Building an image classifier from scratch

A dissertation submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF SCIENCE in Computer Science

in

The Queen's University of Belfast

by

Robert Gray

02/05/22

SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name:	Robert Gray	Student Number:	40230638
Project Title:	Building an image classifier from scratch		
Supervisor:	Dr Niall McLaughlin		

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance. **By submitting your dissertation you declare that you have completed the tutorial on plagiarism at http://www.qub.ac.uk/cite2write/introduction5.html and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.**

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

Student's signature Robert Gray *Date of submission* 02/05/22

[illegible]

Acknowledgements

I would like to thank everyone involved in the allocation of the final year projects. ‘Building an image classifier from scratch’ was my first choice and I am grateful to have been able to explore an area of computer science of which I had no prior experience. This project would not have been possible without the help and encouragement from Queen’s University. In particular I would like to express my gratitude to Dr. Niall McLaughlin for his continued support and suggestions throughout the design and development of this project. Dr. McLaughlin helped outline some of the initial requirements for the project and encouraged research into further areas of exploration throughout the year. Lastly, I would like to thank the coordinators of the two Machine Learning modules I took this year, Dr. Baharak Ahmaderaghi and Dr. Jesus Martinez del Rincon. These two modules acted as both an initial introduction to the area of computer vision and provided an insight into how neural networks can be developed without the use of existing libraries.

Abstract

Computer vision has evolved significantly since its first commercial application by ‘Kurweil Computer Products’ in OCR (optical character recognition) in 1974. Technologies such as autonomous vehicles and facial recognition has led to the increased need for robust feature extraction, image classification and object detection within a scene. This dissertation explores how a variety of feature extraction techniques can be used to train different machine learning models. The primary aims of this project are to understand how these approaches are implemented from scratch without the use of existing libraries and to evaluate how the results obtained compare with existing implementations.

Contents

1 Introduction and Problem Area

1.1 Introduction

The need for robust image classification is becoming increasingly prevalent in our daily lives from the relatively mundane, providing tagging metadata for smart phone photos [1] to the vital, real-time object recognition in autonomous vehicles and braking systems [2]. The rate of data creation is astounding. PwC predicted in 2018 that the ‘universe of data’ would grow to 44 zettabytes by 2020 with Statista projecting 180 created in 2025 alone. Much of this data will be comprised of images and videos. Good computer vision through the implementation of highly specialised feature extraction and machine learning models can automate image classification for vast datasets deemed otherwise daunting or impractical for manual classification. However, in most cases the development of these models relies on the previous efforts of researchers without the fundamental grasp of why these approaches work [3].

1.2 Problem Area

This project aims to research and implement a series of extraction and classification techniques from scratch to build a wider image classification program. The task will focus on existing feature extraction methods that perform well in the general context of image classification and those deemed particularly beneficial in conjunction with a specific dataset.

This paper will explore and aim to answer the following questions.

- What are the most effective feature extraction techniques that can be developed in this project’s timeframe for image classification?
- Which approach (classical or deep learning [4]) to computer vision is the most promising at the end of the project’s development?
- How do the developed solutions perform compared to existing implementations [5]?

As stated, the project will focus on the development of various classification models. In the interest of brevity, the term ‘model’ from here on will refer to a specific technique for classification i.e., KNN (K Nearest Neighbour) along with a specific feature used to train the model for example (RGB or grayscale image data).

1.3 Preliminary research into classification methods

Initial research into computer vision led to two distinct methodologies for classifier development, classical and deep learning.

The classical approach breaks the process of classification down into two steps. Firstly, a purely algorithmic approach to feature extraction. A predefined set of steps are used to

extract features from a sample, with hyper parameters used to adjust how features are extracted for all images. These pre-processed features are then fed into a model and evaluated. Test samples have the same pre-processing techniques applied before prediction. Typical techniques for extracting features included edge data, HOG, SIFT and colour histogram, whilst models consisted of KNN, Logistic Regression, Naïve Bayes and SVM. This approach also appeared more understandable and less challenging to implement as the pre-processing and model development where distinct steps. Moreover, each aspect could be clearly separated into its own stage and tested independently.
Comparing this to deep learning which adjusted how features are extracted during back propagation, which seemed much more likely to yield better classification metrics with the obvious trade-off being design, implementation and testing complexity.

1.4 Existing Solutions

Many development tools exist for image classification and can be used to implement a bespoke solution for a dataset. Some of the most popular libraries include, Scikit-image [6], OpenCV [7], Scikit-learn [8] and Keras [9].

1.4.1 Scikit-image

Scikit-image is an open-source image processing and manipulation Python library first released in 2009. It features numerous packages containing: feature extraction algorithms ranging from canny edge detection to SIFT, packages to adjust image properties and colour space conversion, along with a data package containing test images for experimentation. Scikit also features image IO for persisting extracted features. Scikit-image utilises Cython [10] along with native C code in its algorithm implementations. According to one article Cython can be up to 44 times faster than interpreted python [11]. Scikit has the advantages of a native compiled implementation and clear documentation whilst limiting a developer to a Python based workflow.

1.4.2 OpenCV

OpenCV contains much of the same functionality as Scikit-image and is focused on providing real-time computer vision. OpenCV is written exclusively in C++ and is thus highly optimized. It can also take advantage of GPU accelerated parallel processing through CUDA and OpenCL integrations. OpenCV's native backend can be called from almost any language. This is achieved using wrapper libraries such as 'opencv' for Java and 'opencv-python' for Python. Open CV presents developers with a steeper learning curve given its strongly typed nature, large feature set and highly parametrized functions compared to libraries like Scikit-image.

1.4.3 Scikit-learn

Scikit-learn is a Python library for data analysis and classification. Scikit features implementations for clustering models such as KNN and K means along with linear classifiers including SVM and stochastic gradient descent. Scikit-learn also features extensive metrics functions for model evaluation.

1.4.4 Keras

Keras again is a Python library for deep learning which uses TensorFlow for its implementation but abstracts much of the daunting aspects of neural networks into a high-level API. Keras also benefits from extensive documentation and examples [12] and many tutorials to rapidly develop models. If more granular control over a particular model is necessary Keras is also interoperable with TensorFlow.

1.5 Chosen areas of exploration

The scope for the project is limitless. However, the time isn't. To that end methods from both fields will be developed and evaluated. 'Classical' development will focus heavily on feature extraction techniques and how each compare. Whereas 'Deep learning' will focus on building the framework necessary to construct multi-layer perceptron models, along with a convolutional neural network where back propagation will determine how features are extracted. The aim is to have both methodologies developed to a stage where their performance can be directly compared.

In order to assess how each model compares and how they reflect existing implementations a large well-known dataset is needed. Whilst there are a vast array of suitable candidates such as ImageNet and Open Images Dataset [13] with single and multilabel samples respectively. CIFAR-10 [14] has been chosen for use in this project. CIFAR-10 is comparatively smaller, has one label per sample and is evenly balanced. The dataset consists of 60000 RGB images each 32x32 pixels in size. During development it became clear GPU integration would not be an option due to complexity. The size of each image lends itself to fast, iterative development as it's relatively computationally inexpensive to process. CIFAR-10 consists of labelled samples for the following 10 classes.

- airplane
- automobile
- bird
- cat
- deer
- dog

- frog
- horse
- ship
- truck

The dataset is broken down into 2 sections, train (50000 images) and test (10000 images).

Besides its standardised and precompiled nature, CIFAR-10 has also been heavily experimented with in the field of machine learning. Therefore, when the time comes to evaluate model implementations, comparable metrics are available [5].

After further research into CIFAR-10 a potential pitfall was uncovered that may merit further investigation. One of the golden rules of machine learning is to never train on test. This paper [15] discusses this as an issue with regards to CIFAR-10. The paper criticises the dataset showing that 3.3% of the images in the training set appear in the test set. They also show their models appear 10% less accurate when these duplicate images are removed. They propose using a modified version of CIFAR-10 called ciFAIR-10 where test images that appear in the training set are replaced with images of the same class. Given the scope of this project and the timeframe provided there wasn’t sufficient time to analyse their findings. However, it could be seen as a future area of exploration.

2 Solution Description and System Requirements

2.1 System Objectives

During early research it became clear that a project focusing on multiclass classification would be more challenging than its binary counterpart. It would open up areas of exploration such as One Vs All Classification [16] and more involved activation functions like ‘Softmax’ and its derivative. The project has the overall goal of constructing a range of models and assessing the performance of each. A user interface should facilitate feature extraction and allow an end user to vary the parameters of each model and feature extraction algorithm. The final system should also provide a saving functionality to persist extracted feature data and models to file. The system must also provide standard machine learning metrics to compare model performance.

These initial objectives led to these initial system requirements.

2.2 Initial Requirements

2.2.1 Image Pre-processing

1. Should read in CIFAR-10
2. Should focus on a range of techniques for classification.
3. Should range in complexity to ensure the final solution deliverables are fulfilled.
4. Each feature extraction method should focus on speed given the need for tweaking of algorithms and limited development time.
5. Should persist features across application launches.
6. The system should be extensible ensuring a developer can add new techniques.
7. Each extraction algorithm should be fault tolerant to overflow/underflow for all CIFAR-10 images.

2.2.2 Models

1. Should be highly configurable for a future developer / researcher.
2. Should connect easily to each feature extraction technique.
3. Should allow persisting of trained models to file.
4. Should enable importing of trained models.
5. Should be scalable to handle larger datasets CIFAR-100 [14]
6. Should provide independent metrics.
7. Each model should follow a similar API design and be interchangeable much like existing solutions.

8. In the case of model errors, for example ‘Nan values’ the model should halt and inform the developer / user.

2.2.2 Metrics
1. Should be useable across all models.
2. Provide the following metrics: Accuracy, Precision, Recall, F1 Score and ROC curves.
3. Provide and visualise a confusion matrix.
4. Work in both binary and multiclass contexts.

2.2.4 User Interface
1. Should allow users to extract features from CIFAR-10 set.
2. Should allow users to adjust parameters for each extraction algorithm.
3. Should provide visual examples of feature extraction results.
4. Should allow users to run each developed model with a particular feature.
5. Should allow the user to adjust each models’ settings.
6. Should provide interactive feedback when models are being trained through graphs and progress updates.
7. Should be clear and consistent across each implemented feature.

2.3 System Requirements
From the refined scope and initial requirements outlined above, a solution has been proposed. Each component’s requirements are broken down below.

Functional Requirements

2.3.1 Dataset
1. Should load the CIFAR-10 dataset once from file and cache once for use throughout the applications runtime.
2. Should convert CIFAR-10 integer format to normalised floating point type.
3. The loaded dataset should be easily sub-sampled for UI previews.
4. The dataset should be easily divided into train, test and validation samples.

2.3.2 Feature Extraction
Grayscale
1. Should return an image of the same shape as the input with one colour channel comprised of grayscale values from 0 – 255.

Colour Histogram

2. Should extract 3 tables each containing 256 intensity value counts containing the number of pixels in the channel of that intensity.
3. The sum of the extracted value counts should equal the number of pixels in the image.

Convolution

1. Should have a parameter for determining whether the original size of the image should be maintained.
2. If the size should be maintained the input image should be padded, different padding options should be available to a user or developer i.e., pad with zeros or reflected padding.
3. Should be fault tolerant to issues where the size of the mask > size of the image and pad appropriately, this should be provable in unit tests.
4. Ideally should be optimised for speed as it will be heavily relied upon during CNN training and testing.

Sobel Edge Extraction

1. Should extract grayscale edge data proven visually by the developer and researcher.
2. Aspects such as convolution and magnitude / gradient calculation should be proven in unit tests.
3. Should provide a thresholding value in the implementation (user determined) for eliminating weaker edge values.

Canny Edge Extraction

1. Should generate a Gaussian kernel no larger than the input image.
2. Kernel size should by determined be the user / developer via a provided value for sigma.
3. Kernel should be able to be visualised through a 3d plot.

Histogram of Oriented Gradients

1. Should generate HOG data across 9 discrete bins [17].
2. Provide a block size parameter to manipulate the size each image sub regions and thereby the HOG vector length. The block size must be smaller for smaller images.
3. Provide a visualisation facility to reverse the HOG vector back into its magnitude and direction components and display these as a composite image.

2.3.3 Classification Models

K Nearest Neighbour

1. Should provide a K value for classification of the mode class of neighbouring samples.
2. Should perfectly classify when K = 1 and train data = test data as all distances will be zero.
3. Implementation should be vectorized as distance calculations can't be precalculated and must be calculated at runtime.

Logistic Regression

1. Should provide a variety of weight initialization techniques.
2. Should provide a threshold value for determining a classification.
3. Should provide a learning rate parameter for tuning update impact.
4. Should utilise early stopping when loss value changes are minimal.
5. Collate metrics data for evaluation when training and testing.

Multi-Layer Perceptron

1. Each layer should have suite of easily interchangeable activation functions.
2. Layers must be highly configurable as the success of the model depends on parameter tuning.
3. Should provide a learning rate parameter for tuning update impact.
4. Collate metrics data for evaluation when training and testing.
5. Provide visualisation data for accuracy and loss over the training period.
6. Back propagation should be provable through manual gradient checking.

Convolutional Neural Network

1. The CNN layer should integrate with the existing MLP layer structure to aid development and avoid code duplication.
2. Feature maps extracted throughout training should be visualisable.
3. Convolution must be as fast as possible given the number of filters per layer per sample.

2.3.4 User Interface

1. Should expose the functionality defined above to the end user for each component.
2. Should provide visual feedback of extracted features using example CIFAR-10 images.
3. Should provide model feedback when training models through loss and accuracy curves ideally in real-time.

4. Should provide the ability to save and load each type of model for evaluation.

Non-functional Requirements

1. Clear docstring documentation provided for each component at class and method level.
2. Maintainable consistent code throughout the solution.
3. Follow industry standard coding style for example: PEP 8 for Python
4. UI should be consistent and follow the same layout for each model.
5. The solution (given the size of the dataset) must focus on performance through vectorized implementations, caching, persisted feature data and models.
6. Many feature extraction algorithms contain some functional overlap, common code should be shared amongst these algorithms and not duplicated.
7. Commonly used convolution masks should be stored in one location and referenced throughout the code base.

3 Design

3.1 System Architecture

Figure 3.1 shows a high-level architecture of the developed system. It represents the back-end system flow of CIFAR-10.

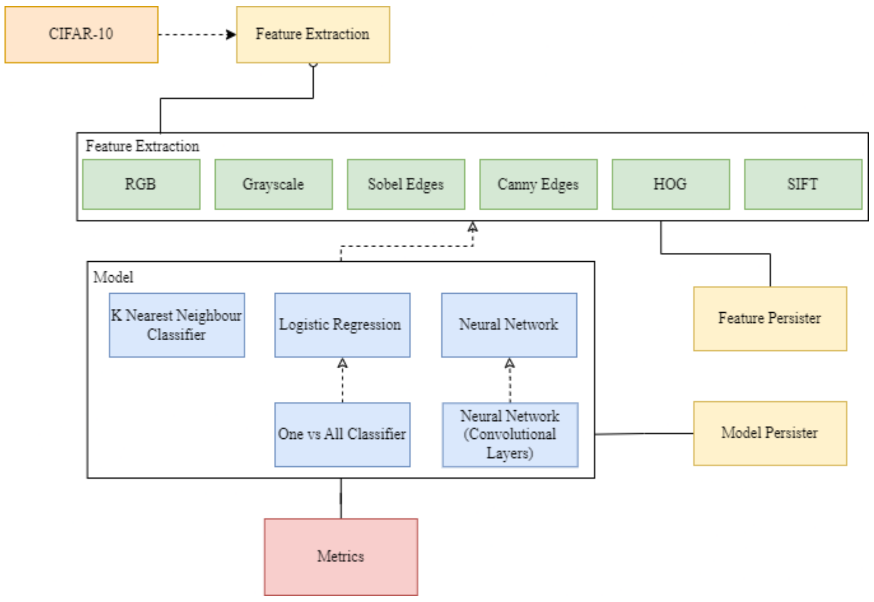


Figure 3.1: Architecture of the Classifier.

3.1.1 Accessing CIFAR-10

One of the most-early design considerations was how CIFAR-10 should be provided to the end user. Initially the most suitable method was thought to be, having the classifier download the dataset directly from the CIFAR-10 webpage [14], extract it and write to disk. It could then check for the presence of these files and avoid downloading them again on subsequent launches. However, the classifier may have models saved in the code repository dependent on the availability of the dataset. For example, a logistic regression model focusing on RGB as opposed to persisted HOG vectors. Checking for online availability, ensuring the dataset format never changes, handling model dependency issues and the requirement of an online connection for first setup all seemed like unnecessary steps. It was therefore decided that the dataset would be directly included in the Git repository. The loading of the dataset is handled by one class and once loaded from disk is cached by the program to avoid excessive memory usage.

There is also a utility class to allow developers to split the dataset into training, validation and testing components with a specified size. Figure 3.1.1 shows a UML diagram for the proposed classes.

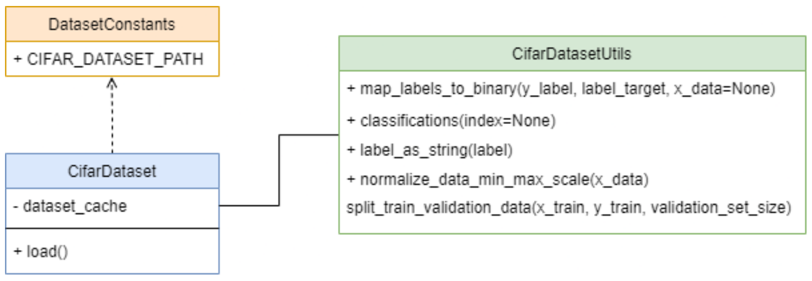

DatasetConstants
+ CIFAR_DATASET_PATH
CifarDataset
- dataset_cache
+ load()
CifarDatasetUtils
+ map_labels_to_binary(y_label, label_target, x_data=None)
+ classifications(index=None)
+ label_as_string(label)
+ normalize_data_min_max_scale(x_data)
split_train_validation_data(x_train, y_train, validation_set_size)

Figure 3.1.1: CIFAR-10 static dataset classes UML.

3.2 Feature Extraction Design

Another early consideration was to ensure each feature extraction technique was contained in its own package and that each had one public function called ‘extract’. This aided development as techniques could be swapped out with minimal changes to the code. Having a single extract method also helped design the feature persister class. The class’s ‘persist’ method takes an extractor instance containing the necessary extraction parameters. Extract can then be called during dataset iteration to extract features. Figure 3.2 demonstrates this functionality conceptually with a generic feature and the persister.

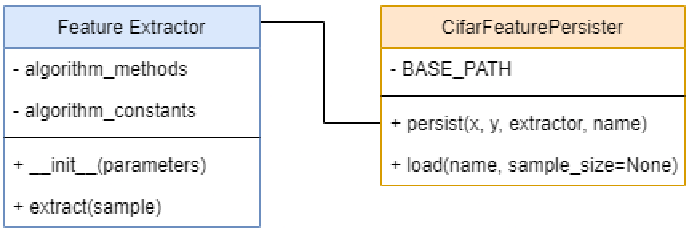


Figure 3.2: Feature extraction API and feature persister UML.

All feature extraction classes follow this concept. Each class is initialised with user defined parameters and 'extract' preforms the actual extraction, returning the feature vector for the provided sample.

Each extraction algorithm is outlined below and specific development details are outlined in the implementation section.

3.2.1 Grayscale

The first feature to be developed simply converted each CIFAR-10 sample from RGB to a grayscale image using the weighted method. The weighted method was chosen instead of the simpler averaging method as a more accurate representation of the original image. Figure 3.2.1 shows how the weights are applied across each pixel for each image channel.

$$gray(x,y) = 0.2989 * f(x,y,R) + 0.5870 * f(x,y,G) + 0.1140 * f(x,y,B) \quad (2)$$

Definition	:	
gray(x,y)	:	output image as grayscale image
f(x,y,R)	:	first color channel, red channel pixel value in specific (x,y) coordinates
f(x,y,G)	:	second color channel, green channel pixel value in specific (x,y) coordinates
f(x,y,B)	:	third color channel, blue channel pixel value in specific (x,y) coordinates

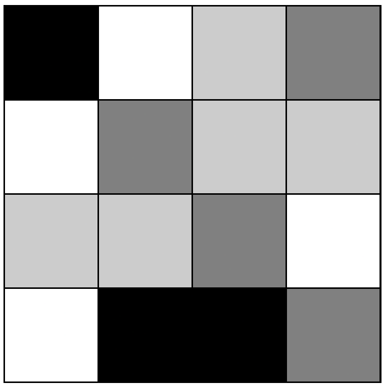
Figure 3.2.1: Grayscale function [18].

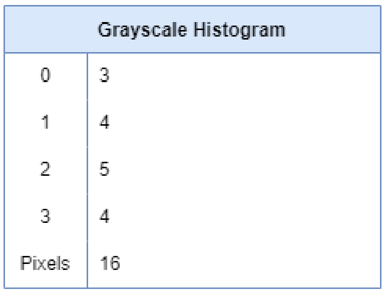
This algorithm was easily vectorized using NumPy by taking the dot product across all channels applying each weight to the corresponding channel for all pixels. The dot product summing the result to a single channel image of the same size.

In order to convert to grayscale, the extract method first asserts that the image is composed of 3 colour channels, if not no extraction is performed and the original image is returned along with printing a warning log message. Grayscale is necessary as a first step for all of the following extraction methods.

3.2.2 Colour Histogram

As stated, each CIFAR-10 image is made up of 3 colour channels, a colour histogram is a count of each colour intensity in a specific channel. Each colour component is stored in 1 byte of data, meaning each component can store 2^8 (256) different values in the range of 0 – 255. To create a colour histogram for an RGB image 3 tables are created. Each table is made up of 256 bins and each entry contains the number of pixels at that intensity level. Figure 3.2.2a gives an example for a single channel (grayscale image) with 4 intensity levels (2^2), the same principle generalises to each of the 3 RGB channels for 256 intensity levels.





Grayscale Histogram	
0	3
1	4
2	5
3	4
Pixels	16

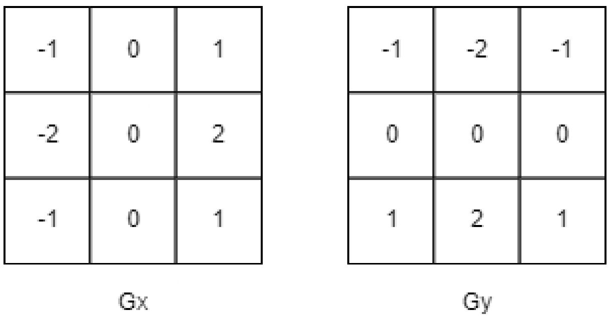
Figure 3.2.2a: Grayscale image and associated histogram table.

Proving the table generation for small images with low bits per pixel is trivial as pixel intensities are easy to distinguish and count. For CIFAR-10, the histogram counts were added and compared to the expected number pixels in the channel, 1024 (32x32). The extraction process returns 3 tables as mentioned. These can then be flattened to a one-dimensional vector before being used to train a model.

3.2.3 Sobel Edge Extraction

During development it was determined that Sobel based edge extraction [19] produced slightly more promising metrics than Prewitt for both KNN and Logistic Regression, this is also backed up by this article comparing different masks for edge filtering [20]. However, both approaches still exist in the application and both algorithms are identical apart from the masks used. The main advantage of the Sobel implementation over other edge extraction methods like Canny is it's time efficiency. This is due to the lack of non-maxima suppression and the reduced number of convolutions per image. A major disadvantage to this approach is the noise generated around each pixel during the extraction process. To help prevent this each sample is first convolved with a mean filter before extraction to smooth the image. Thresholding has been employed in an attempt to partially mitigate this by culling weaker edge pixels to zero below a certain value. However, given the size of each CIFAR-10 sample a lot of potentially meaningful data can be lost during this step.

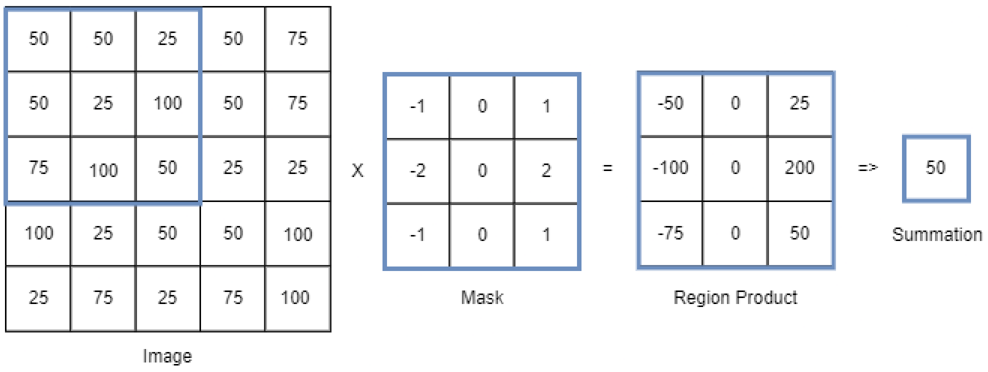
Sobel edge extraction uses the following two masks shown in Figure 3.2.3a for calculating the gradient of each pixel in both the X and Y direction.



$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3.2.3a: Sobel gradient masks.

Each image is convolved twice, once for each direction. As this is the first example of convolution in the system and given that it will be used extensively throughout the classifier an example is provided in Figure 3.2.3b. By default, the stride parameter is 1 though this will differ in other components in the system, thus the convolution function will take the stride as a parameter rather than using the default.



$$\underbrace{\begin{bmatrix} 50 & 50 & 25 & 50 & 75 \\ 50 & 25 & 100 & 50 & 75 \\ 75 & 100 & 50 & 25 & 25 \\ 100 & 25 & 50 & 50 & 100 \\ 25 & 75 & 25 & 75 & 100 \end{bmatrix}}_{\text{Image}} \times \underbrace{\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}}_{\text{Mask}} = \underbrace{\begin{bmatrix} -50 & 0 & 25 \\ -100 & 0 & 200 \\ -75 & 0 & 50 \end{bmatrix}}_{\text{Region Product}} \Rightarrow \underbrace{\boxed{50}}_{\text{Summation}}$$

Figure 3.2.3b: Example of image convolution using Sobel X mask.

Each sub region of the image (equal to the mask size) is multiplied by the mask, the summation of the resulting multiplication becomes the output for that region and is placed into the output matrix (a 2d matrix containing an entry for each sub region). The size of the output image is determined using the formula shown in Figure 3.2.3c and must be calculated for both the width and height of the output image [21].

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}:	number of input features
n_{out}:	number of output features
k:	convolution kernel size
p:	convolution padding size
s:	convolution stride size

Figure 3.2.3c: Convolution output size formula [21].

Both convolved images can be combined to create the edge data by taking the sum of the absolute values (magnitudes of the two images). Figure 3.2.3d describes the magnitude calculation along with a faster albeit less precise version. The resultant magnitude image can be flattened and treated like any other feature vector. In the case of the classifier the more precise method was used.

$$mag(G) = \sqrt{Gx^2 + Gy^2}$$
$$fastmag(G) \approx |Gx| + |Gy|$$
$$G: output\ edge\ image$$
$$Gx: convolution\ output\ sobel\ x$$
$$Gy: convolution\ output\ sobel\ y$$

Figure 3.2.3d: Magnitude formulae.

3.2.4 Canny Edge Extraction

Canny edge extraction was originally outlined in the 1980’s by John Canny [22]. It was decided early on that Canny should also be included in the classifier as a direct comparison to edge extraction. Canny has the advantage of producing thinner edge data with less noise by utilising non-maxima suppression. Non-maxima suppression is described in detail in this article [23] from OpenCV in the context of bounding box detection.

The first step of the Canny algorithm creates and convolves a gaussian kernel across the input image to reduce high frequency noise such as intense speckles or artifacts in the image which may lead to the extraction of false edges. A gaussian blur is used instead of faster alternatives such as mean filtering, as neighbouring pixels help influence the output [24]. The formula for calculating the value for each entry in the gaussian matrix is given in Figure 3.2.4a with sigma denoting the variation around the mean value. A larger sigma value will produce a wider gaussian distribution and a larger kernel will be needed to capture the increased variance. During development care was taken to limit the maximum value of sigma as the size of the input images were small (32x32) and could easily reach a point where the gaussian filter was larger than the entire image. Through experimentation it was found that values higher than sigma=3.5 caused convolution to either fail without padding or produce entirely black or white images, the UI has been designed to limit sigma to a maximum of 3.5.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Figure 3.2.4a: Gaussian formula for a 2d point [24].

The gaussian kernel creation function was designed to include a sigma parameter for tuning the kernel to improve edge extraction. The kernel can also be visualised to help prove that it has been correctly generated. Figure 3.2.4b shows a kernel generated within the application at sigma=1.5.

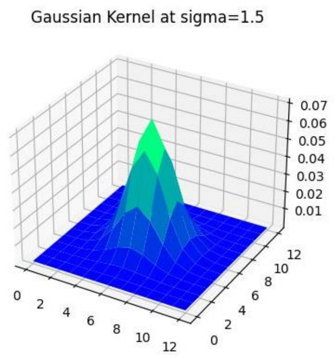
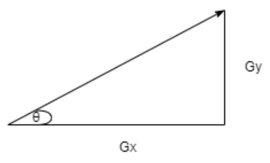


Figure 3.2.4b: Development plot of 2d gaussian kernel.

After smoothing the image, the next stages of the canny algorithm mimic the regular process of Sobel edge extraction and the system was designed to reuse aspects such as convolution and mask matrices without duplication.

The first stage involves applying Sobel edge extraction to the gaussian blurred image to calculate the gradients in each direction as before. The magnitude is calculated and the two convolved gradient images are also used to calculate the directional component using basic trigonometry outlined in Figure 3.2.4c. Theta can be calculated using inverse tan.



$$\theta(x,y) = \arctan\left(\frac{Gy}{Gx}\right)$$

Figure 3.2.4c: Image gradients and how they relate to theta.

The next stage refines the edge data. Sobel convolution does not take neighbouring pixels into account leading to many directly adjacent edge detections. In order to thin out these edges non-maxima suppression is applied to the magnitude data [27] based on the angle of the resultant vector and if the pixel is a local maximum compared to its neighbours along that angle. To achieve this, as each pixel has 8 neighbours angle values are grouped into 45-

degree bins. If a pixel is smaller than its neighbour it is suppressed to 0 thus eliminating a weak edge detection. Again, this is applied to every pixel in the magnitude matrix and care should be taken to avoid running over the width or height of the image triggering an out of bounds exception. The loop should stop at the penultimate pixel in both x and y to ensure it has a neighbour to compare with.

The next stage involves applying a double threshold to the resultant image created during non-maxima suppression. Thresholding aims to remove irrelevant edge data by applying a high threshold to detect strong edges greater than this threshold, along with a low threshold to detect weak edges greater than or equal to this threshold. If a pixel lies below the low threshold it is suppressed, edge magnitude values lower than the weak threshold are considered irrelevant as they are either noise or without neighbours [28]. Figure 3.2.4d helps illustrate thresholding and is taken from the development of the Canny algorithm within the classifier.

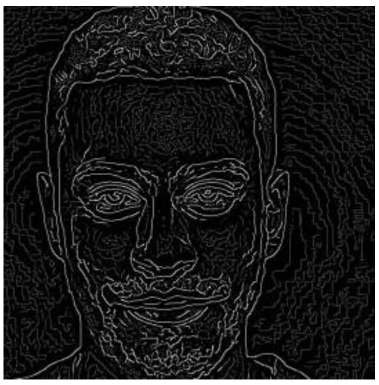


Figure 3.2.4d: Development image illustrating double thresholding.

Thresholds are not definitive as each sample presents different lighting and image artifacts. Thresholds should be specific to the dataset and determined through experimentation. The design allows for this through careful consideration of the default thresholding values and the inclusion of parametrized thresholds in the extraction class constructor.

The final stage aims to unify these 2 edge types. All strong edges can be included in the final feature as a first step. Any weak pixels attached directly to a strong pixel are in effect made strong and are included as well. Noise is unlikely to be detected as a strong edge and it is therefore likely that a weak detection neighbouring a strong detection is part of that same edge. Disassociated weak pixels are likely due to small features, colour variation or random noise and are thus suppressed. The final image is flattened to become the feature descriptor.

3.2.5 Histogram of Oriented Gradients

The next extraction feature to be developed was HOG. HOG gained popularity in 2005 in a paper written by two French researchers [17], despite the concept having been originally patented in 1986. The HOG algorithm consists of extracting an overall representation of an image by breaking the image up into localized cells [29] known as blocks. In each block the magnitudes and direction (shape) of each pixel is observed. These shapes are 'binned' together based on their angle to form a histogram for that region. The process is repeated for each block of the image. Extracting multiple descriptors per image aids the identification of similar regions between training and test data. HOG is partially invariant to geometric changes [17] and has been used extensively in sub image object detection [30]. HOG is not invariant to rotation and scale. However, in most CIFAR-10 images the subjects are positioned similarly and are rarely rotated. HOG is also time consuming to calculate and scales linearly with the image size. Existing CIFAR-10 classifiers gain upwards of a 10% benefit in accuracy when using HOG vectors rather than RGB. The HOG vector is also considerably smaller (dependent on a block size > 1) and thus reduces model training time.

The HOG algorithm is reasonably static in terms of parameters requiring adjustment. The one exception to this is the block size used to break up the image into smaller sub regions. The seminal paper [17] encourages experimentation with this value (8x8 blocks appearing as the consensus elsewhere) [29]. Using an 8x8 block size in a CIFAR-10 classifier would not be advisable due to the small image size resulting in incredibly small HOG vectors with little descriptive information. Through experimentation, block sizes smaller than 4x4 provided better classification metrics. The design decision was made to include a block size parameter in the extraction class and UI.

The first step of HOG is to calculate the gradients for each pixel. In this case Perwitt masks are used instead of Sobel or Gaussian. The paper [17] focused on human detection and discusses using more advanced masks. However, these resulted in poorer metrics over Perwitt as the image became too smooth. Gaussian masks also resulted in slightly worse performance despite the emphasis placed on stronger pixels. As CIFAR-10 contains both inorganic and organic classes it was decided to adopt their findings. As with canny the resulting X and Y convolutions are used to calculate the direction for each pixel. These steps could be completed on a per-block basis. However, they can be applied to the image as a first step to avoid excessive calculation during each iteration. The angles are also shifted to be in the region of 0-180 degrees rather than 0-360 degrees.

Once the magnitudes and angles have been calculated for each pixel, each block region is processed. One histogram (a vector with 9 elements) is generated for each block. Figure 3.2.5a shows an example of a HOG vector represented as a histogram. During the iteration of each block, for each pixel in the block both the angle and magnitude are noted. Each magnitude value is added to the vector at the position corresponding to the angle of the pixel. The histogram block is also normalised after all entries have been placed in the histogram.

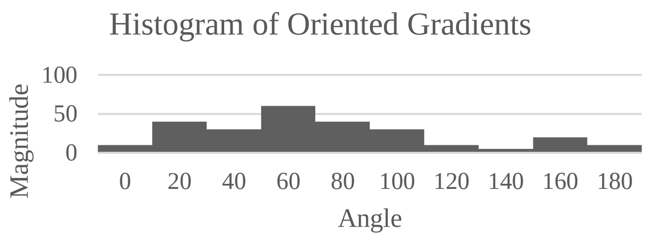


Figure 3.2.5a: Example histogram of gradients for a sub region.

The final HOG vector is created by concatenating each of the block histograms together and flattening.

During development it also seemed essential to visualize the HOG vector as every other feature extraction technique in the classifier has this functionality. To that end each HOG vector can be reversed to extract its magnitude and angle histogram vectors. From that a series of images can be created to represent each block of the image. It was decided that the least computationally intensive way to form such an image would be to create a series of small images for each histogram angle. These could then be read in once, copied and manipulated to adjust their intensity relative to the histogram magnitude at that angle. Figure 3.2.5b shows the images generated using Photoshop rather than at runtime.

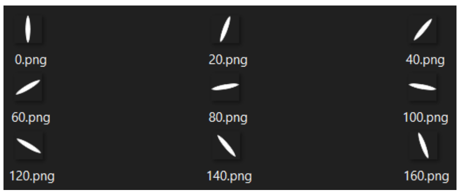


Figure 3.2.5b: Classifier HOG angle images.

Figure 3.2.5c shows how these images are used to visualize a HOG vector.

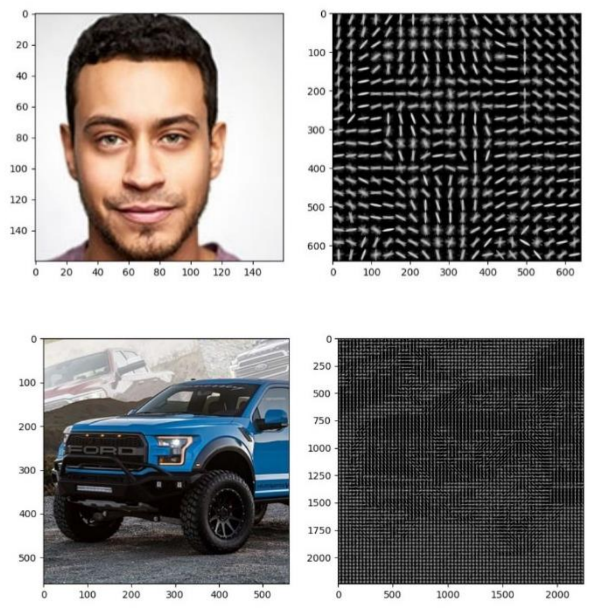


Figure 3.2.5c: Original image and HOG vector representation.

3.2.6 Scale Invariant Feature Transform

The last notable feature extraction technique to be explored was SIFT. SIFT seemed like a natural follow on from HOG as it employs similar binning techniques whilst also exhibiting scale and rotational invariance. Development and time constraints has led to only a partial implementation of the SIFT algorithm. As of this point SIFT key points can be extracted from images and point clusters appear consistent across rotated images as shown in Figure 3.2.6a. However, key point matching is not working. SIFT was implemented following Lowe's paper [31].

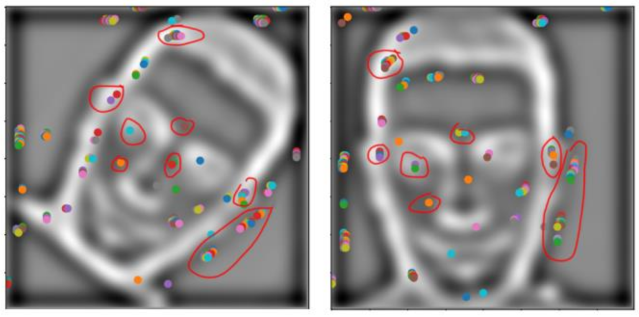


Figure 3.2.6a: SIFT key points rotation invariant (Gaussian level 2)

The first step of SIFT involves creating increasing blurs for a series of downsized images derived from the input. The reduced sizes develop the scale invariance whilst the increasing blur intensities filter noise to define more distant key points [32]. Lowe's paper suggests using 4 octaves and 5 blurs per octave with an initial sigma value of 1.6 increasing by a factor of $k = \sqrt{2}$ for each blur in the octave. Figure 3.2.6b shows each octave and its gaussian blurs forming a scale-space.

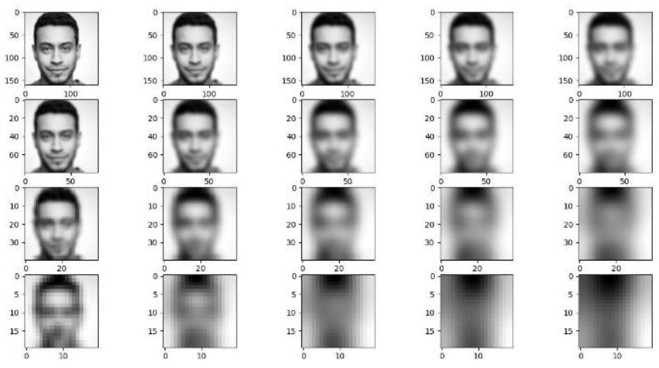


Figure 3.2.6b: Scale-space 4 octaves 5 blurs per octave.

The next stage takes each octave in turn. Each blur is subtracted from the next, 1 from 2, 2 from 3 and so on to form 4 Difference of Gaussian images in each octave. Figure 3.2.6c shows the corresponding DoG images for the space scale above. Lowe's paper suggests using LoG (Laplacian of Gaussian) which is the second order derivative used to locate edge and corner data as an initial step to find areas of interest which may become part of the final key point set. However, this step is costly due to the large number of convolutions required. Lowe proposes subtracting the Gaussian blurs as an approximation of the LoG function [31] it is also described in this paper [34] in the context of Gaussian pyramid generation.

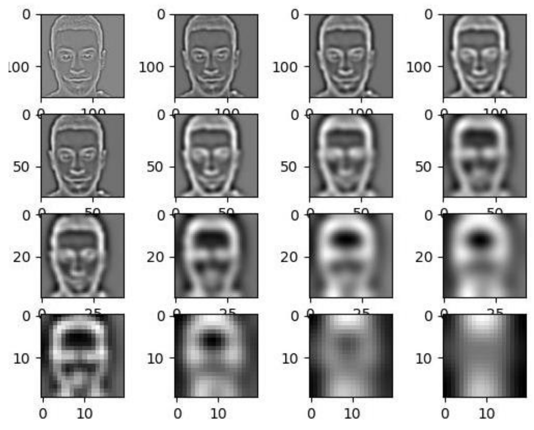


Figure 3.2.6c: Difference of Gaussian images.

In order to refine the initial key points, each one is compared to its 8 neighbours in the current scale and 9 neighbours in the above and below scale as shown in Figure 3.2.6d. A key point is denoted as a local extremum if it is greater than all neighbouring pixels or less than all neighbouring pixels.

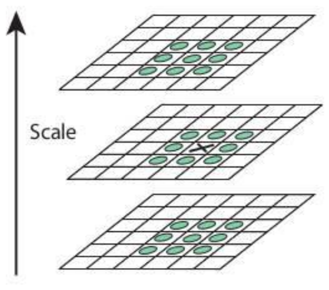


Figure 3.2.6c: Local Extrema Detection [31]

The candidate key points have now been refined to those which only fall on an extreme pixel. However, at this stage each key point location is accurate to the nearest pixel. A key points true location may in reality fall between pixels. A more accurate value is found by substituting the key point into the quadratic Taylor expansion function's derivative [31] with respect to x when x prime is 0. If the difference in any direction > 0.5 the extremum lies on a different point [31]. Eliminating key points based solely on contrast and location is not sufficient as there is an abundance of key points along edges, Figure 3.2.6d shows an early development image demonstrating this.

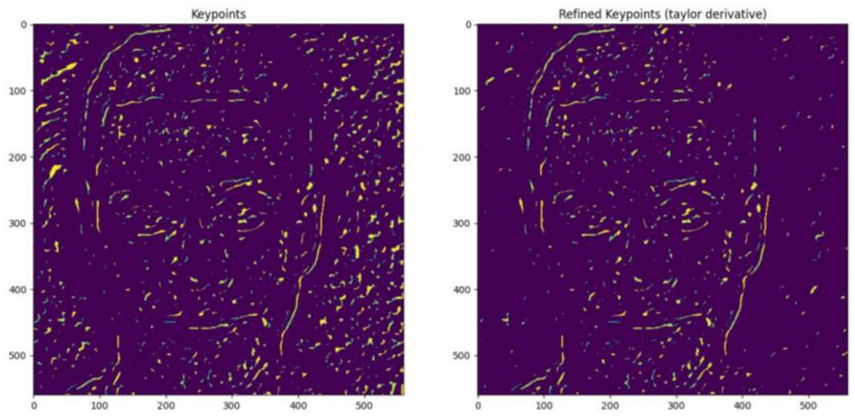


Figure 3.2.6d: Key points before and after Taylor expansion and contrast refinement.

To remove key points along edges whilst maintaining those which help define the shape techniques similar to those in corner detection are employed. First the second order derivative of the pixel at the key point is taken using the Laplacian derivative mask [32]. The second order derivatives are then used to compose a Hessian matrix [35]. Once computed along with the eigen values of the Hessian matrix. The Hessian matrix is multiplied with the extrema. Each feature point is evaluated using the ratio of eigen matrix (principle of curvature). If the ratio > 10 and the contrast < 0.03 the key point can be confidently rejected.

A vector of descriptors can now be generated from the key point data. The descriptor format is similar to that of HOG however each descriptor now ranges from 0-360 degrees. In this paper's implementation a block size of 3x3 was used. When generating SIFT histograms new key points may be created, if a histogram contains a peak over 80% a new key point is added to the list of key points creating a cluster around particular points of interest. Lowe's paper proposes matching descriptors using a nearest neighbour-based algorithm. This papers implementation is limited as further development time would be necessary to achieve key point matching.

3.3 Model Design

The aim of each model's design was to ensure API uniformity to aid UI design and integration with the model persister, much like the feature extraction API. Each model class's public interface drew inspiration from the Keras library and contains: fit, score and predict methods. Figure 3.3a shows a UML diagram of a model class using the logistic regression model as an example.

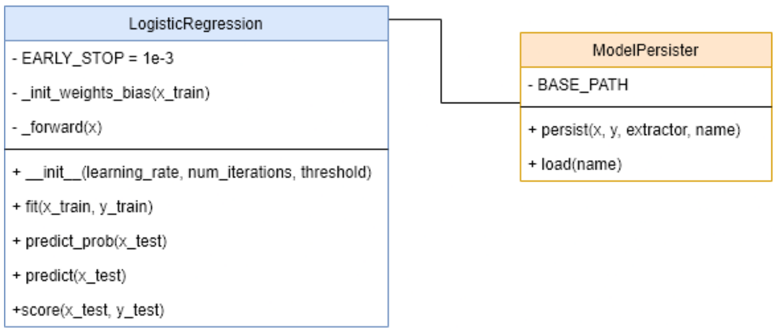
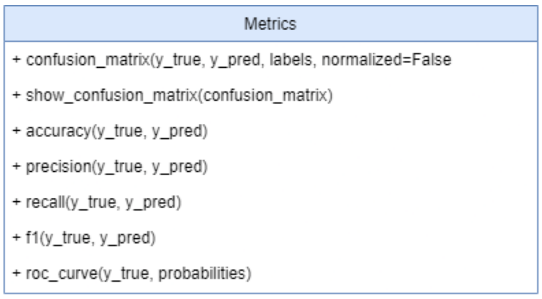


Figure 3.3a: Model UML for logistic regression.

Each model follows this design. The class constructor will setup parameters necessary for training, in this case learning rate and number of training iterations. The fit method will initialise weights and bias matrices and train the model taking in the training feature vector and associated labels. The logistic regression model also features a 'predict_prob' method this is necessary as the probability is directly compared by the One vs All classifier. The last method present in each model class is 'score', this will calculate useful metrics returned as a dictionary when evaluating a model. As metrics exist across all models a static metrics class was also designed. Figure 3.3b describes the proposed metrics class.



Metrics

+ confusion_matrix(y_true, y_pred, labels, normalized=False
+ show_confusion_matrix(confusion_matrix)
+ accuracy(y_true, y_pred)
+ precision(y_true, y_pred)
+ recall(y_true, y_pred)
+ f1(y_true, y_pred)
+ roc_curve(y_true, probabilities)

Figure 3.3b: Metrics class.

Each metric will be tested separately and developed with the flexibility to generate metrics for both binary and multiclass classification

The convolution neural network model features a more complex multi-layer architecture. This will be outlined in detail later in this paper. For now, the classical models will be discussed.

3.3.1 K Nearest Neighbour

KNN is a supervised classification method that classifies a sample based on the number of samples closest to it in N dimensional space. The primary design consideration was the development of a high-speed vectorized function for calculating Euclidean distance between a sample and all training samples. KNN distance calculations cannot be trained and therefore must be computed at runtime, distances can be computed somewhat quickly using NumPy. However, using nested Python for loops to iterate across each of the samples severely limits the rate of calculation. The only functional parameter necessary for changing how the model performs is the K value. K refers to the number of neighbours to match against and was exposed to the user in the UI. A set of precomputed feature vectors are also provided via a drop-down menu. Limitations to the UI inputs where also added to ensure calculation times were not excessive. Training and testing samples sizes were limited to 5000 samples. Figure 3.3.1a shows the Euclidean distance formula for two n-dimensional vectors.

$$euclidean(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Figure 3.3.1a: Euclidean distance.

The resulting scalar values are sorted and the closest (nearest neighbour) is deemed as the classification. When using a value of K > 1 the modal class of the K closest distances is used as the classification. Figure 3.3.1b shows an example a 3 class KNN model with a test sample in grey, in this case K = 3 and the modal class of the 3 nearest neighbours is Class 2.

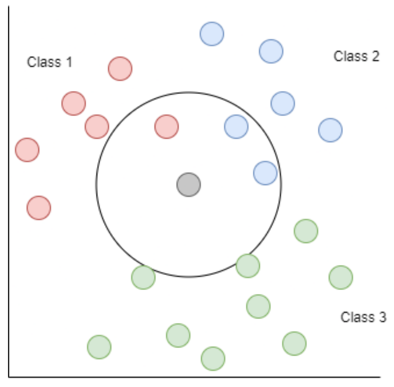


Figure 3.3.1b: Multiclass KNN K = 3.

KNN distance calculations can be proven by taking the same data for training as testing and computing the distances, for each sample there should exist a point in n-dimensional space with a distance of 0 meaning the model should achieve 100% accuracy when K = 1 and an accuracy less than 100% for K > 1.

3.3.2 Logistic Regression

Logistic regression was the second model to be developed. Logistic regression creates a probability distribution for a particular problem usually in a binary classification context. The goal being that this could be used within a One vs All model extending to the multiclass problem of CIFAR-10. Developing a working logistic regression algorithm was also seen as a first step towards a true multiclass perceptron using softmax. Again, the class follows the public interface applicable to each classification model. The constructor was designed to include a learning rate parameter along with an iteration count and a defaulted positive classification threshold value of 0.5. Weight initialization is critical to model convergence [36] [37]. An optional weight type parameter has been included and can be specified to choose different weight initialization techniques. The model supports, random uniform and Xavier initialization. Figure 3.3.2a provides the formula to calculate the Xavier weight limit for a given weight matrix. The weights are initialized using a uniform distribution between the negation of the limit and the limit.

$$limit(in_{units}, out_{units}) = \sqrt{\frac{6}{in_{units} + out_{units}}}$$

Figure 3.3.2a: Xavier uniform weight limit [37].

Logistic regression is similar to linear regression in that a line of best fit is drawn between the class datapoints, logistic regression creates a probability distribution by utilising the logistic function [38] mapping all linear outputs along a sigmoid curve between 0 and 1 making it directly applicable to binary classification. Figure 3.3.2b shows the sigmoid function.

$$sigmoid = \frac{1}{1 + e^{-x}}$$

Figure 3.3.2b: Sigmoid function.

To create a probability distribution that can be trained to accurately train a dataset, three further components are needed namely, a weight matrix a bias for each weight and a loss function to measure training performance [39].

Each input element of the input vector will have and associated weight. Along with one bias value to shift the intercept. Figure 3.3.2c demonstrates a forward pass of an input vector through the linear multiplication and sigmoid activation stages of the perceptron. If an output probability is greater than the positive threshold (defaulted to 0.5) the sample is classified as 1 otherwise 0.

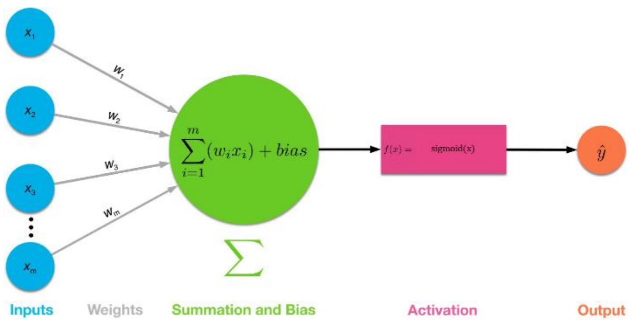


Figure 3.3.2c Sigmoid perceptron [40].

Unlike linear regression the mean squared loss cannot be used in logistic regression as the logistic function is nonlinear and MSE for the logistic function is non-convex, producing many local minimums instead of a true global minimum. Instead, the cross-entropy loss function is used the formula is given in Figure 3.3.2d where N is the number of samples, y is the true value and $\hat{y}$ is the model output probability.

$$Cross\ Entropy\ Loss = -\frac{1}{N}\sum_{i=1}^{N} (y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i)$$

Figure 3.3.2d: Cross entropy loss function

The backward pass is more involved and focuses adjusting the weights and bias to minimize the error between the probability $\hat{y}$ and the true class y.

The first step of back propagation calculates the partial derivative of the error w.r.t the output by subtracting of the true value from the output probability. Next, the partial derivative of the loss function w.r.t the weights and the partial derivative of the loss function w.r.t the bias are calculated using the chain rule [42] and the result is used to update the weight matrix W and bias b outlined in Figure 3.3.2e. The learning rate lr determines the impact of the update on each iteration.

$$Z = W^T X + b$$

$$\hat{y} = sigmoid(Z)$$

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} * \frac{d\hat{y}}{dZ} * \frac{dZ}{dW}$$

$$\frac{dL}{db} = \frac{dL}{d\hat{y}} * \frac{d\hat{y}}{dZ} * \frac{dZ}{db}$$

$$W = W - \left(lr * \frac{dL}{dW}\right)$$

$$b = b - \left(lr * \frac{dL}{db}\right)$$

Figure 3.3.2e: Back propagation using the chain rule.

The model repeats this for the number of iterations specified and the early stopping constant may end training when the change in the averaged loss across all samples is minimal.

3.3.3 One vs All Logistic Regression

The One vs All classifier is merely an extension of the logistic regression classifier. The One Vs All classifier contains ten trained models. Each model has been trained by mapping each label to either 0 or 1. For example, for the first class of CIFAR-10, ‘airplane’ each airplane entry in the dataset is mapped to 1 and the rest are mapped to 0. When predicting a sample, a test image is tested against each classifier in turn, classification is determined by the model with the highest probability.

3.3.4 Multi-layer Neural Network and CNN

The final model and arguably most complex extends the functionality of the logistic regression model into a multi-layer perceptron model. The model is designed to allow convolutional layers but can be used to create a succession of fully connected layers without convolution.

The main entry point to the neural network is the Sequential Model class. The architecture of which is outlined in Figure 3.3.4. Each neural network model is composed of a succession of layers, the output from one layer feeding directly into the input of the next. The sequential model holds a linked list of layers which is traversed during the forward pass. During back propagation the list is traversed in reverse order. Each layer inherits from a base Layer class which includes the functionality to resolve activation functions based on the string provided. Each layer also contains a 'compile' method which is invoked by the Sequential model before training and helps resolve the input and output shapes for each layer, shapes can be resolved provided an input shape is supplied to the first layer's constructor. If no shape is supplied an exception is thrown. However, this is rarely an issue as when a model is created the feature vector shape is supplied as the first layer's input shape.

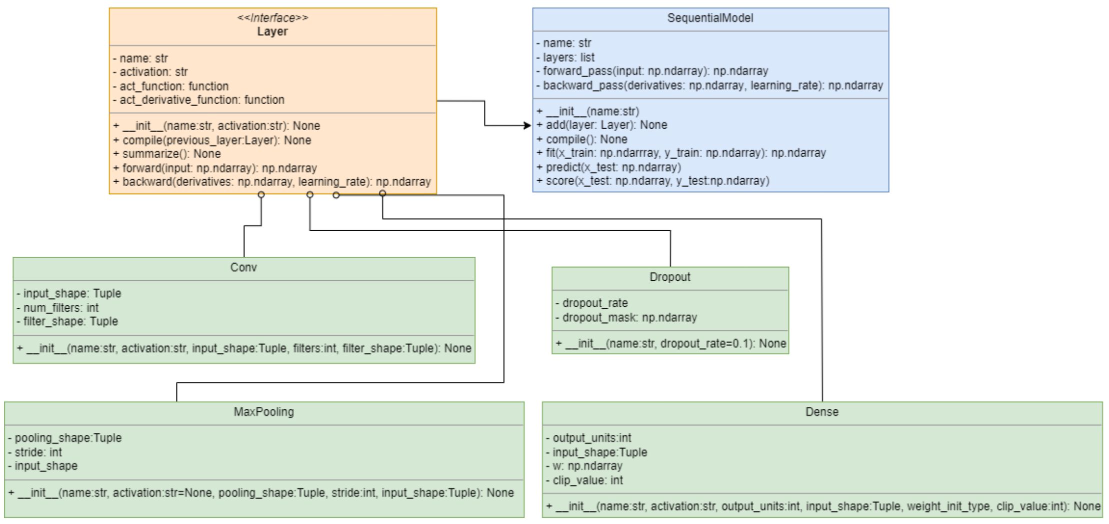

<<Interface>>
Layer
- name: str
- activation: str
- act_function: function
- act_derivative_function: function
+ __init__(name:str, activation:str): None
+ compile(previous_layer:Layer): None
+ summarize(): None
+ forward(input: np.ndarray): np.ndarray
+ backward(derivatives: np.ndarray, learning_rate): np.ndarray
SequentialModel
- name: str
- layers: list
- forward_pass(input: np.ndarray): np.ndarray
- backward_pass(derivatives: np.ndarray, learning_rate): np.ndarray
+ __init__(name:str)
+ add(layer: Layer): None
+ compile(): None
+ fit(x_train: np.ndarrray, y_train: np.ndarray): np.ndarray
+ predict(x_test: np.ndarray)
+ score(x_test: np.ndarray, y_test:np.ndarray)
Conv
- input_shape: Tuple
- num_filters: int
- filter_shape: Tuple
+ __init__(name:str, activation:str, input_shape:Tuple, filters:int, filter_shape:Tuple): None
Dropout
- dropout_rate
- dropout_mask: np.ndarray
+ __init__(name:str, dropout_rate=0.1): None
MaxPooling
- pooling_shape:Tuple
- stride: int
- input_shape
+ __init__(name:str, activation:str=None, pooling_shape:Tuple, stride:int, input_shape:Tuple): None
Dense
- output_units:int
- input_shape:Tuple
- w: np.ndarray
- clip_value: int
+ __init__(name:str, activation:str, output_units:int, input_shape:Tuple, weight_init_type, clip_value:int): None

Figure 3.3.4a: Neural network model and inherited layers UML.

The sequential model can also leverage the existing persisting framework for saving trained models to disk.

Activation functions

The system has been designed to support: ReLU, Sigmoid and Softmax activation functions [44] and their derivative. To ensure each function is interchangeable within each layer (excluding Dropout and Max Pooling as they do not have an activation function) each has been isolated into a method that is resolved dynamically by each layer's constructor. The activation function is called during a layer's forward pass and the corresponding derivative function is called during the back pass.

ReLU

The rectified linear activation function thresholds inputs <=0 to zero and is very inexpensive to calculate. It maintains linearity and enhances convergence [43] more so than sigmoid [44]. It's possibly the most widely used activation function and is seen as the default activation function of convolutional layers [45]. Figure 3.3.4.b outlines the ReLU function and its graph.

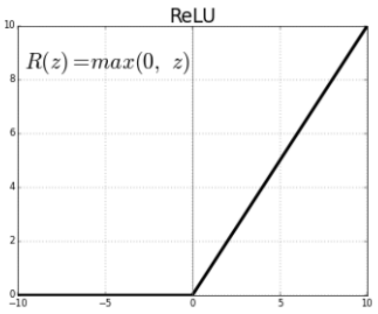


Figure 3.3.4b: ReLU [45]

The derivative of ReLU is simple as output < 0 will have a gradient of 0. Outputs > 0 always have a derivative of 1. The ReLU derivative is undefined at 0 and for the purpose of development it's taken to be 0 at 0.

Sigmoid

Sigmoid follows directly from the logistic regression model and maps inputs to a value between 0 and 1. The derivative of Sigmoid is necessary here as partial derivatives have to be calculated w.r.t a layer's input for multiple layers without access to the layers error. The derivative of the sigmoid function is outlined in Figure 3.3.4c where sigmoid is given as σ. The derivation of the sigmoid derivative is provided in this paper and article, [46], [47].

$$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$$

Figure 3.3.4c: Derivative of the sigmoid function [47].

Softmax

The final supported activation function is Softmax which maps an array of inputs to an array of values which sum to 1. Softmax was included to aid multiclass classification where output from the softmax function can be treated as a probability distribution relating directly to a model's classification confidence. The softmax formula is outlined in Figure 3.3.4d with an example of how the softmax function maps a layer's outputs to a probability distribution. The softmax function can be tested by asserting that the sum is 1.

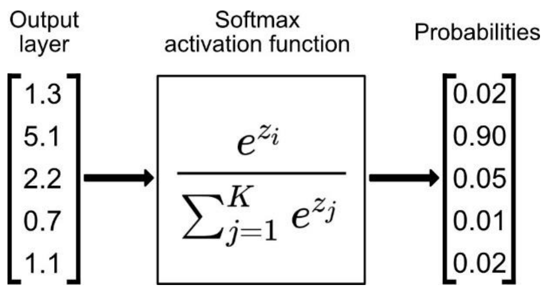

Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

Figure 3.3.4d: Softmax activation function. [48]

As softmax depends on raising e to the power of each input, large inputs inherently lead to overflow and instability. To mitigate these issues the inputs to softmax are often shifted by the maximum value in the input vector [49].

Dense Layer

The first aspect of the neural network to be developed was the fully connected dense layer. The dense layer architecture follows the single layer perceptron outlined in the logistic regression model. The only changes being the dynamic activation function resolved by the string passed to the Dense constructor and the number of output units can now be greater than one. During dense back propagation the error w.r.t the loss or the partial derivatives calculated in the previous layer are passed to the Dense layer and enable the calculation of that layer's derivatives w.r.t its weight matrix and biases. Figure 3.3.4e provides a developmental dense model featuring 3 ReLU layers and a final softmax layer training on a small sample of CIFAR-10.

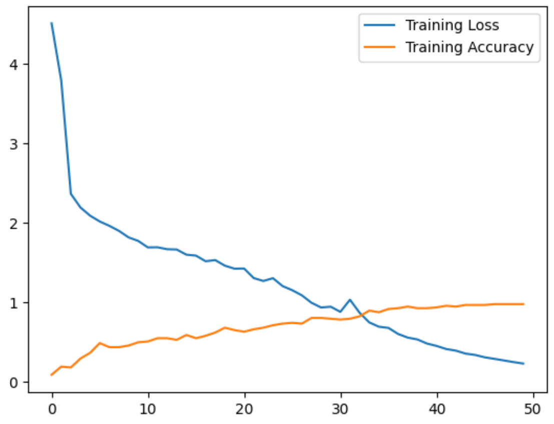


Figure 3.3.4e: CIFAR-10 dense MLP (100 samples).

Exploding Gradients

When developing the Dense layer and testing multiple layers (when using ReLU activated layers) models quickly reached a state of numeric instability resulting in Nan values. To combat this, a clip value was added to the dense layer (defaulted to 1) to threshold gradients > clip value preventing large weight updates and numeric overflow [50].

Dropout Layer
The second layer to be developed was the Dropout layer to mitigate overfitting by randomly setting inputs to 0 [51] temporarily reducing the amount of network connections hopefully aiding generalisation. A dropout rate parameter has been included and is set by the user between 0 and 1 as ratio of the number of connections which should be broken randomly on each forward pass.

Convolution
The final layers developed where convolution and max pooling. The convolutional layer creates a number of randomly initialized filters using the weight initialization methods outlined previously (Xavier and randomly between -1.0 and 1.0) each one of the same dimensions. Each filter is convolved across an input (or matrix of 2D inputs) creating a matrix of down sampled feature maps. The shape of the output feature map can be computed by using the convolution formula stated previously, multiplied by the number of filters. In the case of a CIFAR-10 sample in a layer where $n_{filters} = 32$ and each filter is 3x3 in size.

$$output_w = 28$$
$$output_h = 28$$
$$output_{shape} = (32, 28, 28)$$

An activation function is then applied to each of the elements in the feature map. This is incredibly time consuming using the current convolution implementation so a vectorized approach was developed and is outlined in the implementation section of this paper. During the back pass of the convolutional layer, like all layers the error w.r.t the next layers input will be provided. From those, 2 other partial derivatives must be computed. Firstly, the partial derivatives of the layer’s loss must be computed w.r.t the weights (in this case the filters). Secondly the error w.r.t the input of this layer must be computed [52]. The error w.r.t to filter weight can be used to modify the weights in affect changing how the convolutional layer extracts features [53].

Max Pooling
Max pooling further down samples the feature maps based on a pool size (size of a sub region of the feature map) created during the forward pass of the convolutional layer. Max pooling layers often appear in paired succession with convolution layers in CNN architecture. Max

pooling helps prevent overfitting by abstracting the representation [54] and also reduces the size of each feature map shortening computation time. Figure 3.3.4f illustrates max pooling showing a section of a feature map and how max pooling applies a 2x2 pooling shape to generate a refined feature map.

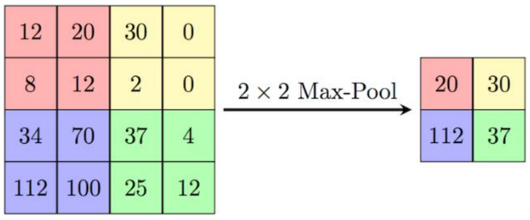
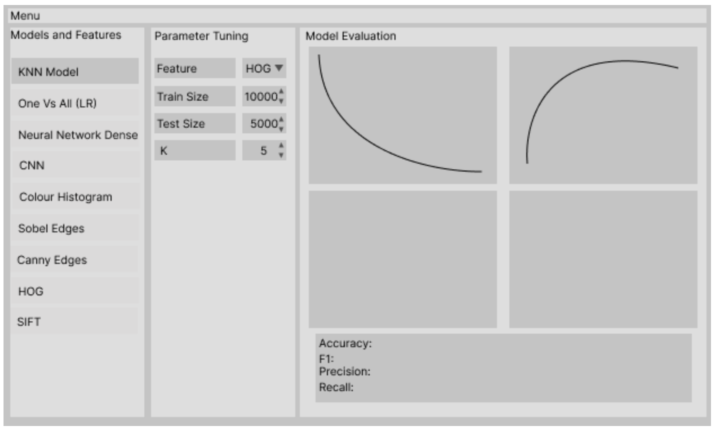


Figure 3.3.4f: Max Pooling [55].

The back propagation of max pooling is comparatively simpler in its implementation than convolution. As there are no weight errors to calculate and no activation function is applied to the layer output, derivatives from the next layer are simply placed in a zero matrix at the position where the feature maps maximum value was found.

3.4 UI Design

Figure 3.2 shows the design for the UI. Each feature of the classifier is accessible through the left-most pane. On click, each button updates the properties panel and associated evaluation pane. The parameter tuning pane allows the end user to adjust the parameters for each feature in this case the ‘K’ value for KNN and the feature vector for KNN to run against in this case HOG. The evaluation pane contains metrics and graphs of the model results. In the case of feature extraction, randomly selected CIFAR-10 images will be displayed with a visualisation of the feature vector after the technique has been applied with the specified parameters.



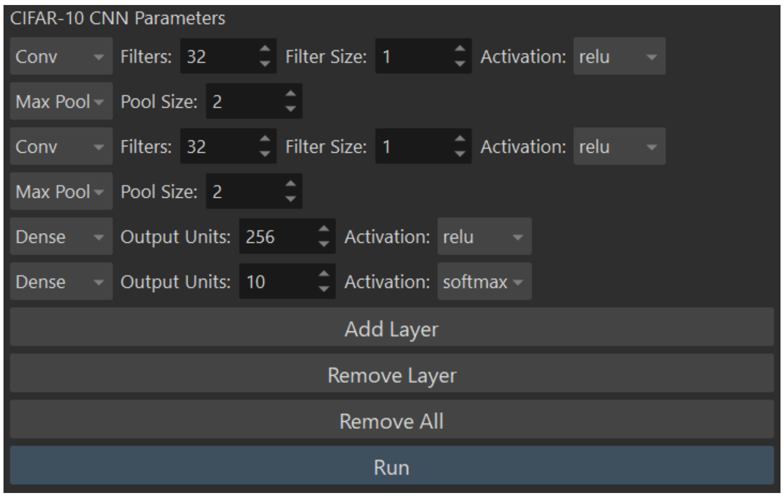
Feature HOG
Train Size 10000
Test Size 5000
K 5

Accuracy:
F1:
Precision:
Recall:

Figure 3.4a: Sketch of the proposed UI.

The 3-way split view seemed the most logical way to layout the UI. Each feature has been developed independently and should be displayed separately, each requiring bespoke parameters. The UI explicitly maps the parameters required in the underlying classes ensuring the end user has the same level of control as a developer accessing public methods.

Custom UI components have been created for building neural network models. With each layer configurable via the UI. Changing a layer type adjusts the parameter fields and sets them applicable to that layer type parameters this is illustrated in Figure 3.4b with each layer and its associated parameters, when the user clicks the 'Run' button the UI components can be converted to a layer that can be added to the sequential model.


CIFAR-10 CNN Parameters

Conv Filters: 32 Filter Size: 1 Activation: relu
Max Pool Pool Size: 2
Conv Filters: 32 Filter Size: 1 Activation: relu
Max Pool Pool Size: 2
Dense Output Units: 256 Activation: relu
Dense Output Units: 10 Activation: softmax

Figure 3.4b: CNN Model UI.

The UI also exposes metrics data from each of the developed models including precision and recall graphs, ROC curves, accuracy and loss curves and model history.

It should be noted that the majority of development time focused on building and testing models from the command line and the UI was a relatively late inclusion. More time would be required to ensure the UI exposed all of the underlying code base along with handling errors as at this stage of development errors are simply logged to the command line.

3.5 Error Handling

Each component of the system has been designed to minimise the chance of errors. For example, UI limitations have been put in place to limit input ranges for numeric values along with disallowing inappropriate input types i.e., string data in numeric fields. In the case of numeric calculations division by zero errors have been mitigated by adding a small epsilon value to the inputs prior to calculation. The UI also provides pre-initialised drop-down menus for application constants. Taking layer type as an example, all layers can be resolved from a string type without relying on correct user input. The application is also focused on providing useful information to developers when building models within the code, exceptions are used extensively along with method level documentation. All model data is stored within the applications data folder ensuring a user has read permissions.

4 Implementation

4.1 Language and Development Environment

The system was implemented using Python 3.9.9 and developed using Visual Studio Code. Python is the most predominant language in the field of machine learning and seemed like the obvious choice for this project. Existing solutions reflect this as the majority use Python. Conveying ideas quickly through each iteration of development was essential and Python helped drive this through its natural syntax style. PEP 8 [56] was adopted as the coding style for the project ensuring classes, functions and variable names retained a consistent naming convention across the code base. Each component of the classifier was developed to include docstring documentation at class and method level. Visual Studio Code was chosen due to its interactive debugging capabilities and the fact that a full featured IDE such as PyCharm wasn’t necessary. Git was chosen as the version control solution due to the need for local and remote backups of code and models, feature branches and as a method for distributing the finished system.

4.2 Project Structure

During initial research into existing solutions this article [57], which presented an approach to structuring machine learning projects separating code, data and reporting. It was decided to adopt this approach albeit in a slightly modified form outlined in Figure 4.2.

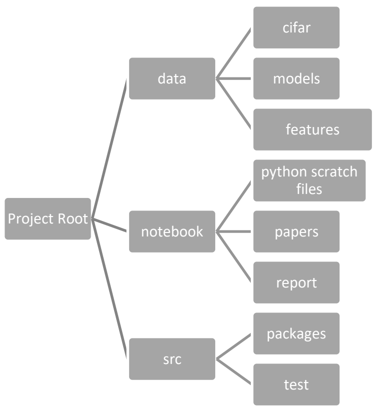


Figure 4.2: Classifier project structure.

4.2 Software Libraries

Given the from scratch nature of the project no machine learning libraries have been used with only libraries that facilitate data loading and natively implemented mathematics being adopted during development. Each library has been included in a requirements file which can facilitate the automatic installation of each dependency using the Python package manager [58].

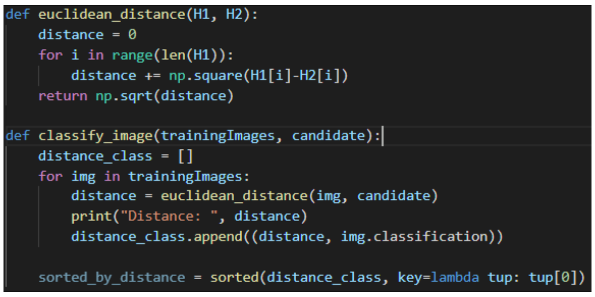
1. NumPy [59]: Perhaps the most important external library which was used for all mathematics and vectorization.
2. Matplotlib [60]: Matplotlib was used to plot model results and to visualise images.
3. Pillow [61]: The Python imaging library was used to load images when testing extraction algorithms on images not included in CIFAR-10 and when building HOG visual images due to the need for alpha channel support when combining images for each histogram angle.
4. Pickle [62]: Pickle was used to load CIFAR-10 as well as saving and loading persisted models.
5. PyQt5 [63]: PyQt5 was chosen as the UI framework for the system. Primarily due to previous experience in the context of C++ GUI development and its custom widget capabilities necessary when constructing the CNN UI. PyQt5 also integrates with Matplotlib graphs so existing image visualization methods can be easily added to the UI without the need to develop two separate visualization systems.

4.3 Important Implementation Decisions

Throughout development a number of implementation decisions were made to improve the performance of each aspect of the classifier, these mainly focused on using NumPy broadcasting to vectorize implementations previously implemented using Python which performed poorly on large samples of the dataset. Some of these key decisions are outlined below.

4.3.1 Vectorizing Euclidean Distance

The KNN model as stated in the design section required that each test sample is compared with all training samples at runtime to perform a prediction. The initial method of performing this action used a loop to iterate across each training sample and calculate the distance between it and the test case. Despite the fact that NumPy performed the calculation, each sample had to be compared to all the training samples leading to an architecture with a time complexity of $O(N^2)$. Figure 4.3.1a is a snippet of KNN code taken from an early stage of development and shows the nested nature of the KNN algorithm which performed incredibly poorly.



```python
def euclidean_distance(H1, H2):
    distance = 0
    for i in range(len(H1)):
        distance += np.square(H1[i]-H2[i])
    return np.sqrt(distance)

def classify_image(trainingImages, candidate):
    distance_class = []
    for img in trainingImages:
        distance = euclidean_distance(img, candidate)
        print("Distance: ", distance)
        distance_class.append((distance, img.classification))

    sorted_by_distance = sorted(distance_class, key=lambda tup: tup[0])
```

Figure 4.3.1a: Early approach to KNN.

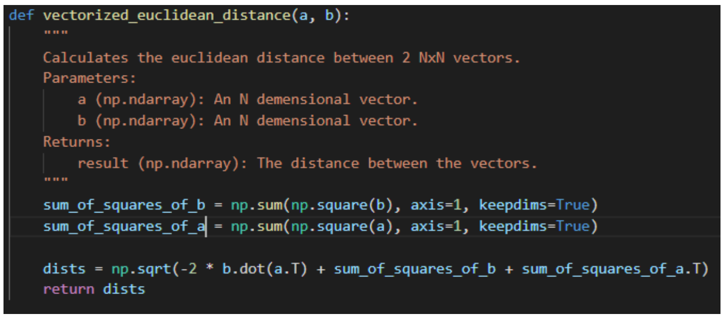
Vectorizing this approach became key to developing a fast KNN algorithm and also formed the basis for developing each subsequent algorithm (where possible) utilising vectorization. As both the training and test data are contained within a matrix with each row being of the same dimension (and the flattened nature of each feature vector). To enable full vectorization the two matrices must be independently squared and summed. This can be achieved by expanding part of the Euclidean formula shown in Figure 4.3.1b.

$$euclidean(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

$$(x - y)^2 = x^2 + y^2 - 2xy$$

Figure 4.3.1b: Expansion of L2 norm of Euclidean distance.

Each element in both matrices can first be squared using NumPy across the first axis of the matrix (second dimension as NumPy axis are 0 indexed) then summed. The multiplication of $2xy$ can be achieved by taking the dot product of x with the transpose of y. Figure 4.3.1c shows the current implementation.

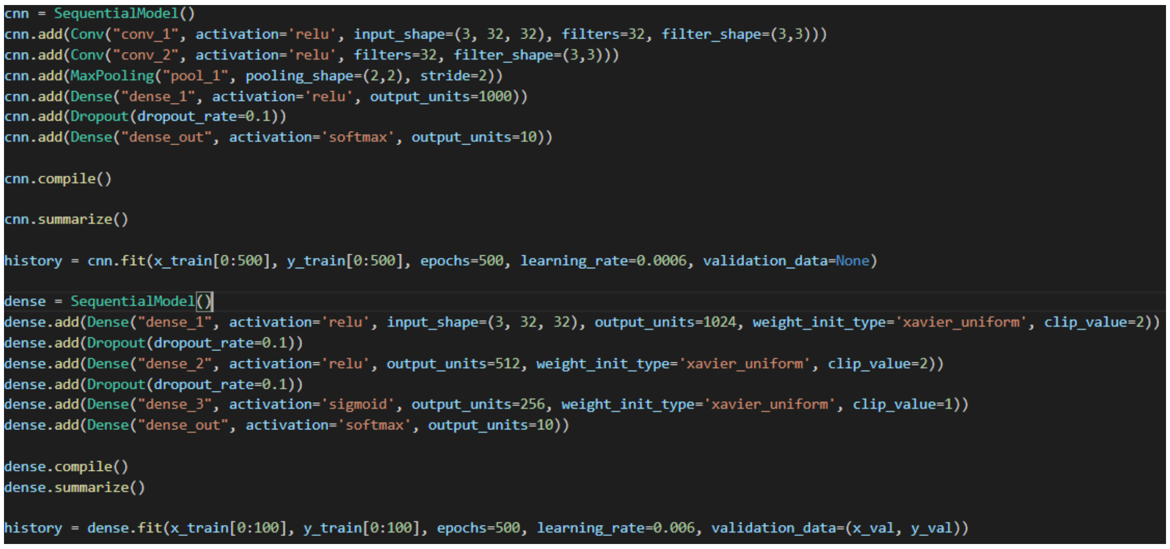


```python
def vectorized_euclidean_distance(a, b):
    """
    Calculates the euclidean distance between 2 NxN vectors.
    Parameters:
        a (np.ndarray): An N demensional vector.
        b (np.ndarray): An N demensional vector.
    Returns:
        result (np.ndarray): The distance between the vectors.
    """
    sum_of_squares_of_b = np.sum(np.square(b), axis=1, keepdims=True)
    sum_of_squares_of_a = np.sum(np.square(a), axis=1, keepdims=True)

    dists = np.sqrt(-2 * b.dot(a.T) + sum_of_squares_of_b + sum_of_squares_of_a.T)
    return dists
```

Figure 4.3.1c: Vectorized implementation.

4.3.2 Neural Network development

One of the factors driving neural network development was to ensure a user or developer could adapt their network designs with ease. Each aspect of a network's functionality is encapsulated into a separate layer class. Figure 4.3.2 shows how a developer can build a multi-layer network by taking the developed system and adding layers as desired.



```python
cnn = SequentialModel()
cnn.add(Conv("conv_1", activation='relu', input_shape=(3, 32, 32), filters=32, filter_shape=(3,3)))
cnn.add(Conv("conv_2", activation='relu', filters=32, filter_shape=(3,3)))
cnn.add(MaxPooling("pool_1", pooling_shape=(2,2), stride=2))
cnn.add(Dense("dense_1", activation='relu', output_units=1000))
cnn.add(Dropout(dropout_rate=0.1))
cnn.add(Dense("dense_out", activation='softmax', output_units=10))

cnn.compile()

cnn.summarize()

history = cnn.fit(x_train[0:500], y_train[0:500], epochs=500, learning_rate=0.0006, validation_data=None)

dense = SequentialModel()
dense.add(Dense("dense_1", activation='relu', input_shape=(3, 32, 32), output_units=1024, weight_init_type='xavier_uniform', clip_value=2))
dense.add(Dropout(dropout_rate=0.1))
dense.add(Dense("dense_2", activation='relu', output_units=512, weight_init_type='xavier_uniform', clip_value=2))
dense.add(Dropout(dropout_rate=0.1))
dense.add(Dense("dense_3", activation='sigmoid', output_units=256, weight_init_type='xavier_uniform', clip_value=1))
dense.add(Dense("dense_out", activation='softmax', output_units=10))

dense.compile()
dense.summarize()

history = dense.fit(x_train[0:100], y_train[0:100], epochs=500, learning_rate=0.006, validation_data=(x_val, y_val))
```

Figure 4.3.2: Development of two models.

Provided an end user supplies an input shape to the first layer of each model, the compile method will determine the input and output shape for each of the subsequent layers in the model. Compile will also initialize each of the layer's weight matrices using the method supplied in the constructor. Each layer can also have a name (this was included to aid debugging) and will be printed for each layer along with its parameters when the summarize method is called for a model.

4.3.3 A Faster Approach to Convolution

During feature extraction development convolution was implemented using Python for loops. Mainly due to the fact that features could be precalculated once for the dataset with little need for recalculation in the future. Project development focused on achieving a minimal viable product and the current speed of convolution appeared acceptable at that point. However, when developing the convolutional layer of the neural network it quickly became apparent that training speed was severely impacted by Python convolution. A typical convolution layer could have 32 or 64 filters each 3x3 in size. If a network contained 3 convolutional layers each generating 32 feature maps per input channel, that could lead to upwards of 32^3convolution per CIFAR-10 input. Existing solutions to this problem do of course exist for Python projects with SciPy offering a convolve function [64] which is implemented natively in C. NumPy also offers a convolve function but kernels are limited to a one-dimensional vector [65]. Initially it was thought that 2D convolution could be implemented using NumPy by padding an image as before, and tiling the convolutional kernel to match the image size and then simply multiplying the two. However, this would eliminate the ability to perform strided convolutions. Further research led to the discovery of NumPy memory layout tricks [66], which enables modification to how NumPy indexes elements of an array. NumPy's sliding window view function was used to extract strided sub regions of an image which could be multiplied with a tiled kernel to convolve an image. This was in the region of ten times faster than convolving using loops. Both methods have been retained within the code base, the faster method targeting the CNN.

4.3.4 Max Pooling Back Propagation

Back propagation of the max pooling layer was difficult to implement without the use of loops. A loop version can be found in the project's Git history. However, an approach similar to the one employed when developing the faster convolution algorithm was used. In order to prove that max pooling using this approach worked correctly small test images where fed into the max pooling layer. Figure 4.3.4 shows the initial and resultant images plotted during max pooling back propagation. The partial derivatives w.r.t the inputs are mocked in this case. The left most image shows a sample image (retained in memory from the forward pass), the middle image highlights the position of the max values for each 2x2 region (pooling shape) and the right most image is the partial derivatives mapped to the appropriate areas where the maximum value was found. Mapping was achieved by tiling the derivative matrix by the factor of the stride value, next the indices of the max values of the input image were extracted using a sliding window view and argmax and used to create a Boolean mask array to set all non-max indices in the derivative matrix to 0.

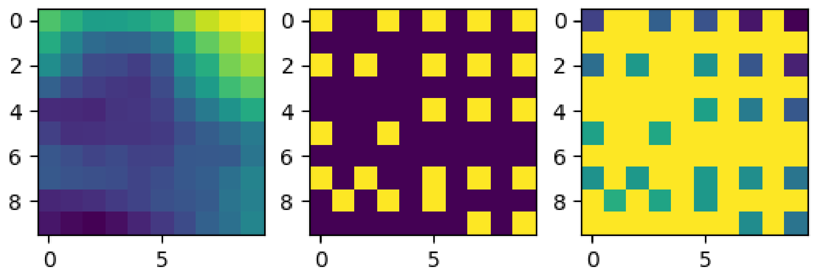
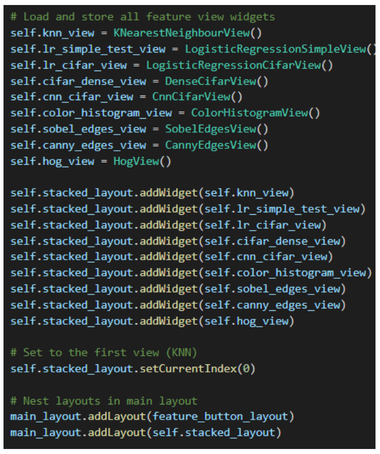


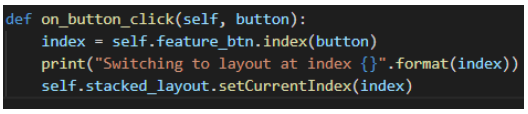
Figure 4.3.4: Proving max pooling.

4.3.5 UI Implementation

As discussed PyQt5 was chosen as the GUI framework. One Main Window was created as a central hub for the classifier. A sequence of view widgets were created one for each model and feature. PyQt5 allows widgets to be swapped out at will via a Stacked Layout. Each view was added to the layout and each menu button was mapped to the view's index in the stacked layout as shown in Figure 4.3.5a.



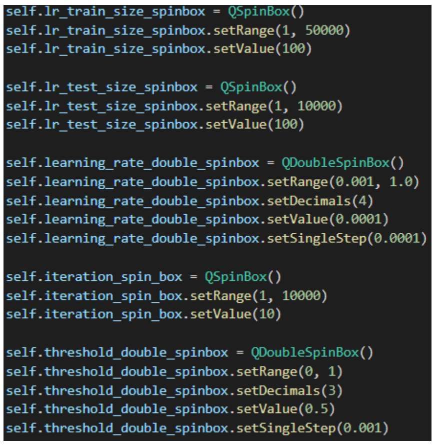
```python
# Load and store all feature view widgets
self.knn_view = KNearestNeighbourView()
self.lr_simple_test_view = LogisticRegressionSimpleView(
self.lr_cifar_view = LogisticRegressionCifarView()
self.cifar_dense_view = DenseCifarView()
self.cnn_cifar_view = CnnCifarView()
self.color_histogram_view = ColorHistogramView()
self.sobel_edges_view = SobelEdgesView()
self.canny_edges_view = CannyEdgesView()
self.hog_view = HogView()

self.stacked_layout.addWidget(self.knn_view)
self.stacked_layout.addWidget(self.lr_simple_test_view)
self.stacked_layout.addWidget(self.lr_cifar_view)
self.stacked_layout.addWidget(self.cifar_dense_view)
self.stacked_layout.addWidget(self.cnn_cifar_view)
self.stacked_layout.addWidget(self.color_histogram_view)
self.stacked_layout.addWidget(self.sobel_edges_view)
self.stacked_layout.addWidget(self.canny_edges_view)
self.stacked_layout.addWidget(self.hog_view)

# Set to the first view (KNN)
self.stacked_layout.setCurrentIndex(0)

# Nest layouts in main layout
main_layout.addLayout(feature_button_layout)
main_layout.addLayout(self.stacked_layout)
```



```python
def on_button_click(self, button):
    index = self.feature_btn.index(button)
    print("Switching to layout at index {}".format(index))
    self.stacked_layout.setCurrentIndex(index)
```

Figure 4.3.5a: Stacked layout and event handler for changing view.

PyQt5 inputs by default disallow input of an incorrect type. String data cannot be placed in numeric input fields. To further limit the chance of problematic inputs, ranges were applied to each input. Figure 4.3.5b shows how ranges were applied the inputs in the logistic regression view.

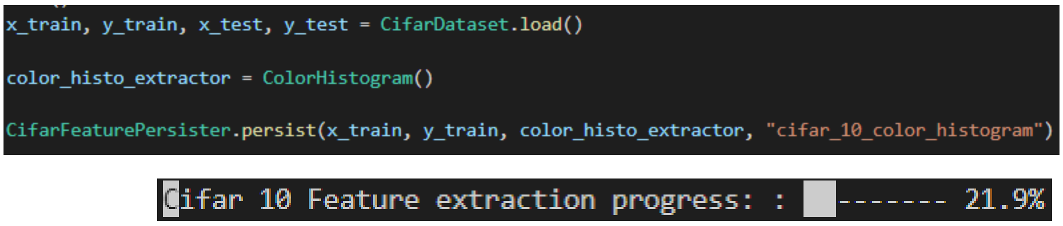


```python
self.lr_train_size_spinbox = QSpinBox()
self.lr_train_size_spinbox.setRange(1, 50000)
self.lr_train_size_spinbox.setValue(100)

self.lr_test_size_spinbox = QSpinBox()
self.lr_test_size_spinbox.setRange(1, 10000)
self.lr_test_size_spinbox.setValue(100)

self.learning_rate_double_spinbox = QDoubleSpinBox()
self.learning_rate_double_spinbox.setRange(0.001, 1.0)
self.learning_rate_double_spinbox.setDecimals(4)
self.learning_rate_double_spinbox.setValue(0.0001)
self.learning_rate_double_spinbox.setSingleStep(0.0001)

self.iteration_spin_box = QSpinBox()
self.iteration_spin_box.setRange(1, 10000)
self.iteration_spin_box.setValue(10)

self.threshold_double_spinbox = QDoubleSpinBox()
self.threshold_double_spinbox.setRange(0, 1)
self.threshold_double_spinbox.setDecimals(3)
self.threshold_double_spinbox.setValue(0.5)
self.threshold_double_spinbox.setSingleStep(0.001)
```

Figure 4.3.5b: Applying input limitations.

4.3.6 Persistence

Figure 4.3.6 outlines the final persistence API for features. This same functionality also applies to trained models. The persist function takes in a dataset along with the extraction class and then calls the extract method in that class. A progress bar is also displayed during extraction. Each developed feature class is interchangeable.



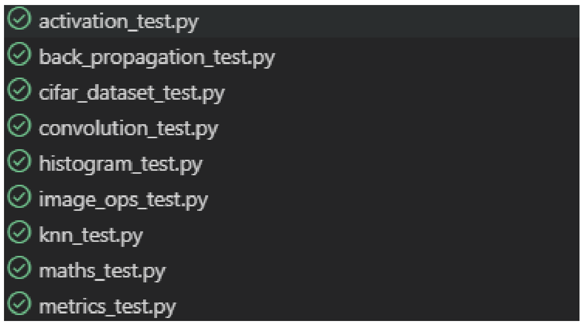
```
x_train, y_train, x_test, y_test = CifarDataset.load()

color_histo_extractor = ColorHistogram()

CifarFeaturePersister.persist(x_train, y_train, color_histo_extractor, "cifar_10_color_histogram")
```

```
Cifar 10 Feature extraction progress: : █------- 21.9%
```

Figure 4.3.6: Histogram extraction and persistence.

5 Testing

The project was primarily tested through unit tests and integration tests. Where necessary unit tests were setup with a range of parameterised inputs. For example, when testing mathematical functions, there are multiple input cases for each test. Some mathematical tests such as sigmoid use expected values gathered from a trusted sigmoid calculator. Metrics functions were tested for mocked data and also tested when evaluating models. Metric tests focused on binary classification and multiclass classification with tests for each. The project makes use of the unittest framework which is a Python testing framework with a workflow similar to JUnit for Java. Each aspect of the system has an associated unit test and all have been tested together manually during model evaluation. Unit tests are also documented. Performance testing has also been conducted as is evidenced in the development history and algorithm refactoring and reimplementation discussed previously. Figure 5 shows the application unit tests files each containing a number of tests.



```
✓ activation_test.py
✓ back_propagation_test.py
✓ cifar_dataset_test.py
✓ convolution_test.py
✓ histogram_test.py
✓ image_ops_test.py
✓ knn_test.py
✓ maths_test.py
✓ metrics_test.py
```

Figure 5: Unit tests.

6 System Evaluation and Experimental Results

6.1 Classifier Evaluation

Each of the developed feature extraction algorithms align with the proposed class design and have been successfully applied to every sample of CIFAR-10 along with external test images. As expected, feature performance ranged massively across models, so to evaluate the success of each technique each was tested against all models, keeping the model variables the same across each feature.

6.1.1 Feature Comparison KNN

Each feature was used to train the KNN model at K=5 with the results shown in Figure 6.1.1.a.

Feature	Accuracy	Ave. Precision	Ave. Recall	F1
RGB	0.34	0.43	0.34	0.33
Grayscale	0.29	0.39	0.29	0.28
Colour Histogram	0.28	0.29	0.28	0.27
HOG	0.21	0.53	0.21	0.16
Sobel	0.25	0.26	0.25	0.23
Canny	0.14	0.20	0.14	0.10

Figure 6.1.1a: Feature metrics KNN K=5

Quite surprisingly the best results for KNN were obtained by performing no feature extraction and instead using the raw CIFAR-10 images (3072 features). Results with regards to RGB and KNN match those gathered during initial project research [5] lending credibility to the implementation of KNN. Running the model with the training and testing sets in their entirety takes approximately 2 minutes depending on vector size with RGB taking the longest time. Canny was the most disappointing feature. It's reasonable to say given that the sample images are so small that much of the image detail is lost through noise reduction and non-maxima suppression. Canny would therefore be more applicable to image datasets where vector size reduction is needed rather than fine detail preservation. There is a correlation between vector size and accuracy. Figure 6.1.1b is a graph of feature size plotted against accuracy.

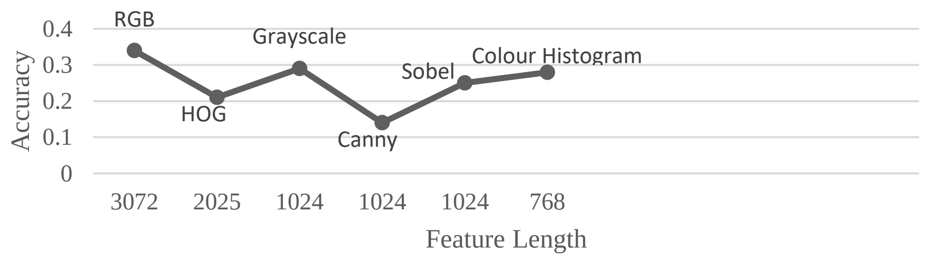


Figure 6.1.1b: Feature length and accuracy.

The graph shows Canny as a clear outlier. This may be due to the sparsity of non-zero values in the feature vector. Canny edges also only have one magnitude value removing colour intensity levels along with discriminative details from the image. Sobel has better metrics as more of the image data is maintained. Figure 6.1.1c demonstrates vector sparsity and lack of details when extracting edges with the Canny algorithm.

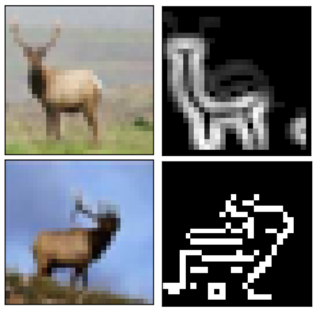


Figure 6.1.1c: Deer sample extracted with Sobel and Canny.

When comparing each feature, the KNN model was kept constant at K=5 as this provided the best overall metrics across the features. However, improvements can be made to the metrics when K is adjusted for each feature. For example, RGB KNN with K=1 increased the accuracy to 0.36.

6.1.2 Proving Logistic Regression

Before training models with the developed features two logistic regression experimental models were created. Firstly, a simple regression model with only one weight to determine whether a number was negative or positive. This model was developed to visualize the probability distribution. Figure 6.1.2a shows the probability distribution and the formation of the logistic curve across three stages of training. The first image shows the randomized initial state of the weight. The second shows a partially trained model with some misclassification and the final shows the fully realised probability distribution.

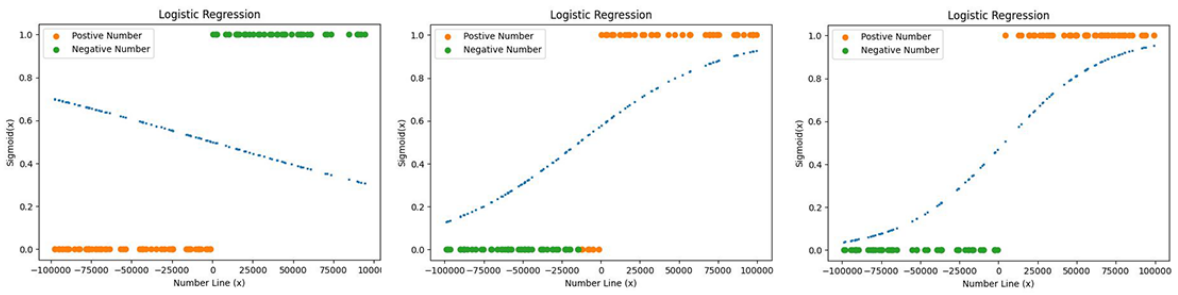


Figure 6.1.2a: Positive negative model training stages.

To evaluate the logistic regression model in a more real-world context whilst focusing on binary classification as a first step, CIFAR-10 was broken down into two classes. Organic (frog, deer, bird, cat, horse, dog) and inorganic (airplane, automobile, ship, truck) and the regression task was completed over 5000 iterations at a learning rate of 0.008. HOG provided the most promising ROC curve with the largest AUC. Figure 6.1.2b shows the ROC curve on test data.

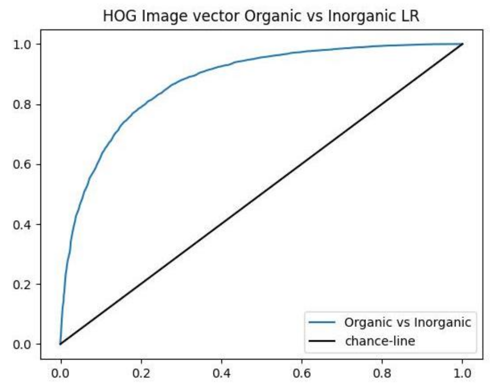


Figure 6.1.2b: Organic vs Inorganic ROC curve.

On average the logistic model takes 5-30 minutes to train depending on the iteration count.

6.1.3 Logistic Regression One vs All

As the one vs all classifier contains 10 classes training time increases significantly. Testing focused on the entire dataset with models taking upwards of 5-10 hours to train. One vs all demanded a much larger number of iterations to discriminate effectively across the 10 classes (up to 10000 per class) with training times also scaling with vector size. The best results were achieved using HOG and RGB. Each individual classifier reached accuracies of over 90%. However, this is not reflective of the models as the dataset is inherently imbalanced as 9/10 of the samples are marked as negative in each classifier. The imbalanced nature of each classifier's training data is reflected in the initial values of loss and accuracy obtained throughout the training period shown in figure 6.1.3a and 6.1.3b, accuracy was measured every 5 iterations. However, loss and accuracy reach a point of class discrimination given enough iterations.

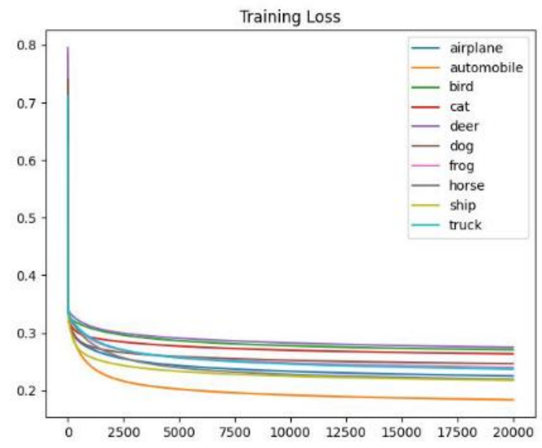


Figure 6.1.3a: One vs all loss curve for each classifier (HOG).

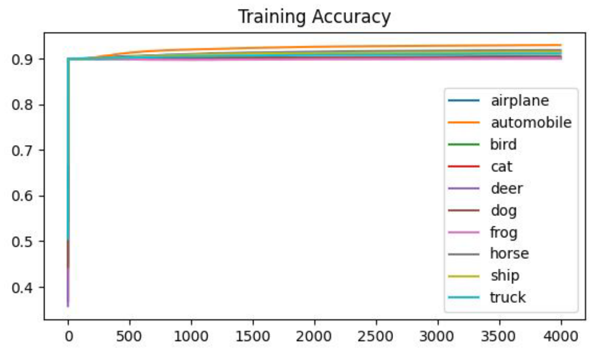


Figure 6.1.3b: One vs all accuracy curve for each classifier (HOG).

Rather than focusing on accuracy, each classifier can instead be evaluated by looking at precision and recall, presented in figure 6.1.3c and 6.1.3d. These metrics are reflective of the test data. Each target class refers to the logistic regression classifier for one of the CIFAR-10 classes where the positive class (1) is the target class and the rest of the samples are marked as the negative class (0).

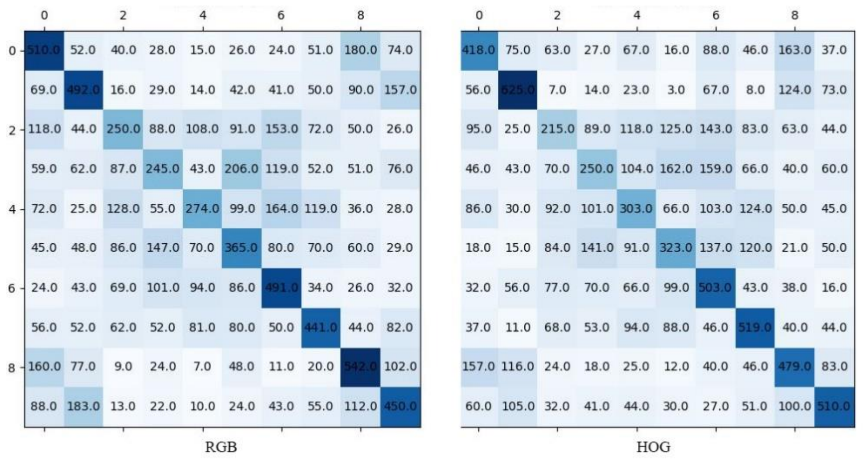
Target Class - Label	Precision	Recall	F1
Airplane - 0	0.60	0.16	0.25
Automobile - 1	0.66	0.21	0.32
Bird - 2	0.41	0.02	0.03
Cat - 3	0.39	0.02	0.04
Deer - 4	0.47	0.04	0.07
Dog - 5	0.50	0.03	0.07
Frog - 6	0.55	0.12	0.19
Horse - 7	0.67	0.13	0.21
Ship - 8	0.52	0.19	0.28
Truck - 9	0.56	0.19	0.28
Average	0.53	0.11	0.18

Figure 6.1.3c RGB metrics.

Target Class - Label	Precision	Recall	F1
Airplane - 0	0.64	0.18	0.28
Automobile - 1	0.72	0.38	0.50
Bird - 2	0.54	0.04	0.07
Cat - 3	0.47	0.04	0.08
Deer - 4	0.28	0.01	0.03
Dog - 5	0.46	0.09	0.15
Frog - 6	0.37	0.08	0.13
Horse - 7	0.71	0.22	0.33
Ship - 8	0.62	0.21	0.32
Truck - 9	0.57	0.16	0.25
Average	0.54	0.14	0.21

Figure 6.1.3d HOG metrics.

When the classifiers are combined and used to predict a test sample the class with the highest probability is taken as the classification. RGB and HOG reached a peak accuracy of 40% and 42% respectively. The two confusion matrices of the test predictions are shown in Figure 6.1.3e.



RGB

	0		2		4		6		8	
0	510.0	52.0	40.0	28.0	15.0	26.0	24.0	51.0	180.0	74.0
	69.0	492.0	16.0	29.0	14.0	42.0	41.0	50.0	90.0	157.0
2	118.0	44.0	250.0	88.0	108.0	91.0	153.0	72.0	50.0	26.0
	59.0	62.0	87.0	245.0	43.0	206.0	119.0	52.0	51.0	76.0
4	72.0	25.0	128.0	55.0	274.0	99.0	164.0	119.0	36.0	28.0
	45.0	48.0	86.0	147.0	70.0	365.0	80.0	70.0	60.0	29.0
6	24.0	43.0	69.0	101.0	94.0	86.0	491.0	34.0	26.0	32.0
	56.0	52.0	62.0	52.0	81.0	80.0	50.0	441.0	44.0	82.0
8	160.0	77.0	9.0	24.0	7.0	48.0	11.0	20.0	542.0	102.0
	88.0	183.0	13.0	22.0	10.0	24.0	43.0	55.0	112.0	450.0

HOG

0		2		4		6		8	
418.0	75.0	63.0	27.0	67.0	16.0	88.0	46.0	163.0	37.0
56.0	625.0	7.0	14.0	23.0	3.0	67.0	8.0	124.0	73.0
95.0	25.0	215.0	89.0	118.0	125.0	143.0	83.0	63.0	44.0
46.0	43.0	70.0	250.0	104.0	162.0	159.0	66.0	40.0	60.0
86.0	30.0	92.0	101.0	303.0	66.0	103.0	124.0	50.0	45.0
18.0	15.0	84.0	141.0	91.0	323.0	137.0	120.0	21.0	50.0
32.0	56.0	77.0	70.0	66.0	99.0	503.0	43.0	38.0	16.0
37.0	11.0	68.0	53.0	94.0	88.0	46.0	519.0	40.0	44.0
157.0	116.0	24.0	18.0	25.0	12.0	40.0	46.0	479.0	83.0
60.0	105.0	32.0	41.0	44.0	30.0	27.0	51.0	100.0	510.0

Figure 6.1.3e: One vs all confusion matrices (RGB left HOG right).

The HOG model provides a high degree of both class specificity and sensitivity. Whilst some of the differences are minimal and there are some exceptions, on average there are fewer false negatives and false positives in the HOG confusion matrix. True positive values are also more evenly distributed in the HOG confusion matrix.

Precision Recall curves and ROC curves were generated for each classifier shown in figures 6.1.3f and 6.1.3g.

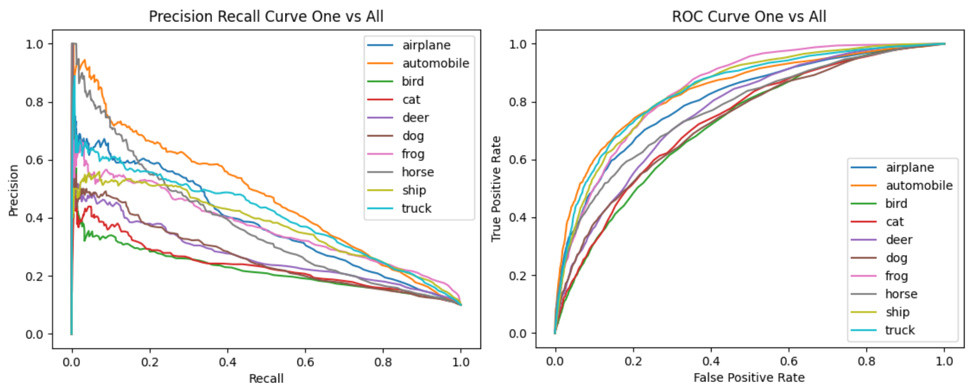


Figure 6.1.3f: Precision/Recall and ROC (RGB)

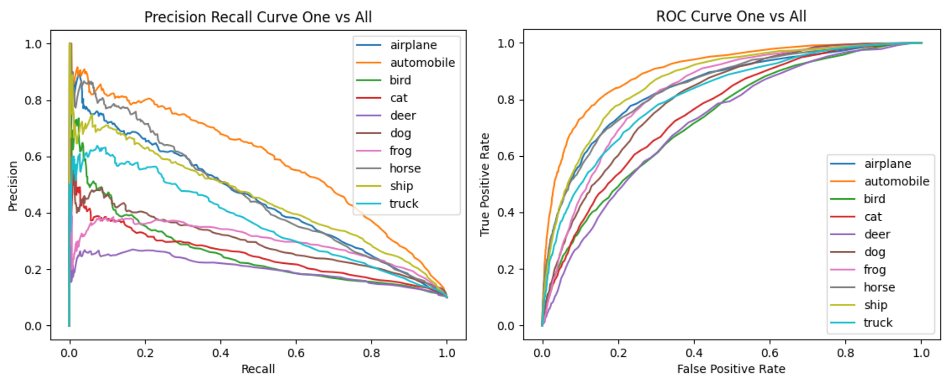


Figure 6.1.3g: Precision/Recall and ROC (HOG)

Both of the graphs generated with HOG exhibit a larger area under the curve compared to the RGB model. The average precision for each classifier was also calculated by taking the area under the graph for each curve using a high number of thresholds, calculating the area (approximated as a series of rectangles). The mean of all the average precisions was calculated as 0.33 for RGB and 0.36 for HOG.

Due to the extensive training times, less promising features were trained using a reduced sample size. From those, the best performing features (Sobel and Colour Histogram) where ran on the full dataset. Again, like KNN the worst performing feature was Canny. Sobel and colour histogram provided higher accuracies than Canny but overall proved less effective as a means of classification compared to RGB and HOG vectors.

6.1.4 Softmax Regression MLP

A series of dense networks were created to evaluate back propagation with regards to the softmax activation function. Before training models with the entire dataset each network was tested with small samples, Figure 6.1.4 shows a 3-layer model with 2 ReLu activated layers and a softmax output layer.

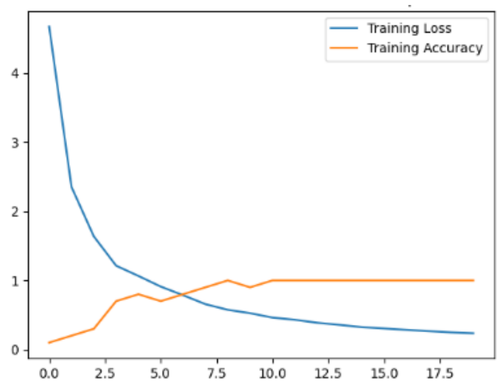


Figure 6.1.4a: Dense network ReLu + Softmax 100 Samples.

The graph shows that the network is learning as the training accuracy increases to an over-fit state as the cross-entropy loss approaches 0 across 20 epochs. Each dense network required vast amounts of time to train across the entire CIFAR-10 set. The gradients are clipped to 1 in this particular model to prevent exploding gradients.

A softmax regression model sampling 1 in 5 (9600 samples) was also trained using HOG vectors. The network contained one fully connected ReLu activated layer (Xavier initialized) with input/output dimensions of 2025 (HOG vector length) and a softmax layer with 10 outputs. The resulting training, validation loss and accuracy curves are shown in figure 6.1.4b. The validation data consisted of 2000 samples extracted from and not inclusive of the training data hence 9600 training samples $\frac{(50000-2000)}{5} = 9600$.

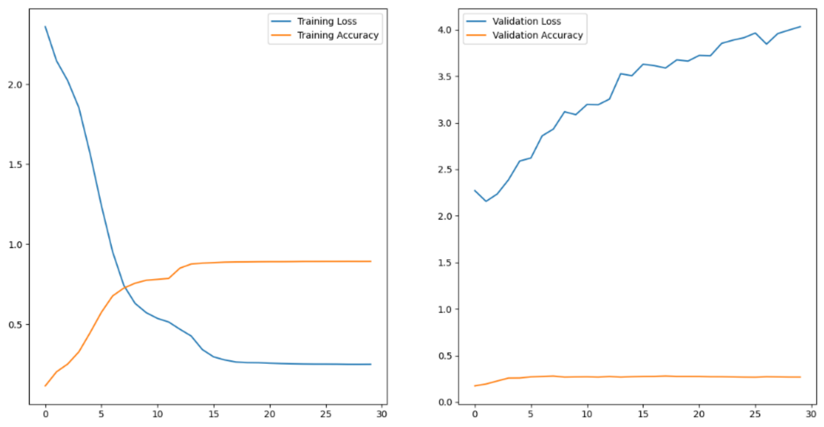


Figure 6.1.4b: Training and validation loss and accuracy curves.

Again, the training curves appear consistent with a learning model with accuracies reaching 90% or greater. Validation accuracy grows from 11% to 27%. However, the loss also increases. It’s uncertain as to why this is occurring. When the model was used to predict the labels of test data the accuracy was 28%. Performing training on larger MLP networks with larger portions of the dataset became infeasible taking large fractions of hours to complete each epoch.

6.1.5 CNN

As of the end of development the CNN is not learning. It is believed that the flaw lies in convolution back propagation. However, the design of the network is robust and the issues could be addressed in future developments. There previously was a loop-based version of the CNN class. However, it took so long to train to produce anything meaningful in terms of metrics. Gradient descent was used to train the models and stochastic gradient descent was tested to try an improve the rate of learning with similarly poor results.

6.1.6 Classifier Comparison

After evaluating each feature with each model, we can draw conclusions to the questions outlined at the start of the project.

The most effective model developed was logistic regression with a peak accuracy of 42% on the test dataset. Considering the chance of correct random classification is 1 in 10 the logistic regression model can be viewed as a significant improvement and the metrics also align with those of existing models. KNN also performed respectably on RGB images but lacked the ability to learn from feature vectors with the highest classification accuracy only reaching 34%. The Dense Neural network has also shown promise when training on smaller samples of the dataset. However, when applied to larger sample sizes i.e., 1 in every 2, the computational time is incredibly extensive taking approximately 12 hours to complete 20 epochs, this is due to the iterative approach to training rather than a vectorized one. The loss across that time does steadily decrease though at the end of development this really cannot be considered a beneficial solution to classifying the CIFAR-10 dataset in its entirety.

Overall, the logistic regression model provides the best balance in terms of computational efficiency and classification accuracy, models can also be trained and saved making them much more appropriate from an end users perspective.

6.2 Evaluation of the Project

6.2.1 Final System Evaluation

Given the potential scope of the project and the final features delivered, the project can be considered a success. Each of the areas outlined in the introduction and requirements sections have been explored and implemented. Each model has shown promise with KNN and Logistic Regression models showing comparable or better metrics than those found during model research [5]. The most disappointing aspect of the system is the convolutional neural network. Much more time would be needed to fully understand and implement back propagation in the convolutional layer. Each component has been developed in an extensible manner that could easily be taken further by a future developer each provides useful documentation along with informative exception messages.

6.2.2 Future Improvements

The majority of development time focused on algorithm implementation. Reaching the end of development there are a number of areas which would require further development.

1. UI: The GUI came as a rather late inclusion during development and it's clear that not all components of the system effectively translate to the UI. To that end if there were more development time available additions including a file dialog browser would be added to enable the user to choose a location for saving data. The ability to load models from file from within the UI would also be added.
2. CNN Development: It's clear that the most lacking model and the one with the most promise is the CNN. Back propagation simply isn't working and more time would be necessary to understand how back propagation can be implemented when using the current approach to convolution.
3. PCA: Had there have been more development time available it would have been interesting to implement principal component analysis [67] in order to reduce feature vector lengths and decrease training time.
4. GPU Integration: Despite implementing much of the system using NumPy vectorization some parts still use iteration. If more time were available further research into Python CUDA [68] would be undertaken to determine whether existing functionality could be parallelized leveraging the GPU.
5. Multi-threading: On the subject of parallelism, the single threaded nature of the application creates challenges when trying to collate results for many combinations of models and features. Multi-threading in Python is challenging (when using the CPython implementation) due to the Global Interpreter Lock

which prevents multiple native threads executing Python code at the same time [69]. During evaluation it became essential to run multiple instances of the application as CPU usage rarely reached 50% when running a single model. One solution would be to create a launcher application that manages the UI along with file handling etc. and giving it the ability to spin off separate worker sub-processes [70] to run extraction and model training.

6. Testing: The application lacks UI functional testing. As a future UI is developed user acceptance testing should also be carried out. Back propagation testing is also lacking with the only working indication coming from loss metrics. Manual gradient checking was added at a late stage of development. However, it isn't extensive.

6.2.3 Closing Thoughts

Whilst I am pleased with how far the project has come along, I am disappointed not to have successfully implemented a working CNN. I am happy with the application architecture and think it could be easily extended by someone in the future. As many aspects of the program remained in a state of development during the late stages of the project not enough time was allocated to ensure system resiliency through testing. The application also feels fragmented, I would have liked to have had time to try and combine models through boosting [71]. Having had no prior experience with feature extraction or machine learning before this year I feel that through the development of each of these components I have become much more experienced in Python, machine learning development and have gained an understanding into many of the underlying algorithms otherwise taken for granted when simply using prebuilt libraries.

7 References

The source code for this project can be found in the following Gitlab repository

- FYP - Cifar Image Classifier
  https://gitlab2.eeecs.qub.ac.uk/40230638/FYP-Cifar-Image-Classifier

[1] J. Fu και Y. Rui, ‘Advances in deep learning approaches for image tagging’, APSIPATransactions on Signal and Information Processing, τ. 6, 2017.

[2] H. Chae, C. M. Kang, B. Kim, J. Kim, C. C. Chung, and J. W. Choi, “Autonomous braking system via deep reinforcement learning,” in 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), 2017, pp. 1–6.

[3] Daniel Piekarz “PharmaBoardroom - standing on the shoulders of giants: The art & science of machine learning & AI for drug discovery & repurposing,” PharmaBoardroom, 08-May-2020. [Online]. Available: https://pharmaboardroom.com/articles/standing-on-the-shoulders-of-giants-the-art-science-of-machine-learning-ai-for-drug-discovery-repurposing/. [Accessed: 08-Apr-2022].

[4] N. O’Mahony et al., “Deep learning vs. Traditional computer vision,” in Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2020, pp. 128–144.

[5] Y. Abouelnaga, O. S. Ali, H. Rady, and M. Moustafa, “CIFAR-10: KNN-based ensemble of classifiers,” arXiv [cs.CV], 2016.

[6] “scikit-image: Image processing in Python — scikit-image,” Scikit-image.org. [Online]. Available: https://scikit-image.org/. [Accessed: 11-Apr-2022].

[7] “Home,” OpenCV, 09-Feb-2021. [Online]. Available: https://opencv.org/. [Accessed: 11-Apr-2022].

[8] "scikit-learn: machine learning in Python — scikit-learn 0.16.1 documentation", Scikit-learn.org, 2022. [Online]. Available: https://scikit-learn.org/. [Accessed: 11-Apr- 2022].

[9] Keras Team, “Keras: the Python deep learning API,” Keras.io. [Online]. Available: https://keras.io/. [Accessed: 11-Apr-2022].

[10] “Cython: C-Extensions for Python,” Cython.org. [Online]. Available: https://cython.org/. [Accessed: 11-Apr-2022].

[11] “Cython vs CPython for faster Python,” Contract Engineering, Product Design & Development Company - Cardinal Peak, 19-Aug-2016. [Online]. Available: https://www.cardinalpeak.com/blog/faster-python-with-cython-and-pypy-part-2. [Accessed: 11-Apr-2022].

[12] Keras Team, “Code examples,” Keras.io. [Online]. Available: https://keras.io/examples/. [Accessed: 11-Apr-2022].

[13] A. Kuznetsova et al., “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale,” arXiv [cs.CV], 2018.

[14] “CIFAR-10 and CIFAR-100 datasets,” Toronto.edu. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 11-Apr-2022].

[15] B. Barz and J. Denzler, “Do we train on test data? Purging CIFAR of near-duplicates,” J. Imaging, vol. 6, no. 6, p. 41, 2020.

[16] S. Padhy, Z. Nado, J. Ren, J. Liu, J. Snoek, and B. Lakshminarayanan, “Revisiting one-vs-all classifiers for predictive uncertainty and out-of-distribution detection in neural networks,” arXiv [cs.LG], 2020.
[17] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05), 2005, vol. 1, pp. 886–893 vol. 1.
[18] “Image enhancement using spatial domain filtering,” School of Computer Science. [Online]. Available: https://socs.binus.ac.id/2017/03/20/image-enhancement-using-spatial-domain-filtering/. [Accessed: 15-Apr-2022].
[19] “An implementation of Sobel edge detection - rhea,” Projectrhea.org. [Online]. Available: https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection. [Accessed: 16-Apr-2022].
[20] N. Tsankashvili, “Comparing edge detection methods - Nika tsankashvili,” Medium, 20-Jan-2018. [Online]. Available: https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e. [Accessed: 15-Apr-2022].
[21] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” arXiv [stat.ML], 2016.
[22] J. Canny, "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.
[23] “Non Maximum Suppression: Theory and implementation in PyTorch,” LearnOpenCV – OpenCV, PyTorch, Keras, Tensorflow examples and tutorials, 02-Jun-2021. [Online]. Available: https://learnopencv.com/tag/non-maximum-suppression/. [Accessed: 16-Apr-2022].
[24] Sciencedirect.com. [Online]. Available: https://www.sciencedirect.com/topics/mathematics/gaussian-filter. [Accessed: 16-Apr-2022].
[25] Ed.ac.uk. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm. [Accessed: 16-Apr-2022].
[26] “What does Sigma mean in Gaussian filter?,” Mulloverthings.com. [Online]. Available: https://mulloverthings.com/what-does-sigma-mean-in-gaussian-filter/. [Accessed: 16-Apr-2022].
[27] Umd.edu. [Online]. Available: https://www.cs.umd.edu/class/fall2019/cmsc426-0201/files/11_CannyEdgeDetection.pdf. [Accessed: 16-Apr-2022].
[28] “Canny edge detection,” Iitd.ac.in, 2009. [Online]. Available: https://www.cse.iitd.ac.in/~pkalra/col783-2017/canny.pdf. [Accessed: 16-Apr-2022].
[29] olaniyan, “H.O.G for image classification,” Kaggle.com, 27-May-2019. [Online]. Available: https://www.kaggle.com/code/olaniyan/h-o-g-for-image-classification/notebook. [Accessed: 19-Apr-2022].
[30] T. Yamuna, “Human identification based on the Histogram of Oriented Gradients,” Ijert.org. [Online]. Available: https://www.ijert.org/research/human-identification-based-on-the-histogram-of-oriented-gradients-IJERTV3IS071378.pdf. [Accessed: 19-Apr-2022].
[31] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” Int. J. Comput. Vis., vol. 60, no. 2, pp. 91–110, 2004.

[32] M. Ning, “SIFT(Scale-invariant feature transform),” Towards Data Science, 09-Apr-2019. [Online]. Available: https://towardsdatascience.com/sift-scale-invariant-feature-transform-c7233dc60f37. [Accessed: 19-Apr-2022].
[33] “Laplacian of Gaussian Filter,” Academic.mu.edu. [Online]. Available: https://academic.mu.edu/phys/matthysd/web226/Lab02.htm. [Accessed: 20-Apr-2022].
[34] T. Lindeberg, “Scale-space theory: A basic tool for analysing structures at diferent scales,” Kth.se. [Online]. Available: https://people.kth.se/~tony/papers/scsptheory-review.jas94.pdf. [Accessed: 20-Apr-2022].
[35] H.-K. Ryu, J.-K. Lee, E.-T. Hwang, L. Jing, H.-H. Lee, and W.-H. Choi, “A new corner detection method of gray-level image using Hessian matrix,” in 2007 International Forum on Strategic Technology, 2007, pp. 537–540.
[36] S. K. Kumar, “On weight initialization in deep neural networks,” arXiv [cs.LG], 2017.
[37] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” AISTATS, 2010.
[38] E. W. Weisstein, “Sigmoid function,” https://mathworld.wolfram.com/SigmoidFunction.html
[39] H. Bonthu, “An introduction to logistic Regression,” Analytics Vidhya, 11-Jul-2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/07/an-introduction-to-logistic-regression/. [Accessed: 22-Apr-2022].
[40] N. Kang, “Multi-layer neural networks with sigmoid function— Deep Learning for Rookies (2),” Towards Data Science, 27-Jun-2017. [Online]. Available: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f. [Accessed: 22-Apr-2022].
[41] R. S. Bhat, “Why not Mean Squared Error(MSE) as a loss function for Logistic Regression?” Towards Data Science, 15-Sep-2019. [Online]. Available: https://towardsdatascience.com/why-not-mse-as-a-loss-function-for-logistic-regression-589816b5e03c. [Accessed: 22-Apr-2022].
[42] W. M. Cs, Presentica.com, 2017. [Online]. Available: https://doc.presentica.com/10725479/5ea7fd15d5fe8.pdf. [Accessed: 22-Apr-2022].
[43] E. Chigozie, W. Nwankpa, A. Ijomah, and S. Gachagan, “Activation functions: Comparison of trends in practice and research for deep learning,” Arxiv.org. [Online]. Available: https://arxiv.org/pdf/1811.03378.pdf. [Accessed: 23-Apr-2022].
[44] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, and G. E. Hinton, “On rectified linear units for speech processing,” in International Conference on Acoustics, Speech and Signal Processing. IEEE, 2013, pp. 3517–3521, IEEE. https://doi.org/10.1109/ICASSP.2013.6638312.
[45] B. Xu, N. Wang, T. Chen, and M. Li, “Empirical evaluation of rectified activations in convolutional network,” arXiv [cs.LG], 2015.
[46] A. A. Minai and R. D. Williams, “On the derivatives of the sigmoid,” Neural Netw., vol. 6, no. 6, pp. 845–853, 1993.
[47] Arc, “Derivative of the sigmoid function,” Towards Data Science, 07-Jul-2018. [Online]. Available: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e. [Accessed: 23-Apr-2022].

[48] D. Radečić, “Softmax activation function explained,” Towards Data Science, 18-Jun-2020. [Online]. Available: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60. [Accessed: 02-May-2022].
[49] I. Vasyltsov and W. Chang, “Efficient softmax approximation for deep neural networks with attention mechanism,” arXiv [cs.LG], 2021.
[50] G. Philipp and J. G. Carbonell, “The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions,” Arxiv.org. [Online]. Available: https://arxiv.org/pdf/1712.05577v4.pdf. [Accessed: 23-Apr-2022].
[51] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” Jmlr.org. [Online]. Available: https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf. [Accessed: 23-Apr-2022].
[52] Z. Zhang, “Derivation of backpropagation in convolutional neural network (CNN),” Github.io, 2016. [Online]. Available: https://zzutk.github.io/docs/reports/2016.10%20-%20Derivation%20of%20Backpropagation%20in%20Convolutional%20Neural%20Network%20(CNN).pdf. [Accessed: 23-Apr-2022].
[53] M. Agarwal, “Back propagation in convolutional neural networks — intuition and code,” Becoming Human: Artificial Intelligence Magazine, 14-Dec-2017. [Online]. Available: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199. [Accessed: 23-Apr-2022].
[54] DeepAI, “Max Pooling,” DeepAI, 17-May-2019. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/max-pooling. [Accessed: 23-Apr-2022].
[55] “Max-pooling / pooling,” Computersciencewiki.org. [Online]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling. [Accessed: 29-Apr-2022].
[56] “PEP 8 – style guide for python code,” Python.org. [Online]. Available: https://peps.python.org/pep-0008/. [Accessed: 24-Apr-2022].
[57] Pykes, K., 2020. Structuring Machine Learning projects. [online] Medium. Available at: https://towardsdatascience.com/structuring-machine-learning-projects-be473775a1b6
[58] “Pip 22.0.4,” PyPI. [Online]. Available: https://pypi.org/project/pip/. [Accessed: 24-Apr-2022].
[59] “NumPy documentation — NumPy v1.22 Manual,” Numpy.org. [Online]. Available: https://numpy.org/doc/stable/. [Accessed: 24-Apr-2022].
[60] “Matplotlib — Visualization with Python,” Matplotlib.org. [Online]. Available: https://matplotlib.org/. [Accessed: 24-Apr-2022].
[61] “Pillow,” PyPI. [Online]. Available: https://pypi.org/project/Pillow/. [Accessed: 24-Apr-2022].
[62] “pickle — Python object serialization — Python 3.10.4 documentation,” Python.org. [Online]. Available: https://docs.python.org/3/library/pickle.html. [Accessed: 24-Apr-2022].
[63] “PyQt5,” PyPI. [Online]. Available: https://pypi.org/project/PyQt5/. [Accessed: 24-Apr-2022].

[64] “scipy.signal.convolve — SciPy v1.8.0 Manual,” Scipy.org. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html. [Accessed: 24-Apr-2022].
[65] “numpy.convolve — NumPy v1.22 Manual,” Numpy.org. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.convolve.html. [Accessed: 24-Apr-2022].
[66] “numpy.lib.stride_tricks.as_strided — NumPy v1.22 Manual,” Numpy.org. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.lib.stride_tricks.as_strided.html. [Accessed: 24-Apr-2022].
[67] A. J. Milewska, D. Jankowska, D. Citko, T. Więsak, B. Acacio, and R. Milewski, “The use of Principal Component Analysis and logistic regression in prediction of infertility treatment outcome,” Stud. Log. Gramm. Rhetor., vol. 39, no. 1, pp. 7–23, 2014.
[68] “GPU Accelerated Computing with Python,” NVIDIA Developer, 19-Nov-2013. [Online]. Available: https://developer.nvidia.com/how-to-cuda-python. [Accessed: 26-Apr-2022].
[69] “GlobalInterpreterLock - Python Wiki,” Python.org. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed: 26-Apr-2022].
[70] “RQ: Workers,” Python-rq.org. [Online]. Available: https://python-rq.org/docs/workers/. [Accessed: 26-Apr-2022].
[71] B. Campillo-Gimenez, S. Bayat, and M. Cuggia, “Coupling K-nearest neighbors with logistic regression in case-based reasoning,” Stud. Health Technol. Inform., vol. 180, pp. 275–279, 2012.