── l2-penalty-0.1-tol-nodes-LR-viz.png
│   └── latest-prediction.png
├── src
│   ├── digit_classification_GUI.py
│   └── models
├── test_images
│   ├── test_image.jpg
│   ├── test_image_2.jpg
│   ├── test_image_3.jpg
│   ├── test_image_4.jpg
│   └── test_image_5.jpg
└── user_manual
    ├── 5a.png
    ├── 5b.png
    ├── 5c.png
    ├── 5d.png
    ├── 5e.png
    ├── 5f.png
    ├── 5g.png
    ├── 7.png
    └── overview.png
```

The following table explains the directory structure in more detail:

Folder	Description
`graphics`	A folder containing *.gif* files of the background graphics.
`output`	A folder containing the output produced by the when using the GUI. This includes a folder with trained CNN models, network graphs of the fully connected part of the CNN models, bar plot showing latest prediction on self-chosen image, visualisation of importance of input nodes in logistic regresision models.
`src`	A folder containing all the source code used for the project: • The *digit_classification_GUI.py* script contains the GUI code. • The *models* subfolder contains scripts for the logistic regression and convolutional neural network models that are utilised in by the GUI ○ The *model_utils* subfolder contains a script with utility functions used in the model scripts
`test_images`	This folder contains five test images that the user can use to test the *Classify New Image* function
`user_manual`	Contains all the images used for the user-manual

User-manual

This section provides a brief walk-through of the structure of the GUI and introduces the interactive functionalities to the user. *Figure 4* displays the overall architecture of the app, and the arrows indicate how the user is able to navigate between the different pages.

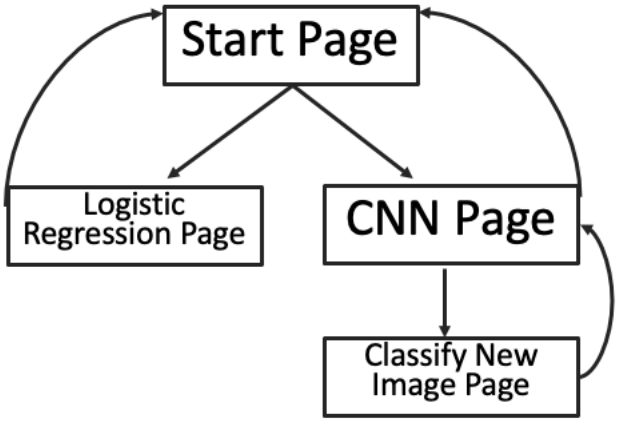

Figure 4 - Overall structure of the GUI. When the script is executed from the command line, the user lands on the 'Start Page'

When the GUI script is executed, a new window opens and the user lands on the **Start Page:**

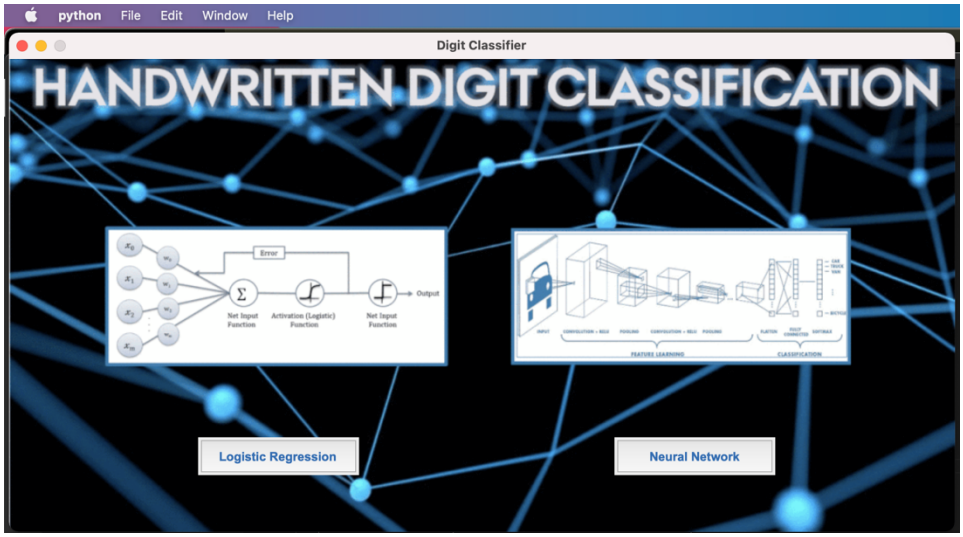

Figure 5a - Start Page

From here, the user has the option of navigating to the **Convolutional Neural Network Page** or the **Logistic Regression Page**. E.g. when the 'Neural Network' button is pressed, one is redirected to the **Convolutional Neural Network Page:**

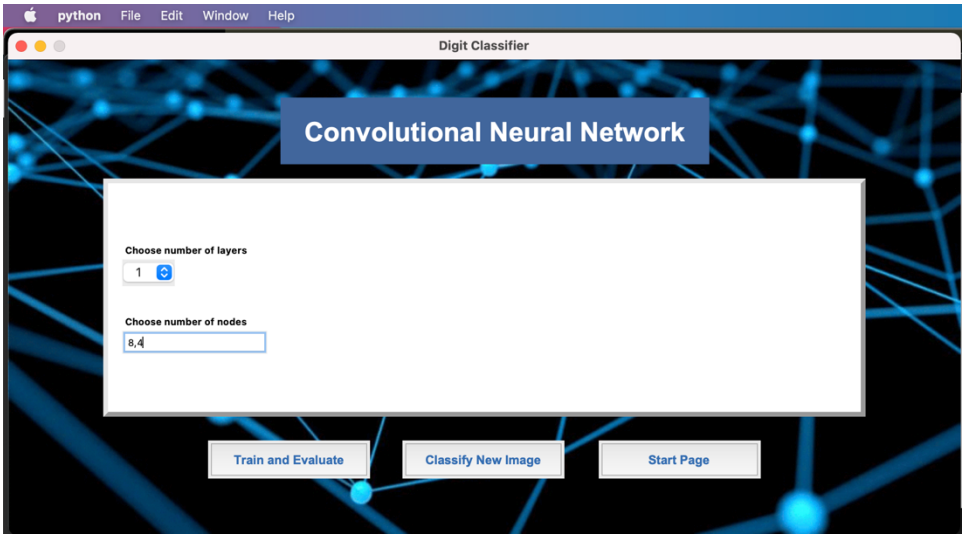

Figure 6b - CNN page before a model has been trained

On this page, the user is able to design his/her own convolutional neural network by tweaking the number of layers and nodes in fully connected part of the network. 2 convolutional layers along with a max-pooling and a dropout at layers are pre-defined in the CNN model script and cannot be adjusted. The dropdown allows the user to choose between 1, 2 or 3 fully connected layers. The user then has to fill in the desired number of nodes for the respective layers. The number of nodes for each layer should be separated by a comma. Whenever the 'Train and Evaluate' button is pressed, the designed model starts

training on the MNIST training data. All models are set to run through 3 epochs with a batch size of 64. Note that training takes a few minutes to complete, however, progress bars should appear in the terminal enabling the user to follow the process. Also note that a model will not initiate training if there is a discrepancy between number of chosen fully connected layer and the length of the list of number of nodes that are input.

When the model has finished training, its performance is tested on the test data and a table containing evaluation metrics is printed on the page. Additionally, a network graph showing the structure of the fully connected part of the network, as designed by the user, is drawn onto the page. The network graph and the trained model is saved to the *output* folder:

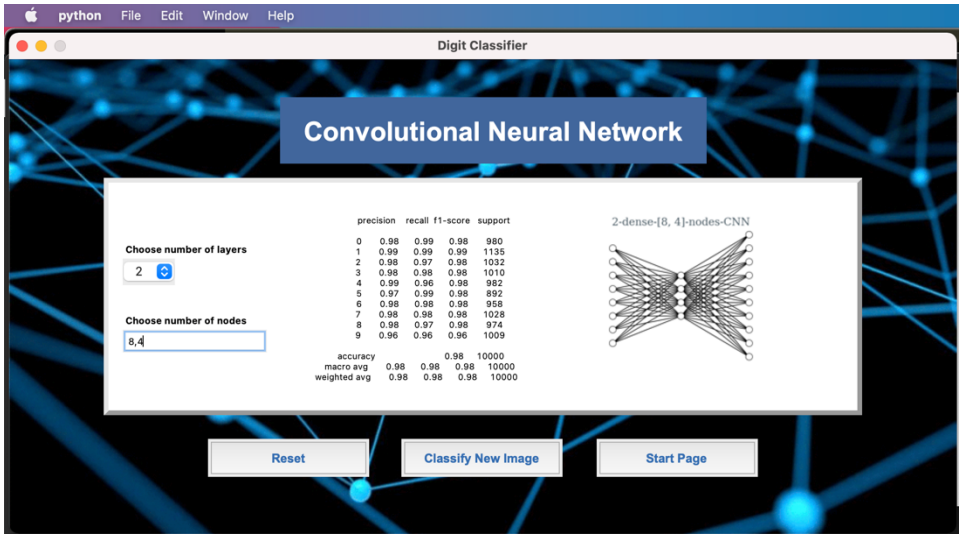

Figure 7c - CNN page after a model has been trained

Once a model has been trained and tested, the user has multiple options. He/she can either decide to reset the inputs and remove the plotted figures by pressing the 'Reset' button. He/she can also decide to return to the **Start Page** by clicking the 'Start Page' button. Lastly, the user can decide to proceed to a new page on which he/she can upload a self-chosen image file and let the most recently trained model make a prediction:

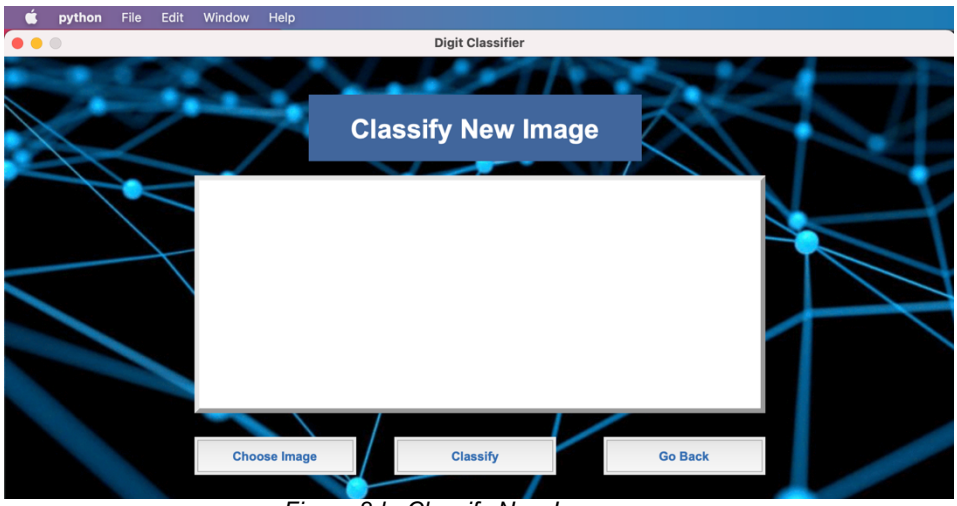

Figure 8d - Classify New Image page

Here, the user can choose an image from his/her computer by pressing the 'Choose Image' button. The *test_images* folder in the assignment folder holds 5 images which the user can use as test images for prediction. Once an image has been chosen, the user can let the model make a prediction by pressing the 'Classify' button. The page shows both the uploaded image along with a bar plot and a label showing the digit prediction and prediction certainty.

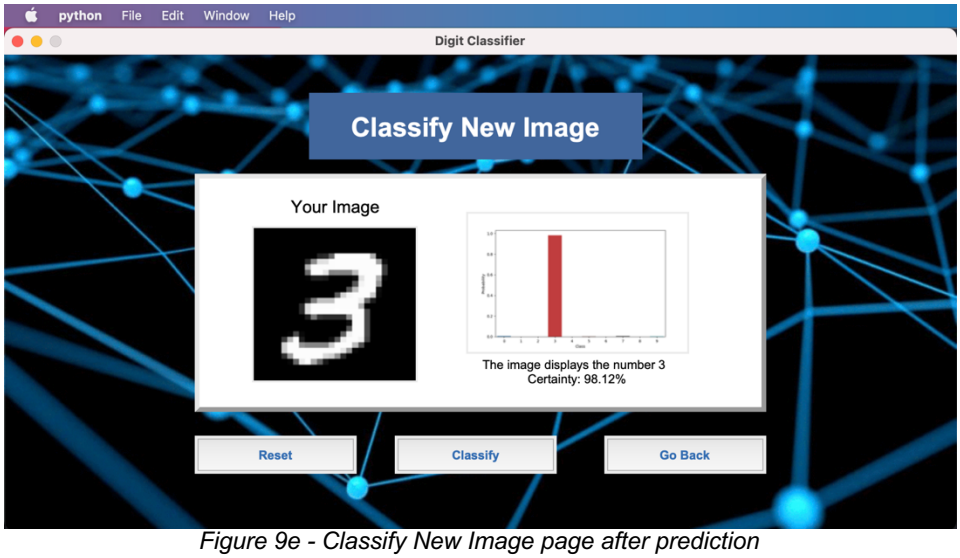

Figure 9e - Classify New Image page after prediction

By pressing the 'Go Back' button and then the 'Start Page' the user is redirected back to the start page. From here, the user can click the 'Logistic Regression' button and instead be redirected to the **Logistic Regression Page:**

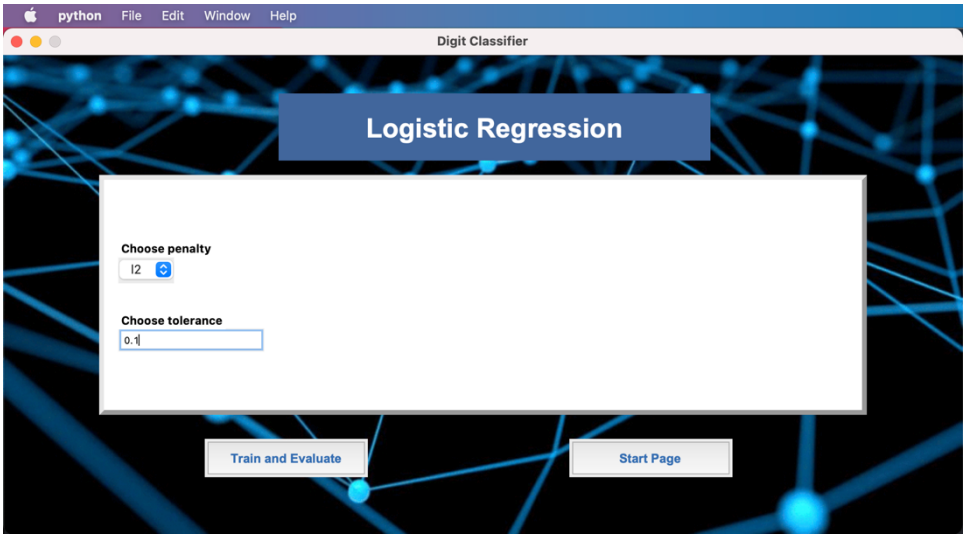

Figure 5f - Logistic Regression Page

On this page, the user is able to adjust the *penalty* and *tolerance* hyperparameters in a logistic regression classifier. The dropdown allows the user to implement either *l1*, *l2* or *no* penalty. If either *l1* or *l2* penalty is chosen, the user should set the tolerance level (takes a float). The tolerance indicates the stopping criterion for the model when searching for optimal parameters. When the 'Train and Evaluate' button is pressed, the specified logistic

regression model is trained and tested. Note that this may also take a minute or two. To minimise training time, the model only trains on 7500 images and is evaluated on a test set consisting of 2500 images.

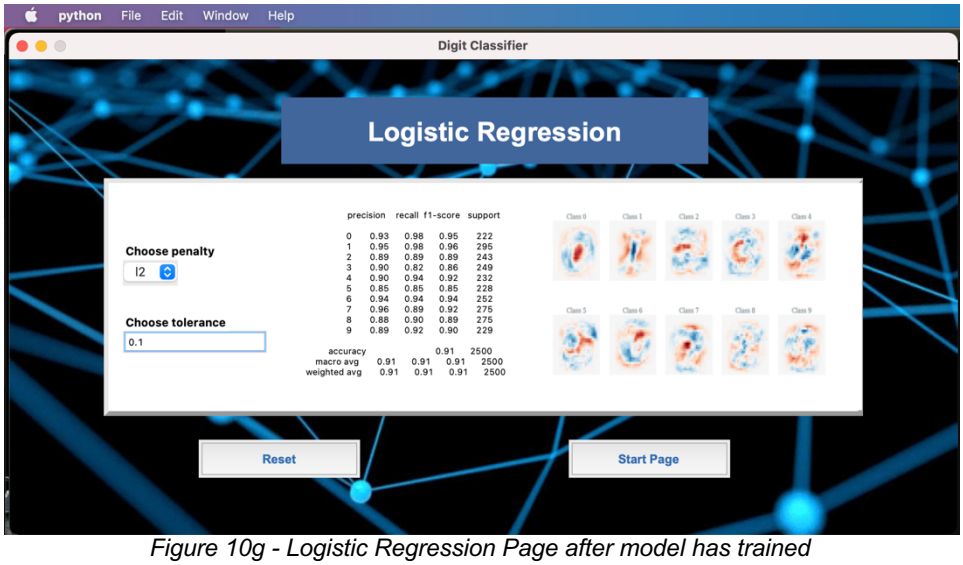

Figure 10g - Logistic Regression Page after model has trained

After model training, a table showing the evaluation metrics is display along with a visualisation of the most important input nodes for each digit class.

Methods

The *tkinter GUI* is coded using the principles of object-oriented programming (Lundh, 1999). The *MainApp* class initiates a container in which it stores and manages the four interactive page classes. The *MainApp* class also holds all utility functions employed in the page classes to control functions such as button presses. PIL (Clark, 2015), seaborn (Waskom, 2021) and matplotlib (Hunter, 2007) are all heavily used for plotting and image handling.

The logistic regression classifier is coded using functions and the MNIST data set collected from the Scikit-learn library (Pedregosa et al., 2011). To minimise compute intensity and user-waiting time, the model is only trained on 7500 images. The *sage* solver is used as the solver. The script outputs an evaluation report and a visualisation of the most influential input feature for each of the 10 output nodes.

The convolutional neural network in the *cnn_mnist.py* script is coded using the Keras interface for the TensforFlow library (Abadi et al., 2016). The network has two convolutional layers followed by a max-pooling layer and a dropout layer (dropout = 0.25). The hyperparameters for the fully connected layers are determined by the user in the GUI .
For training the model parameters, the *Adam Optimiser* was used as the optimiser and *categorical crossentropy* was chosen as the loss function. The model trains across three epochs with a batch size of 64. This may not be enough to fully optimise the network but it minimises the waiting time for the user. The script returns an evaluation report and outputs a network graph of the fully connected layers

Discussion of Results

The main purpose of this project was to create a user-friendly environment in which machine learning novices could gain fun and insightful, hands-on experiences with digit classification algorithms; both by interactively controlling some of the hyperparameters and by being introduced to intuitive visualisations. I believe that these criteria have been achieved successfully.

The GUI also enables one to gain an insight into the workings and shortcomings of the two different algorithmic approaches. Despite its simplistic nature and shallow model structure, the logistic regression classifier is able to classify the handwritten digits with an accuracy of up to 92%. This benchmark is, however, greatly exceeded by the convolutional neural network which can be tweaked to performs with an accuracy above 98%. When training numerous models with varying it also becomes apparent that the. hyperparameters for the fully connected layers have minor influence on overall model performance and that all inputs within reasonable limits yields model performing with an accuracy around 98%. Though fully connected layers can influence a model when parameterized cleverly, for a simple network like this, most power lies in the convolutional layers (Basha et al., 2020).

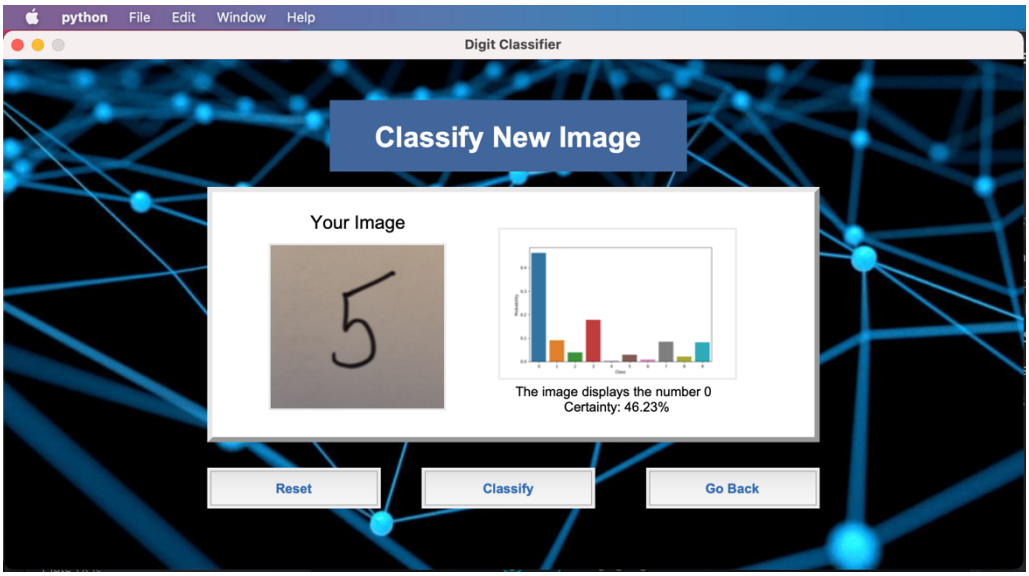

Figure 7 - CNN model having trouble with predicting the number 5

Lastly, the page on which the user can upload and test the model on a self-chosen image is ideal for examining how well models generalises to other images of handwritten digits. As seen in *figure 5e*, the model correctly predicts the digit with utmost certainty when faced with an image that is compositional identical to the images in the MNIST dataset. However, when faced with an image of a digit that is equally graspable to humans, but on e.g. a background with a different colour (as fan be seen in *figure 7*), the model is screwed and appears to be guessing at random. The MNIST dataset is extremely homogenous and, therefore, it is not able to generalise to images that have not been processed in the same manner.

Critical Evaluation

Though the GUI works as intended, there are numerous aspects that could be improved during future development:

- An obvious improvement/expansion would be to enable the user to adjust the convolutional layers of the CNN models. As anticipated, different fully connected architectures have little influence on the performance of the CNN; the power lies mostly in the convolutional layers.
- It would be nice to also enable classification of self-chosen images with the logistic regression models to examinate how well these models generalize.
- Last minute bug: The a (non-fatal) warning appears in the terminal when redirecting to the **Start Page** when no models have been trained as there are no 'panels' to delete
- It would be very nice include 'help' icons throughout the GUI with pop-up boxes appearing when hovering over with the cursor. This could e.g., provide guidelines for how to properly fill in the entry boxes for *tolerance* and *number of nodes*.
- Restructuring code: This project was a first time-attempt at coding a multi-page GUI with *tkinter* and it was admittedly more complex than anticipated. Thus, much of the code could be improved or streamlined by an expert developer. Many functions could be collated into more compact functions to provide more structure.
- One could easily change the model scripts to also make them executable on their own in the command line and make them output performance metrics in the terminal
- It would be meaningful to also print out model training development plots to monitor over/underfitting
- Lastly, it would enhance user experience to provide progress bars in the GUI during model training

Un-building LEGO® Creations using Computer Vision

Link to the assignment folder (subfolder of the overall repository) can be found:
https://github.com/bokajgd/Visual-Analytics-Exam/tree/main/SA-Unbuilding-LEGO

(LEGO® is a registered trademark of the LEGO Group, which does not sponsor, endorse, or authorise this. Visit the official LEGO® website at http://www.lego.com)

Description

For this self-assigned assignment, I decided to combine my passion for the wonderful toy LEGO® with my interest in coding to create fun and playful computer vision project. By expanding on two of the key pillars of this course, namely edge detection and image classification, I have attempted to create a module that is capable of detecting and classifying every single brick used in a LEGO®. Using object detection, the LEGO® model is segmented into its individual parts which are subsequently identified using a convolutional network. When people - no matter the age - see an awesome creation made out of LEGO® bricks, the first though that instantaneously springs to mind is: “Wow, how do you build that!?”. Designing and writing building instructions is a long and exhaustive manual process and few people take the time to do so. So wouldn’t it be mind-blowing if one had a piece of software that via concepts from machine learning and computer vision could, at least to a certain extent, disclose some of the techniques and bricks used in a beautiful model?

With this project, I attempt to crawl the humble first steps down the long path towards automating the process of deconstructing LEGO® models to figure out how they are constructed in a user-friendly and streamlined process.

The task I set myself can be demarcated into two subtasks.
1. Modelling, training and evaluating an image classifier able to distinguish between different types of LEGO® bricks.
For this task, I downloaded a data set of 40.000 artificially rendered images of 50 different types of LEGO bricks (800 different angles of each brick) from this link.

2. Segmenting an image of a LEGO® model into its separate parts and subsequently predicting their brick types by using the aforementioned classifier.

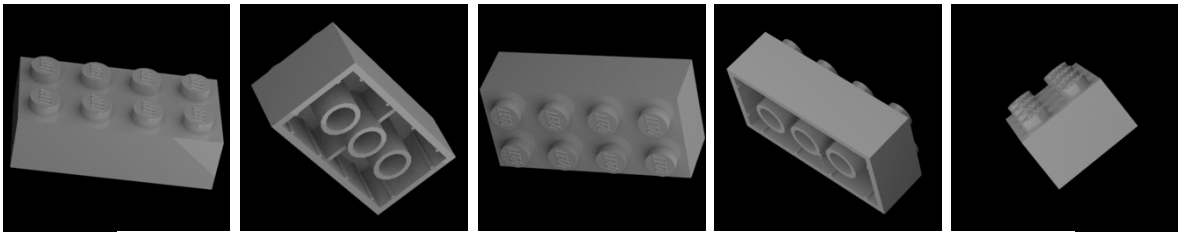

Figure 7 - Five images of a 2x4 brick from different angles

Usage

If not already open, open a terminal window and redirect to the home folder of the cloned repository (see *General Instruction*). Remember to activate the virtual environment. Then, jump into the folder called S*A-Unbuild-LEGO* using the following command:

```
cd SA-Unbuilding-LEGO
```

The main script relevant for execution is the *SA-Unbuilding-LEGO.py* script which is located directly in *src* folder. This script takes an image of a LEGO® model, partitions it into individual bricks and outputs a collage with classifications for all of the bricks. The module loads in the parameters from a pre-trained CNN that is trained in the *cnn/legoCNN.py* script on 4.200 images from the dataset. This script can also be tested if desired, however, only a minor subset of the data has been included in the GitHub repository and a folder with the optimal pre-trained model is already provided in the *outputs/model_outputs* subfolder.

To get an understanding of how *SA-Unbuilding-LEGO.py* be executed and which arguments should be provided, run the following command:

```
python3 src/SA-Unbuilding-LEGO.py -h
usage: SA-Unbuilding-LEGO.py [-h] [-inp --input_creation]

[INFO] Identifying Letters using Edge Detection

optional arguments:
  -h, --help            show this help message and exit
  -inp --input_creation
                        [DESCRIPTION] Name of the file of the input
image of a LEGO(R) creation
                        [TYPE]        str
                        [DEFAULT]     snake.png
```

The *snake.png* file is a digitally rendered image of a simple LEGO snake that is located in the *data/creations* subfolder and has been tested on the script. It is recommended that you use this default image for your first run, but feel free to play around with the script and upload an image of your own simple LEGO creation later.
The script can now be executed using the following command.

```
# With input image specified
python3 src/SA-Unbuilding-LEGO.py -inp snake.png

# Without specification of input image
python3 s src/SA-Unbuilding-LEGO.py
Input image file name is not specified.
Setting it to 'snake.png'.
```

Structure

The structure of the assignment folder can be viewed using the following command:

```
tree -L 2
```

This should yield the following graph:

```
├── data
│   ├── creations
│   ├── example_images
│   ├── test
│   └── train
├── output
│   ├── detected_bricks
│   ├── model_outputs
│   ├── snake_bricks_with_bboxes.png
│   ├── snake_bricks_with_contours.png
│   └── snake_detected_vs_predicted_bricks.png
└── src
    ├── SA-Unbuilding-LEGO.py
    └── cnn
```

The following table explains the directory structure in more detail:

Folder	Description
`data`	A folder containing the data used for the project. • The *creations* subfolder contains the image of the LEGO snake, *snake.png* (Built using Bricklink Studio 2.0) • The *example_images* subfolder contains standard images of each of the 6 types of bricks that the classifier has been trained on • The *test* and *train* folders respectively hold the *test* (480 images) and *train* (4.200 images)
`src`	A folder containing the source code for the project. • *SA-Unbuilding-LEGO.py* is the main script • The *cnn* folder contains the script for training and evaluating a cnn model on the train and test data

`output`	A folder containing the output produced by the Python scripts:
	• *detected_bricks*: This subfolder holds 400x400 pixel images of the individual detected bricks found in the analysed LEGO model on a black background to match the images from the data set
	• *model_outputs*: The subfolder holds outputs from the *legoCNN.py* script.
	  ○ *lego-CNN_2_epochs.model* folder contains pre-trained model
	  ○ *train_val_history_2_epochs.png* shows model training development over 2 epochs

	○ *name_of_classes.txt* holds a chronological list of the names of the brick types that the classifier can predict • *snake_bricks_with_bboxes.png* shows the input image of the snake with bounding boxes drawn around the individual bricks • *snake_bricks_with_contours.png* shows the input image of the snake with contours drawn around the edges of the individual bricks • *snake_detected_vs_predicted_bricks.png* is a collage showing the classifier predictions for each of the detected bricks

Methods

The convolutional neural network in the *legoCNN.py* script is coded using the Keras interface for the TensforFlow library (Abadi et al., 2016). The data set is pre-processed to suit the model structure by using the *tf.keras.preprocessing.image_dataset_from_directory()* function. Images are handled in a 1-channel grayscale colour space to minimise compute intensity as no images in the training set is coloured. The network has two convolutional layers followed by a max-pooling layer, a dropout layer (dropout = 0.25) and two fully connected dense layers and a final softmax layer for outputting predictions. As I quickly realised that this task is insanely complex, it was necessary take a few steps down the complexity ladder. Thus, the network is only trained on 6 different common bricks and any analysed models should only consist of these: 2x4 brick, 2x3 brick, 2x2 brick, 2x4 plate, 2x3 plate, 2x2 plate.
For training the model parameters, the *Adam Optimiser* was used as the optimiser and Sparse Categorical Crossentropy was chosen as the loss function. The model trains across two epochs with a batch size of 16. The script outputs the model structure and evaluation results in the terminal when executed. Furthermore, it saves a figure that tracks the development during training.

The *SA-Unbuilding-LEGO.py* is built up around the same core object-oriented structure and principles as the *A3-Edge-Detection.py* script. The object segmentation process (OpenCV (Bradski, 2000)) utilizes the changes in hue to detect edges. Firstly, the input image is transformed into HSV colour space. Then, it is split into the three respective channels and only hue channel is kept. Lastly, canny edge detection is performed along with a single dilation step before the contours around the bricks can be drawn. The biggest contour is removed from the list (always a contour that stretches around the entire model) and bounding rectangles are drawn around the individual bricks. These areas are then cropped out and stitched onto a black 400x400 pixel canvas to replicate the style of the images which the classifier has been trained on. One-by-one these images are pre-processed and fed to the classifier for prediction. The script outputs a final collage showing all the detected bricks side-by-side with the predictions made by the script.

Discussion of Results

Figure 7 shows the resulting brick segmentation when running the LEGO snake through the tool. This aspect works remarkably well and all of the bricks end up being outline by a tight bounding box. The edge detection steps enabled the fine-grained drawing of two discrete

thin lines on both sides of the dark edges that separate the individual bricks and this entails that each brick is fully surrounded by a distinct contour.

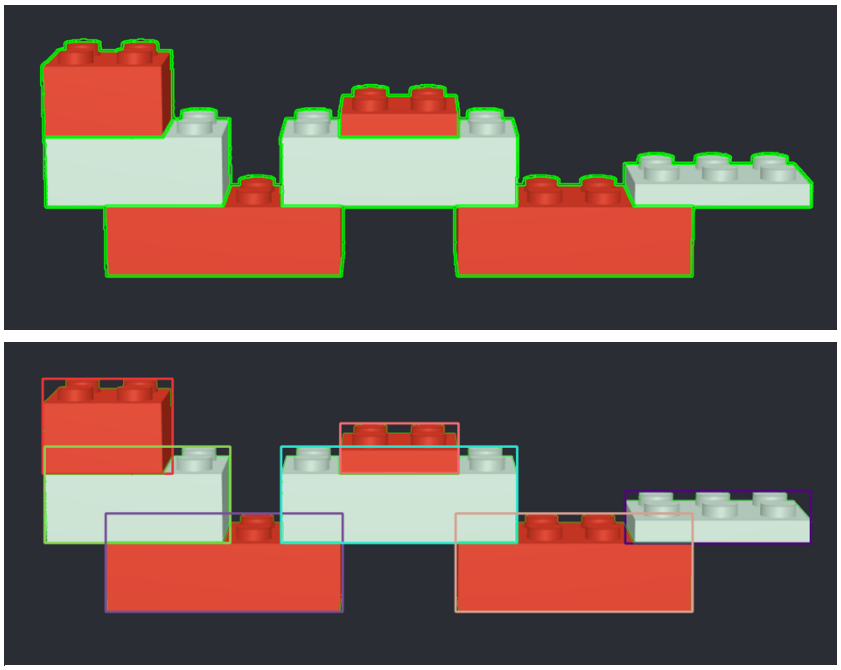

Figure 8 - Images of segmented LEGO snakes. Top: Contours have been drawn around the bricks. Bottom: Bounding boxes have been drawn around the contours

Let's redirect our attention towards the performance of the classifier. *Figure 9* shows the models performance metrics developments across multiple epochs. After two epochs, the validation accuracy has already stagnated while the train accuracy proceeds to increase, and the train loss drastically drops. This indicates that the model is starting to overfit, and it is best to halt the parameter search here. This may appear alarming at first but is important to keep in mind that the training data for each brick category solely consists of renderings of

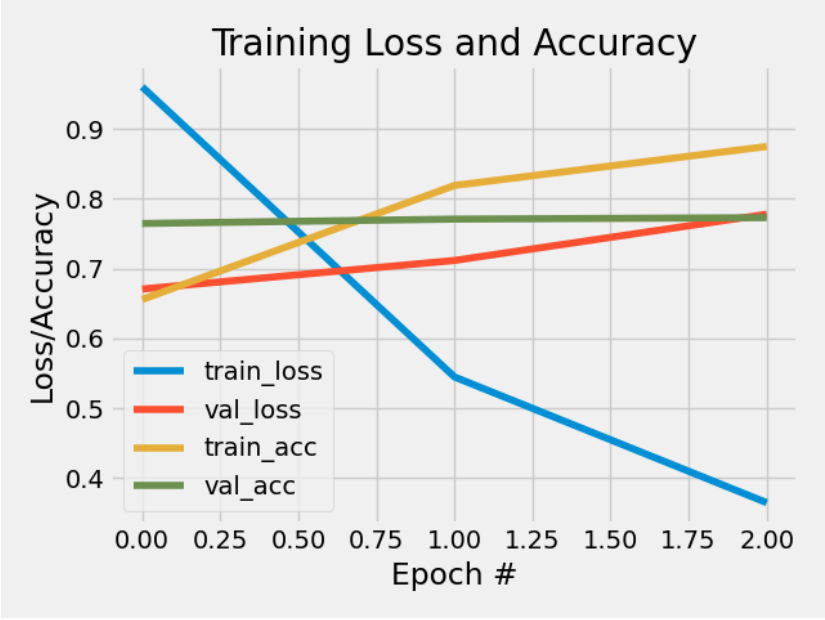

Figure 9 - Graph of the development in metrics during model training over 2 epochs

the same brick taken at different angles and therefore it is foreseeable the overfitting could quickly arise.

From *figure 9* it can also be observed that the *validation accuracy* starts off higher than the *training accuracy*. This may seem peculiar at first, but it most likely due to the dropout noise that is applied during model training but during inference.

The trained CNN model yields an overall accuracy of 77% on the test data which seems to be decent under the circumstances. Figure 10 shows how well the model has performed at correctly classifying the bricks identified in the LEGO snake. Here, it is apparent that only two (brick no. 1 and no. 3) out of seven bricks are predicted spot-on by the model. This clearly suggests that the tool is still highly faulty even when tested on the most simplistic LEGO models. Two out of seven is, nonetheless, still better than chance and it is also worth noting that all bricks are correctly identified as bricks. Akin to the problems faced when training a simple digit classifier on the MNIST, this LEGO classifier is trained on an extremely homogenous dataset, and it is therefore expected that it generalises very poorly when faced with unfamiliar data. A robust model would as a minimum need a much more comprehensive data set of labelled images of LEGO bricks in many different settings.

All in all, it has been a tremendously fun project and measured solely on the basis of my initial ambitions, the results are certainly satisfactory. The tool is very highly reliable on sharp changes in colour and hue and, hence, it would face great challenges if met with even a simple model with bricks of the same colour clustered together. Were the vision of an omniscient, magical scan-a-model-and-receive-a-perfect-manual that I eagerly conveyed in the *description* section to ever be realized, it would demand much more complex modelling, more advanced data, and better equipment.

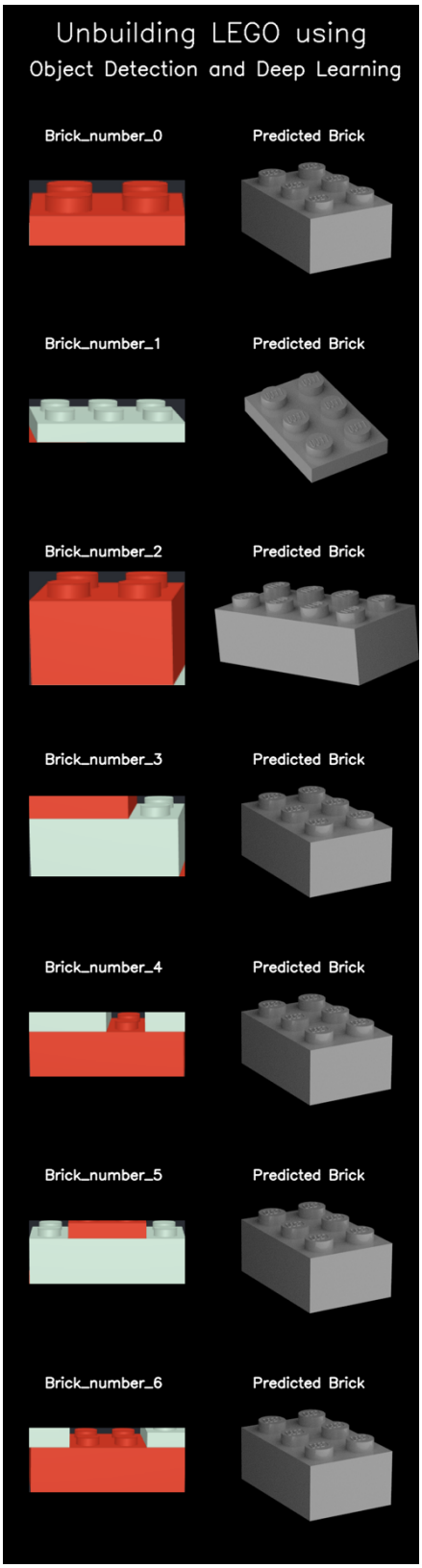

Figure 10 - This figure shows the individually extracted bricks from the snake and the corresponding brick predictions computed by the classifier. Only two bricks are classified spot-on, but all bricks are predicted to be bricks and both plates are predicted to be plates.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

Basha, S. S., Dubey, S. R., Pulabaigari, V., & Mukherjee, S. (2020). Impact of fully connected layers on performance of convolutional neural networks for image classification. Neurocomputing, 378, 112-119.

Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.

Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence, (6), 679-698.

Clark, A. (2015). Pillow (PIL Fork) Documentation. readthedocs. Retrieved from https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf

Davis, L. S. (1975). A survey of edge detection techniques. Computer graphics and image processing, 4(3), 248-270.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. IEEE Annals of the History of Computing, 9(03), 90-95.

LeCun, Y., Cortes, C., & Burges, C. J. (2010). MNIST handwritten digit database.

Lundh, F., 1999. An introduction to **tkinter**. URL: www. pythonware. com/library/**tkinter**/introduction/index.

McKinney, W., & others. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51–56).

Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel, O., … others. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830.

Waskom, M. L. (2021). Seaborn: statistical data visualization. Journal of Open Source Software, 6(60), 3021.

Wichmann, F. A., Sharpe, L. T., & Gegenfurtner, K. R. (2002). The contributions of color to recognition memory for natural scenes. Journal of Experimental Psychology: Learning, Memory, and Cognition, 28(3), 509.

Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. CreateSpace.