THE USE OF SYNTHETIC VS REAL DATA IN THE TRAINING OF AN OBJECT DETECTION NEURAL NETWORK

Nasser Ksous, 18013044, James Banton and David White

Abstract

The method of object detection is done using neural networks which require large datasets of training images to train the network to detect these objects. There are currently no available datasets for the use of training a neural network to detect objects from afar as if it was a camera on a helicopter. Therefore, the use of synthetic data will need to be tested as an alternative to real data. Research has been done to investigate the current state of object detection and compare the two main object detector models: R-CNN and YOLO to find the most suitable model for this project. For the artefact, a neural network library has been attempted to be made to create an object detector inspired by the YOLO model to be trained and tested in a synthetic environment. However, due to time constraints, the library produced cannot create and train an object detector instead it can create a classifier. This classifier has been tested using the MNIST dataset and after 1000 iterations, it has been trained to get an accuracy of 91% when it was tested. The produced library can be added to and built upon in the future to be able to produce an object detector neural network and then it can be trained and tested using synthetic data.

Table of Contents

Glossary

VJ Detector – Viola-Jones Detector

HOG – Histogram of Oriented Gradients

DPM – Deformable Part-based Model

NN – Neural Network

CNN – Convolutional Neural Network

R-CNN – Region proposal with Convolutional Neural Network

YOLO – You Only Look Once

XP – eXtreme Programming

RPM – Regional Proposal Network

SVM – Support Vector Machine

mAP – mean Average Precision

AP – Average Precision

TPs – True Positives

FPs – False Positives

IoU – Intersection over Union

BB – Bounding Box

FPS – Frames Per Second

FCN – Fully Convolutional Network

GPU – Graphics Processing Unit

CPU – Central Processing Unit

Key Words

Neural Network, Object Detection, YOLO, R-CNN, Synthetic data, Real-Time

1.0 Introduction

The main problem to overcome when using object detection in cameras attached to helicopters is that the helicopter will be a lot further away from the objects and thus it makes it harder to see the objects. This means that the data used to train the detector will need to have a variety of angles, weather conditions and distances to detect objects in many scenarios that a helicopter will be in. It can be very difficult to acquire in real life and therefore an alternative method of producing data is needed. Synthetic data is currently used as an alternative in the car industry and are created in a synthetic environment to produce a large set of data with a variety of data. This project looks at replicating that concept but for cameras on helicopters. Looking at specifically whether synthetic data or a combination of synthetic and real data can help train an accurate detector that can also work in almost real-time.

1.1 Aim

Investigate and produce an object detector that can be used on helicopter cameras.

1.2 Objectives

- Research into the current state of object detection
- Research and analyse the different methods of creating an object detector to find the best method for this project
- Create an object detector that can take in both synthetic and real data
- Modify the detector to improve accuracy and speed to work in almost real-time
- Implement the object detector in a synthetic environment to test its viability

2.0 Background

To test the quality of synthetic vs real data, an object detector needs to be built so that the datasets can train this detector. Therefore, multiple solutions of example models will need to be compared to decide on the best object detector for detecting objects from a helicopter camera.

2.1 Viola-Jones Detector

Before looking at any potential solutions for this project, the background of object detection is important to look at how the techniques have developed and changed over the years. In the initial stages of object detection, algorithms were used to detect objects in an image that was passed into it. In 2001, P. Viola and M. Jones created one of the first algorithms to detect human faces in real-time. This was a significant milestone for object detection because it was 10 times faster than any other algorithms at the time. As a result of their contribution, the algorithm was named the Viola-Jones (VJ) detector after its creators. The VJ detector used sliding windows which search all potential locations of faces to see if a window contained a face (Viola & Jones, 2001). Figure 1 below shows the output of the VJ detector in which all the faces in the picture are outlined with a bounding box. This is the basic objective of any object detector.

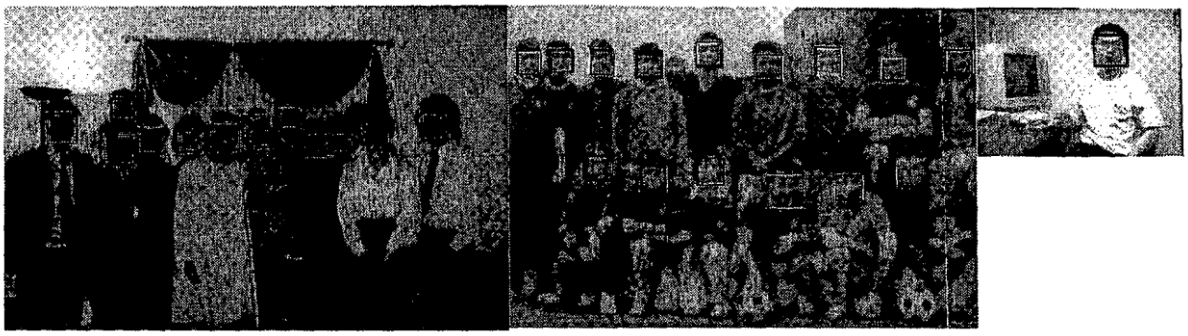


Figure 1: Output of the VJ detector on several test images from the MIT+CMU test set.

2.2 Histogram of Oriented Gradients

The Histogram of Oriented Gradients (HOG) was created by N. Dalal and B. Triggs in 2005. It was a version of the scale-invariant feature transform algorithm that was used on detecting pedestrians of varied sizes. It used a grid of cells of the same size and local contrast normalization which improved the accuracy of the detector. The HOG rescales the input

images whilst keeping the detection windows uniform and is an important base for future detectors (Dalal & Triggs, 2005).

2.3 Deformable Part-Based Model

As an extension of the HOG detector, the Deformable Part-Based Model (DPM) was created by P. Felzenszwalb and later improved greatly by the works of R. Girshick. The DPM was a model that was trained on how to split up an object into parts and thus detect these parts individually instead of looking for the object as a whole (Felzenszwalb, et al., 2008). Previously the filters used to detect parts of an object were manually determined but DPM utilises a supervised learning method that configures these filters automatically. An example of this is shown in figure 2 in which an image of a car is split into distinct parts such as the wheels and doors. These parts are detected using a yellow bounding box and the whole car is detected using a blue bounding box.

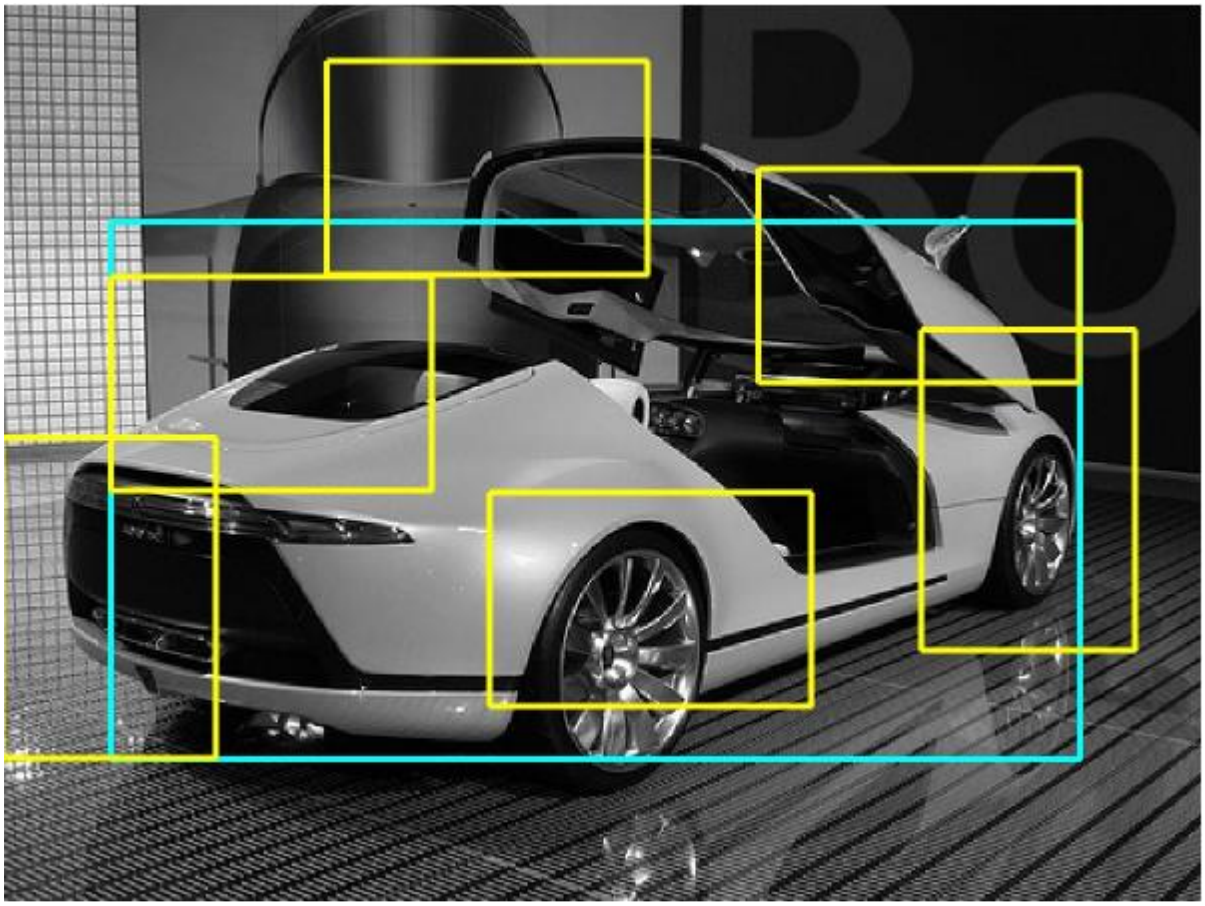


Figure 2: Image of a car with the parts detected with bounding boxes by the DPM

R. Girshick improved the accuracy of this method by using context priming, bounding box regression and hard negative mining techniques. Context priming makes use of the

surroundings around the object to determine what the object is likely to be. Bounding box regression uses prediction bounding boxes to reduce the potential area where the object is likely to be. The location of the prediction boxes is configured manually by the user based on the likely locations and sizes of the objects. A false positive may be produced by an algorithm when it believes that the output bounding box is correct when it is either not on the object or only on part of the object. Hard negative mining techniques take any false positives provided by training the algorithm and label them as negative results so that the algorithm learns that it does not contain an object (Felzenszwalb, et al., 2010).

2.4 Convolutional Neural Networks

2014 saw the start of the deep learning era with the proposal of regions with convolutional neural network (RCNN) by R. Girshick. RCNN produces a set of object candidate boxes that are rescaled and fed into a convolutional neural network (CNN). A neural network (NN) is a computational representation of the human brain in which a model consists of multiple layers of multiple nodes. An image is split into multiple sections and each node represents a section of the image. Each layer uses predefined weights and biases to calculate the output of that layer using the calculation: output = input * weight = bias. The image is processed by each layer in the model to produce an output which is then fed into the next layer. Once every layer has been processed the image output is produced in the form of nodes that represent an object. This is represented in figure 3 below (Khashei & Bijari, 2010).

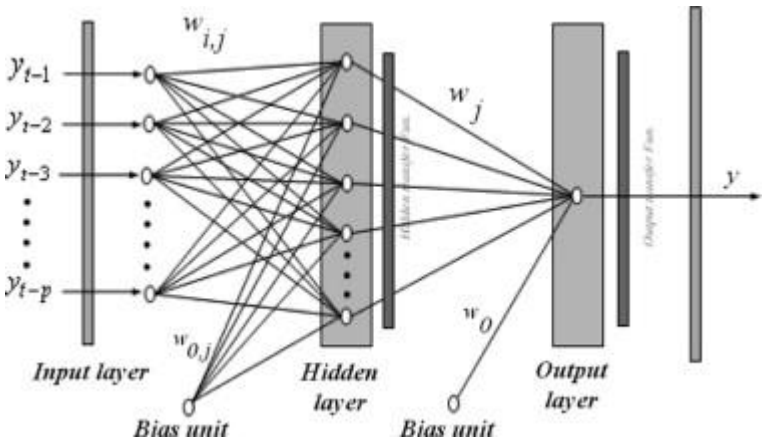


Figure 3: A representation of a neural network of layers and nodes

A CNN is a version of a NN that uses convolutional layers and pooling layers alongside the standard fully connected layer found in neural networks. The convolutional layer uses filters to detect specific parts of the image such as edges and can modify the image to accentuate

certain parts such as making the image sharper. Some examples of filters are shown in figure 4 below (Albawi, et al., 2017).

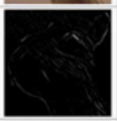
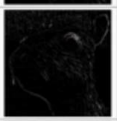
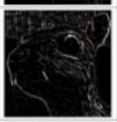
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figure 4: Multiple examples of filters used in the convolutional layer

The pooling layer uses filters on the pixel values produced by the convolutional layer. These filters simply use mathematically functions on each section of the image. For example, a max pool filter of 2x2 shown in figure 5 will take the max value of the 2x2 part of the image. So, a section of an image of 4x4 is reduced to a 2x2 section (Albawi, et al., 2017).

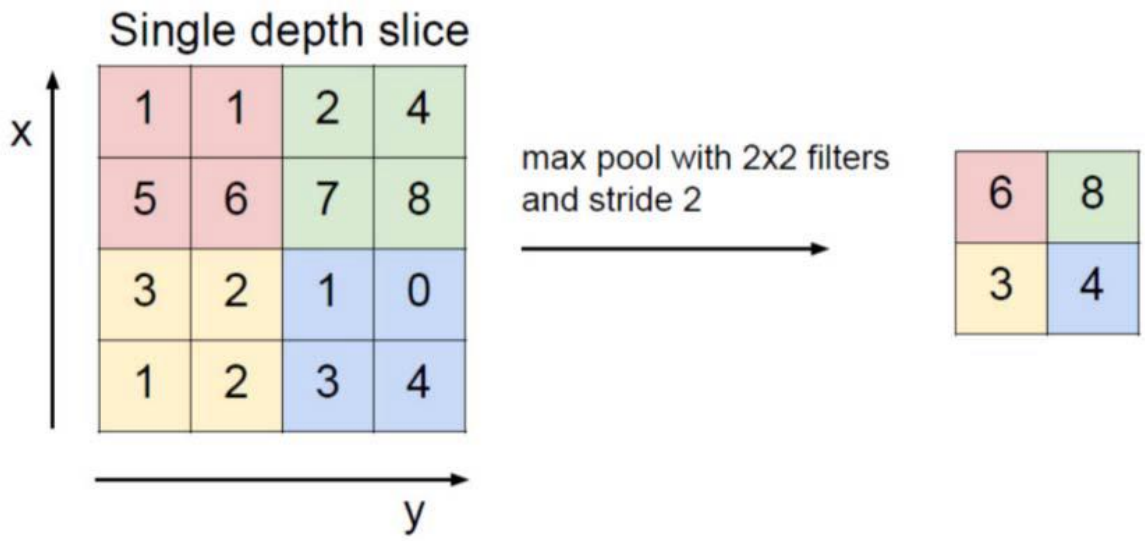


Figure 5: Example of a max pool

2.5 Region proposal with convolutional neural networks

R-CNN produced a significant performance increase and an improvement of the previous algorithms. Even though R-CNN was a massive improvement on previous algorithms, it produces around 2000 overlapped proposals per image which leads to a slow speed of detection (about 14 seconds per image) (Zou, et al., 2019). The RCNN model was then improved on the Fast RCNN and Faster RCNN to improve the speed of detection to make the models detect in almost real-time.

2.6 You Only Look Once

A different approach to object detection using NNs was proposed by (Redmon, et al., 2016). They called it You Only Look Once (YOLO) and as the name suggests, it uses a single neural network to produce prediction bounding boxes and classify the probabilities of an object being detected in one evaluation. A result of using a single neural network makes the model a lot faster than RCNN and can process 155 frames per second. This makes the model a lot more viable for real-time object detection (Redmon, et al., 2016).

3.0 Analysis

Many modern object detectors use CNNs over the previous detection models such as the VJ model and DPM as discussed in the background. This is because the accuracy and speed of these detectors are higher than the previous models. Beyond image detection, the previous models are not suitable for video detection analysis but helped provide the basis for the CNN

models (Jiao, et al., 2021). As a result of this, it is only necessary to compare modern CNN based detectors for use in object detection. As described previously in the background, the two main detectors are the R-CNN and YOLO models.

There is a multitude of different object detection models and to compare them all would be unnecessarily time-consuming since most object detectors have a base architecture similar to that of either R-CNN or YOLO. Therefore, the project will be looking at using R-CNN or YOLO as a base for the object detector whilst also recognising other minor changes to improve these architectures to create a better object detector for use in a helicopter camera.

3.1 R-CNN

Region proposals with convolutional neural network (R-CNN) was produced by R. Girshick. R-CNN has three phases for object detection: producing regional proposals, feeding these proposals into a CNN to extract a fixed-length feature vector from each region and finally using the vectors in a set of class-specific linear SVMs (Girshick, et al., 2014). The first phase uses a selective search algorithm to create about 2000 region proposals on where a potential object could be in an image. The selective search uses an algorithm created by Felzenszwalb & Huttenlocher (2004) that segments the image based on comparing pixels to find edges and grouping similar pixels. The grouping is recursively done using a hierarchical algorithm that groups similar regions into larger regions, judging them on size and texture. From this, a set of boxes can be drawn around the segments that may contain an object (Felzenszwalb & Huttenlocher, 2004). This is shown in figure 6 below.

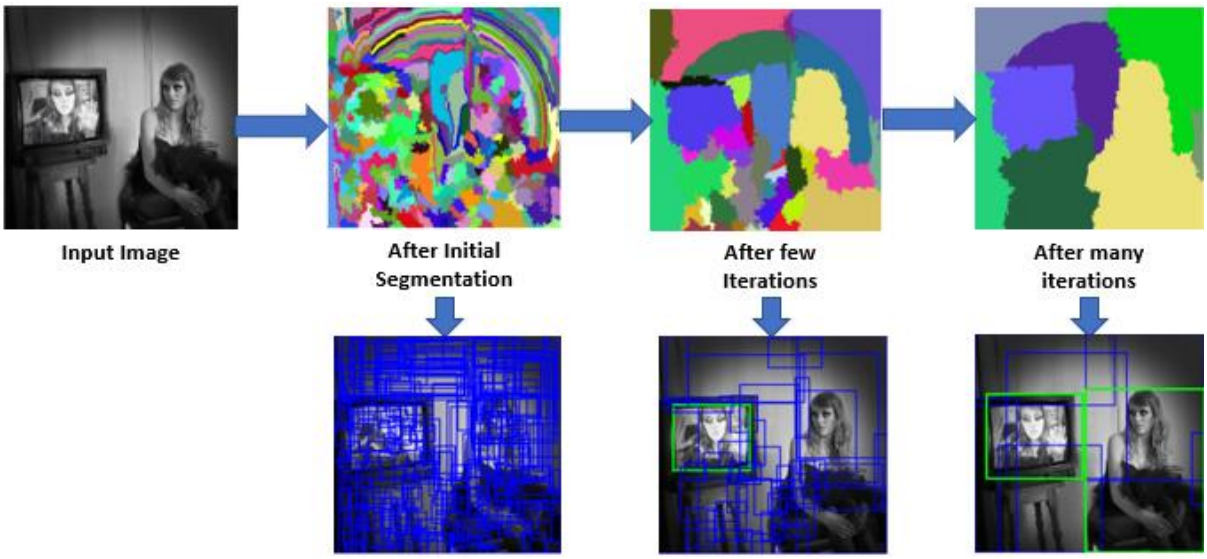


Figure 6: An example of a selection search used on an image to produce region proposals.

The region proposals are all warped and resized to fit as input for a CNN. The output of the CNN is a 4096-dimension vector of features. The CNN used for R-CNN has five convolutional layers and 2 fully connected layers. In the final phase, the feature vectors are given a value using a support vector machine (SVM) that has been trained for that class (Girshick, et al., 2014). An SVM is an algorithm that has been trained to classify data and output as a specific category of object.

3.1.1 Fast R-CNN

As mentioned in the background, R-CNN was improved to develop Fast R-CNN and Faster R-CNN. Fast R-CNN improves R-CNN by taking the entire image and the proposals as input for the feature extractor CNN in which the whole image is processed with multiple convolutional and pooling layers to create a convolutional feature map. Using this feature map, a region of interest pooling layer extracts a feature vector. This feature vector is processed by a sequence of fully connected layers and then two output layers to estimate the probability value and the four values need to show the bounding box (Girshick, 2015).

3.1.2 Faster R-CNN

Faster R-CNN replaces the selection search algorithm with a region proposal network (RPN) which is a CNN that creates region proposals with a range of aspect ratios and scales. The RPN shares a set of convolutional layers with the detection CNN and uses different sized anchors to generate various sizes of proposals. These anchor boxes have 3 different scales and aspect ratios, thus using 9 anchor boxes to generate the region proposals (Ren, et al., 2017).

3.2 YOLO

You Only Look Once (YOLO) is a method of object detection created by Redmon, et al. (2016) that uses a single neural network to predict bounding boxes and class probabilities in one single evaluation. This is considered an end-to-end method of object detection and therefore can be optimized at any stage of the detection pipeline. To do this, the input image is divided into grids of a predefined size. The grid cell that the object's centre falls into is then associated with that object. It produces prediction bounding boxes from the grid associated with each object and the confidence scores for these boxes. YOLO does this for all classes of objects in the image at once and the confidence score determines how sure the model is that the prediction box contains an object. 5 values are stored in each bounding box. The first two are the *x* and *y* coordinates of the centre of the box relative to the associated grid cell. The height and width values are also stored but are relative to the image. Finally, a separate confidence value to the class confidence is stored alongside these values. This confidence value evaluates how accurate the bounding box is in comparison to the true original bounding box (Redmon, et al., 2016). This overview of YOLO is visually represented in figure 7 below. These values are stored together as a prediction tensor of size S x S x (B*5+C) where S is the number of grid cells, B is the number of bounding boxes per grid cell and C is the number of object classes to classify between (Redmon, et al., 2016).

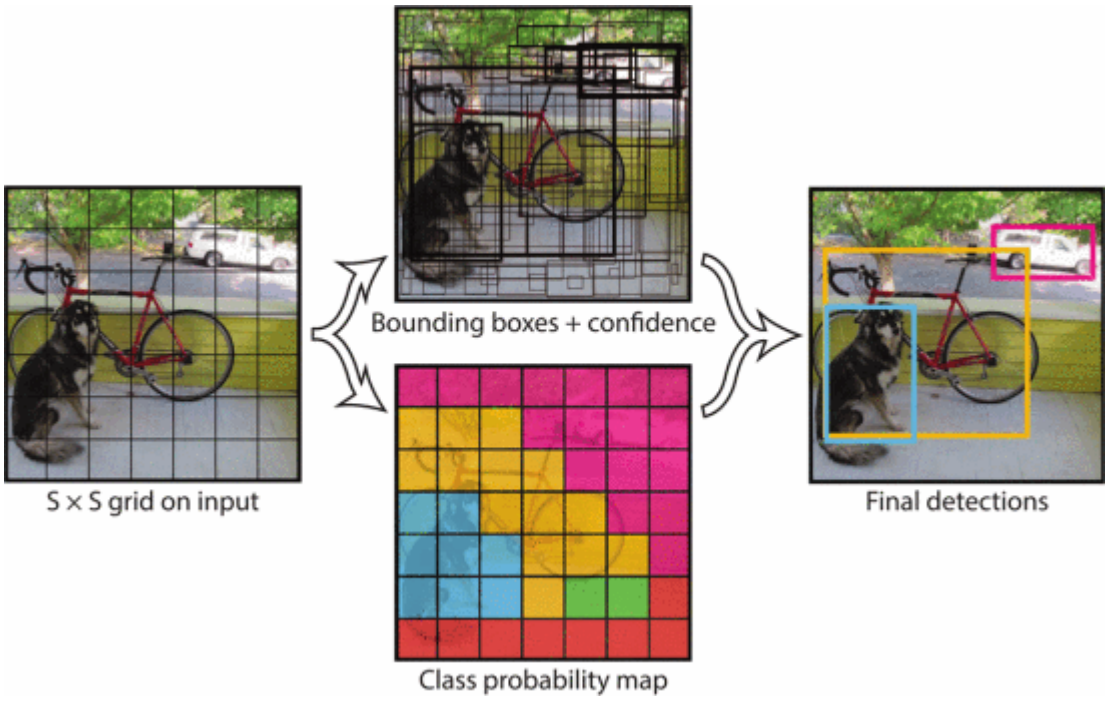


Figure 7: Visual representation of the YOLO pipeline.

3.2.1 YOLOv2

YOLOv2 (or YOLO9000 as its sometimes referred to) is the successor to YOLO with significant changes to improve the recall and decrease the number of localization errors. The first change was to implement batch normalization into all the convolutional layers (Redmon & Farhadi, 2017). Batch normalization calculates the mean (μ) and the variance (σ^2) of the activation values. It then normalizes the activation vector of values ($Z\hat{}$) so that each node's output is standardised across the normal distribution on the batch. Finally, a linear transformation is applied with two parameters to adjust the standard deviation and the bias (Huber, 2020).

Another major change made was to use a high-resolution classifier. Previously in YOLO, the classifier was trained on images of a 224 x 224 resolution and then increase to 448 x 448 for detection. However, YOLOv2 trains the classifier on images of 448 x 448 resolution which allows the network to modify the filters to be more accurate on high-resolution images. On top of this change, YOLOv2 takes inspiration from Faster R-CNN and uses anchor boxes instead of the fully connected layer to predict bounding boxes. This means that YOLOv2 predicts over a thousand boxes per image in comparison to the 98 boxes that YOLO predicts. It has led to a slight decrease in accuracy but a higher recall rate which means that the model "has more room to improve" (Redmon & Farhadi, 2017).

A new classification model called Darknet-19 is used in YOLOv2 to classify objects. It is similar to the VGG model and the model that was previously used in YOLO in that it uses 3 x 3 filters and after every pooling stage the number of channels is doubled. This model has 19 convolutional layers and 5 max-pooling layers. It has a higher accuracy of 91.2% in comparison to 88% for the YOLO classifier and 90% for the VGG classifier (Redmon & Farhadi, 2017).

The final difference between YOLOv2 and its predecessor is that it has been trained using a variety of input resolutions so that it can be a viable detector for a variety of image resolutions. The difference in the detections of the different resolutions is evident in table 2 below in which the accuracy and speed are compared to other models. This table shows that increasing the resolution will increase the accuracy but decrease the speed of detection (Redmon & Farhadi, 2017).

Table 2: Table showing the speed and accuracy of YOLOv2 trained on different resolutions and other models.

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 288 × 288	2007+2012	69.0	91
YOLOv2 352 × 352	2007+2012	73.7	81
YOLOv2 416 × 416	2007+2012	76.8	67
YOLOv2 480 × 480	2007+2012	77.8	59
YOLOv2 544 × 544	2007+2012	**78.6**	40

3.2.2 YOLOv3

YOLOv3 is the successor to YOLOv2 and again has been modified to improve both speed and accuracy. One of the ways in which this has been done is by changing the way the classes are predicted. Previously the SoftMax algorithm was used to determine what classes may be contained in the bounding box but this is changed to a multilabel approach. The multilabel approach uses independent logistic classifiers such as binary cross-entropy loss which is used in the training stage (Redmon & Farhadi, 2018).

Binary cross-entropy is calculated using the average of the log of corrected probabilities. Corrected probabilities are the predicted probability that the image contains an object from its original class. Because the probabilities are between 0 and 1 the log will always be negative, so the average of the logs must be multiplied by -1 to get the loss value (Saxena, 2021). The use of binary cross-entropy improves the classification when there are multiple overlapping objects as SoftMax assumes “that each box has exactly one class which is often not the case” (Redmon & Farhadi, 2018).

The feature extractor used in YOLOv2; Darknet-19 is improved in YOLOv3 by making it have 53 convolutional layers instead of the 19 layers. When compared to the previous model and the ResNet-101 and 152 models, the accuracy is improved to 93.8% but runs a lot slower than Darknet-19 but still faster than the ResNet models. This is illustrated in table 3 below (Redmon & Farhadi, 2018).

Table 3: Accuracy and speed of the 2 Darknet and ResNet models used for feature extraction

Backbone	Top-1	Top-5	Bn Ops	BFLOP/s	FPS
Darknet-19 [15]	74.1	91.8	7.29	1246	**171**
ResNet-101[5]	77.1	93.7	19.7	1039	53
ResNet-152 [5]	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

3.3 Comparison of accuracy

The accuracy of object detectors is measured using the mean average precision (mAP). To get the mean average precision, the average precision (AP) is calculated for each image. To calculate the AP, the precision and recall need to be calculated. Precision is a way of calculating how accurate a prediction is by dividing the running total of true positives (TPs) by the running total number of TPs and false positives (FPs). The recall is the percentage of the found true positives out of all the possible true positives in the top N predictions. This is calculated by dividing the running total of TPs by the total of TPs and false negatives (FNs). The mathematical definitions are shown in figure 8 below (Hui, 2018).

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

TP = True positive

TN = True negative

FP = False positive

FN = False negative

Figure 8: Mathematical definitions for precision and recall.

To determine whether a prediction is correct, the Intersection Over Union (IoU) needs to be above 0.5. IoU is the comparison of the pre-defined bounding box (BB) and the prediction BB. It is calculated by dividing the area of overlap between the BBs and the area of union between the two BBs. This is illustrated in figure 9 below. The running average precision is plotted against the corresponding recall value in a graph similar to that in figure 10 below. Finally, the area under the curve is calculated to get the AP (Hui, 2018).

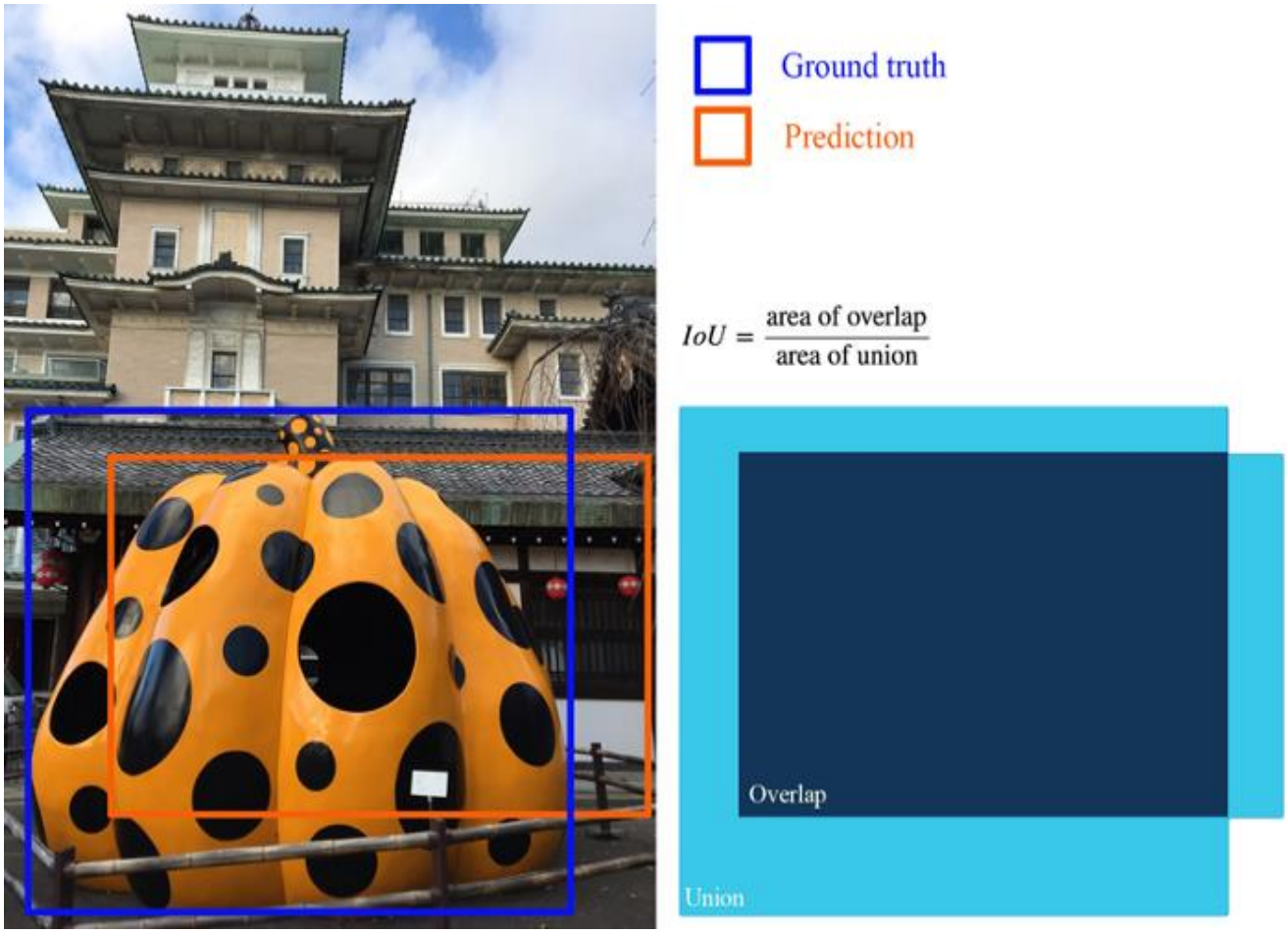


Figure 9: Example of calculating IoU for object detection

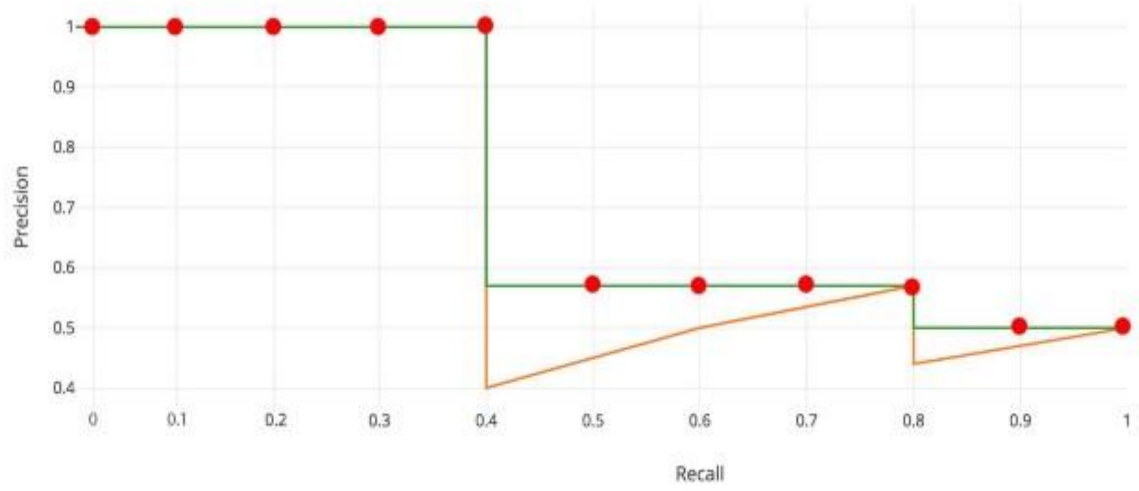


Figure 10: Example graph of the precision vs recall value for the best predictions.

YOLOv2 can have an mAP of up to 78.6 and Faster R-CNN can reach up to 76.4 mAP which is not a significant difference in accuracy. Juan Du (2018) stated that the reasons why YOLOv2 has a high accuracy are because it uses global features and creates context clues to help find

the objects. This results in a lower chance of predicting false positives in the background of the image. On the other hand, R-CNN creates many background false positives. YOLOv2 also learns the representations of objects which makes it a lot more accurate at detecting objects in a wide variety of contexts/mediums. For example, generalizing from natural images to man-made images such as artwork (Du, 2018).

YOLOv2 has a lower accuracy at detecting smaller objects that are in groups together such as detecting people from afar who are in a large crowd. YOLOv2 also falls short when generalizing to objects in uncommon aspect ratios and configurations (Du, 2018). This is also backed up by Ye, et al. (2021) that found YOLO to be less accurate in extreme scenarios such as detecting small objects (Ye, et al., 2021).

In a 2021 survey completed by Hakim & Fadhil (2021), they state that YOLO has an mAP of 63.4 which is significantly less accurate than Faster R-CNN. This is contrary to the survey previously mentioned by Du (2018) who stated the precision of YOLO is higher than Faster R-CNN with an mAP of 78.6. Even though the 2021 survey was completed more recently than the previous survey, Hakim & Fadhil (2021) only look at YOLO which is the predecessor to YOLOv2 (Hakim & Fadhil, 2021). Therefore, it is worth considering that both R-CNN and YOLO have multiple models with each successive model attempting to improve on the previous model. This means that when comparing which model to use, it is worth considering the iteration of that model that was used. Because of this, the changes made in the latter version of the models will need to be looked at when trying to make the model more accurate.

3.4 Comparison of speed

The speed of object detection can be crucial in deciding which model to use because often, 100s of images may be needed to be passed through at once. Also, speed is important when using these models on videos and to be used in real-time. Typically, the speed of object detection is measured in frames per second (FPS). However, there is a trade-off on accuracy when trying to get a model to detect faster.

The speed of R-CNN was increased in the later iterations, Fast R-CNN and Faster R-CNN. R-CNN predicts over 1000 proposal bounding boxes and most of these boxes overlap. Because most of the prediction bounding boxes overlap, a lot of these boxes are unnecessary which makes this model inefficient for object detection when looking at the speed (Wang, et al.,

2019). The speed was increased by replacing the selective search with a region proposal network that uses anchor boxes to produce prediction bounding boxes. As a result of this, Faster R-CNN can reach speeds of 18 FPS. Even though this sped up the model, Faster R-CNN is still not fast enough for real-time but fast enough for video detection.

On the other hand, YOLO only predicts 100 bounding boxes per image which dramatically increases the speed of prediction and also produces fewer unnecessary prediction bounding boxes. As a result of this, YOLO can reach speeds of up to 155 FPS for Fast YOLO, 91 FPS for YOLOv2 at 288x288 resolution and up to 45 FPS for YOLOv3 at 320x320 resolution (Redmon & Farhadi, 2017) (Redmon & Farhadi, 2018). All these speeds are faster than Faster R-CNN because YOLO doesn't use an RPN to predict bounding boxes and instead divides the image into grids and uses 5 anchor boxes instead of the 9 anchor boxes that R-CNN uses (Wang, et al., 2019).

Taking the speeds into account, YOLO is the faster model and therefore will be a better choice if speed is necessary.

3.5 Comparison of utility

The final factor that determines the best model to use for object detection is the utility that each model is used. By looking at the usage of each model in different scenarios, the models can be compared based on which model will be best suited for this project. To do this, similar scenarios to detecting objects from a helicopter will need to be looked at to find out which model is best at detecting objects from afar.

In a study of using object detection for detecting roofs from above, Musyarofah et al (2019) found that using Mask R-CNN resulted in an mAP of 90.14%. Mask R-CNN is a predecessor to Faster R-CNN that builds upon it by adding a small fully convolutional network (FCN) that predicts segmentation masks on the RoI of the objects. This FCN works alongside the classification and bounding box predictions to produce an output similar to that in figure 11 below. These segmentation masks are computed using the output of the RoI classifier and provide a mask represented by floating-point numbers instead of binary numbers. These masks are then used to compute the loss in a more accurate way than using binary masks (Musyarofah, et al., 2019).

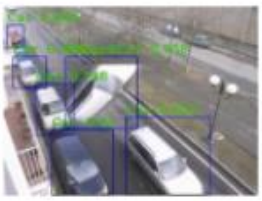
Figure 11: Illustration of the masks shown on roofs

This study uses top-down images from UAVs which is very similar to the usage of a detector for helicopter cameras. Therefore this study shows that the use of Mask R-CNN is suitable for detecting objects from afar and therefore would be good at detecting objects from a helicopter camera. The issue with this study is that only images were fed into the object detector and there was no mention of the speed of the Mask R-CNN used. Even though the accuracy of Mask R-CNN in this scenario was high at 90.14%, this study does not test it against videos and therefore the accuracy of real-time detection is unknown. This means that it may not be suitable for real-time usage but is highly accurate for detecting objects in the sky on images.

A study that looks at real-time object detection completed by Chen et al. (2021) used YOLOv3 as a base to build upon to detect objects in a database of static drone images. Even though the object detector was tested against images, it was deduced that YOLOv3 can be used in real-time at 30 FPS due to the low inference time (Chen, et al., 2021). Because drone images were used in this study the results will likely be similar to that of a helicopter camera since they are both airborne situations and require the detection of objects from a large distance. This means that YOLOv3 will be a suitable base architecture for real-time detection based on the results of this study.

Another study that looks at real-time object detection is a study done by Raskar & Shah (2021) that looks at object detection to detect forged frames in videos. In order to meet the real-time threshold for videos YOLOv2 was used. To detect forged frames, the objects have been rotated, flipped or scaled up and the object detector must detect the objects that may have been manipulated in one of these ways. The results of one of the videos used is shown in table 4 below in which the confidence score ranged from 0.95 to 0.99 and the FPS stayed consistent at 30 FPS (Raskar & Shah, 2021).

Table 4: Video Forgery detection results from a sample video

Type of attack	Output: forgery detected frame	Confidence score	Total no. of frames	Frame rate (FPS)	Total no. of frames forged
Scaling 1 (Scale-Up)		0.95	557	30	16
Flip 1		0.99	557	30	17
Flip2		0.95	557	30	18
Rotate 1		0.99	557	30	18
Rotate2		0.97	557	30	21

The results for all tests followed the example test given above the confidence scores ranged from 0.95 to 0.99 for all tests and it is suggested that YOLOv2 is good at detecting "small, occluded stationary and moving objects with proper classification and localization" (Raskar & Shah, 2021). This is good for detecting objects from a helicopter camera because the objects are likely to be far away and therefore are smaller on the camera. It is also suitable for this project's scenario because objects will need to be detected in all weather conditions and as a result may be occluded by precipitation which can be detected using YOLOv2. On top of the accuracy, the fact that YOLOv2 can do this in 30 FPS shows that it is viable for detecting objects in real-time and this is corroborated by the previous study on static drone image detection which also showed the suitability of YOLO based models at real-time object detection.

3.6 Chosen Model

After looking at the differences between R-CNN and YOLO based models, the base architecture that will be used in this project is YOLO. This is because YOLO is a one-stage detector which makes the models a lot faster than R-CNN and therefore is more suited to real-time object detection. Real-time object detection is needed for this project because the objects will need to be detected whilst the helicopter is in flight. On top of using YOLO as a base, techniques such as batch normalization and using the Darknet classifier were used in the later iterations of YOLO to make the detection more accurate.

4.0 Research Methodology

4.1 Waterfall Method

There are a variety of methodologies that can be used in project management to help produce a product. A traditional approach to project management is to use the Waterfall Method. The Waterfall Method is a plan-driven method which has a heavy focus on the planning stages of project management and thus requires good communication from the clients on what they expect from the project. As a result, the Waterfall Method creates a strong structure with the project documented and planned from start to finish. This will likely consist of deadlines, responsibilities and work packages (Thesing, et al., 2021). The waterfall process is heavily based on planning and thus the planning helps with the understanding and the implementation of the project to produce a better result (Rubin, 2013). An issue with the Waterfall Method is that if an issue were to occur throughout the development phase, then it is hard to make the necessary adjustments and therefore will make it harder to meet a working product (Evelin, et al., 2021).

4.2 Agile Methods

Agile methods are based on flexibility and using an iterative approach that refines the early stages so that it makes the development stages more efficient and reduces the need for customer interaction. Often, the development stage is split into teams that work together on a specific aspect of the project. The main components of Agile methods are a continuous design, flexible scope and freezing specifications (Serrador & Pinto, 2015). An advantage of agile methods over the traditional methods is that having smaller teams will have more focus on the specific aspects of the project, but this leads to issues with the management where these separate teams are required to coordinate (Petersen & Wohlin, 2009).

4.2.1 Kanban Method

One of the agile methodologies is the Kanban Method. It has a large focus on what tasks need to be done and when will they need to be done. As stated by (Lei, et al., 2017), Kanban does this by “prioritizing tasks, and defining workflow as well as lead-time to delivery”. This means that tasks are sorted on the importance and therefore emphasises these tasks to make sure

they are completed in time. Kanban is generally visualized using cards arranged in categories such as backlog, work in progress (WIP) and completed. The main idea is to reduce the number of cards in the WIP category and so only the required features are implemented, and the specifications are tailored to what can be completed in the allotted time.

4.2.2 Scrum Method

Another agile methodology is the Scrum Methodology. This method consists of iterations called sprints that last for a month. Similar to that of Kanban, work that needs to be done is put in the backlog and then prioritizes the tasks at the start of the sprint period. Once the tasks have been prioritized, they are moved to the sprint backlog. Daily meetings are held throughout the sprint to modify the scope of these tasks and sort out any issues that have occurred. Once the sprint period has ended, the work produced during that sprint is demonstrated and reviewed to determine whether that sprint work is added to the larger project for the release to the customer (Metherall, et al., 2007). In a scenario in which Scrum was used, it was easier to find and deal with errors earlier on in the pipeline as a result of the daily meetings that the Scrum teams had (Rubin, 2013). With the Scrum method focusing heavily on shorter iterations, the objectives to achieve the final product may become unclear (Thesing, et al., 2021). This can lead to a different product from what was initially conceived by the customer.

4.2.3 Extreme Programming Method

An alternative Agile method is Extreme Programming (XP). It is described by (Erickson, et al., 2005) as “the coding of what the customer specifies, and the testing of that code to ensure that the prior steps in the development process have accomplished what the developers intended”. This means that XP is based heavily on time management and thus reduces the chance of additional features being added so that the customers receive the product on time. However, features can be added in the latter stages of development if specified by the customer once the base features have been implemented. This is in contrast to most other methodologies since the norm is to specify all features before development has started and

thus these features are dynamic using XP. A reason to use XP over other agile methods is that XP has a large emphasis on reducing the workload by only implementing the features necessary for the customer. As mentioned before, this helps to produce the product in time for the deadline (Erickson, et al., 2005). Karlsson, et al. (2000) implemented XP and found that even though they benefitted from the use of XP, the implementation of it "proved quite challenging".

4.3 Chosen Methodology

Kanban is the most appropriate methodology for this project because it allows the project to be split up into smaller tasks and a Kanban board is a good visual aid to help organise these tasks. Since the project is made under a short time constraint, the tasks can be prioritised to ensure the tasks are made based on what can be completed in the allotted time. The waterfall process is not used in this project because this method is not iterative such as the agile methods and this project requires multiple iterations because multiple models will need to be trained and tested to create the best model for object detection. The reason why the Scrum method is not used is that it is heavily based on working in sprint groups and a lot of the features will be harder to translate for a solo project. XP was not chosen simply because this project will not require additional features beyond the initial project scope and therefore continuous testing and implementation will not be necessary.

5.0 Project Plan

The plan for the project can be found in the Gantt Chart (see appendix 1). The research of the neural networks is planned to take ten weeks because the history of object detection and the analysis of the most suitable methods of object detection will need to be looked at. The analysis is planned to take five out of the ten weeks because the methods will need to be described in detail and compared with each other to find the most suitable method of object detection. The design is expected to take five weeks because every aspect of the artefact will need to be decided upon and will have a crossover with the implementation because the iterative design process will be used in conjunction with the Kanban method so the artefact will be designed, implemented, tested and then re-designed. The implementation is planned

to take ten weeks because that allows the maximum amount of time to work on the artefact whilst allowing enough time for the artefact to be tested and evaluated.

6.0 Design

6.1 Scope

The artefact for this project will be the object detector that will be used to detect objects from a helicopter camera. This can be broken down into these parts:

- Parsing of images.
- Creation of the convolutional neural network.
- Training of the convolutional neural network.
- Testing of the model in an engine environment.

The parsing of images is straightforward and can be done using a library so that the image data can be fed into the neural network as input data. The creation and training of the CNN can either be done by creating the library and the model from scratch or by using a pre-made model and training it using custom data. Using a pre-made model will be possible to train and test to get results within the time given but will not be an adequate amount of work to discuss the implementation and evaluate it as an artefact.

Therefore, for this project, the artefact will be a CNN library created from scratch to create a model that will be able to detect objects. This will prove to be a challenge to do in the given time frame. Consequently, the model will not be at a finished stage and therefore it will not be in a state for it to be tested in an engine environment. The artefact will need to be tested using other means and that will be another aspect of the artefact that will have to be designed. To decide how to go about the CNN library, the different models that are currently used for object detection will need to be analysed to find the best approach for the artefact.

6.2 Hardware and Software Needed:

- Computer/Laptop with a Graphics Processing Unit (GPU) – This is so that the training of the model can be done using a GPU which will be a lot faster than using a Central Processing Unit (CPU).
- C++ using OpenNN library – C++ is faster than alternative programming and allows for memory management. OpenNN is an open-source neural network library that will be used to parse the images as input data. OpenNN will not be used to create the model (Artelnics, 2021).

6.3 Artefact Design

6.3.1 Iterative Design Process

The iterative design process will be used alongside the Kanban methodology to design and implement the artefact. The iterative design process is a cyclic process that will cycle after a series of fixed-length smaller projects. During a cycle, the small project is designed, implemented, tested and evaluated and then added to the main project. The idea is that over the many cycles, the main project gets bigger and more adapted. The design is also adapted after these cycles as the project grows and evolves (Larman, 2005).

One of the reasons why it will be used for this project is that the scope of the project is too large for the given time frame. Therefore, testing the project in the originally planned way will not be possible if there is not a complete artefact. So, if the project cannot be tested as a whole, each subsection needs to be tested to prove that the artefact works well with what has been made so far. The testing part of the iterative design process can be used to test each part of the project without needing the project to be finished before testing.

Another reason for using the iterative design process with this project is that it will help choose which parts of the project to work on next. With the scope being too big for the current time frame, the whole project will not be finished in time and as a result, the tasks to get there will have to be prioritized based upon which can produce an unfinished working

artefact. The iterative design process helps to show early visible progress so that the tasks to do first will be prioritized by which ones show the most progress and therefore an artefact can be shown regardless of if it is not finished in time (Larman, 2005).

This project will be using the iterative design process but without the concept of cycles. This is because the time frame is too short to break down into smaller cycles and therefore will be used alongside Kanban. Instead of using cycles, the project will be broken down into smaller tasks on the Kanban board and each task will have to be designed, implemented and tested before considering that task done and moving on to the next task.

6.3.2 Neural Network Library Design

As previously mentioned, the ideal object detector to make is the YOLO model. This uses the Darknet neural network library which has been used as the inspiration for this artefact (Bochkovskiy, et al., 2020).

The initial design of the neural network class is shown in the UML class diagram in figure 12 below (Appendix 3).

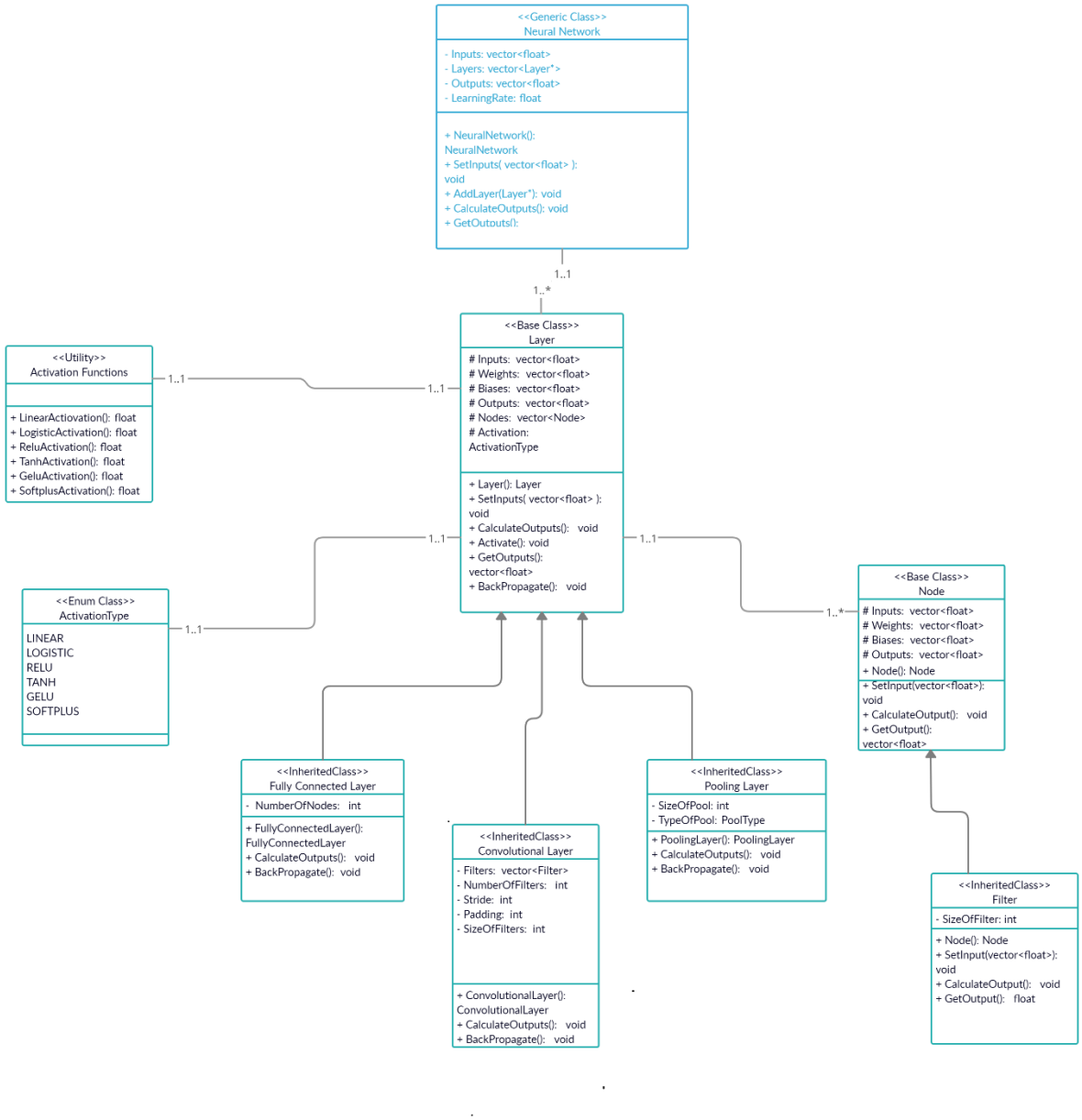


Figure 12: UML Class Diagram

The overall structure is a neural network class that holds an array of layers that each hold an array of nodes. The maths logic is handled by the nodes and the tying together of the nodes is handled by the layers and the passing of inputs/outputs between layers is handled by the neural network class. This is done to allow a more dynamic creation and modification of layers and the network to be able to combine a multitude of different layer combinations. For example, figure 13 below shows the network architecture of YOLOv2 which consists of multiple different layers with a variety of filter and output sizes.

YOLO V2 network architecture.

Layer description	Kernel size	Layer output
Input to network		288 × 288 × 3
Convolutional layer1	3 × 3	288 × 288 × 32
Max Pooling layer	2 × 2	144 × 144 × 32
Convolutional layer2	3 × 3	144 × 144 × 64
Max Pooling layer 2	2 × 2	72 × 72 × 64
Convolutional layer3	3 × 3	72 × 72 × 128
Convolutional layer4	1 × 1	72 × 72 × 64
Convolutional layer5	3 × 3	72 × 72 × 128
Max Pooling layer 3	2 × 2	36 × 36 × 128
Convolutional layer6	3 × 3	36 × 36 × 256
Convolutional layer7	1 × 1	36 × 36 × 128
Convolutional layer8	3 × 3	36 × 36 × 256
Max Pooling layer 4	2 × 2	18 × 18 × 256
Convolutional layer9	3 × 3	18 × 18 × 512
Convolutional layer10	1 × 1	18 × 18 × 256
Convolutional layer11	3 × 3	18 × 18 × 512
Convolutional layer12	1 × 1	18 × 18 × 256
Convolutional layer13	3 × 3	18 × 18 × 512
Max Pooling layer 5	2 × 2	9 × 9 × 512
Convolutional layer14	3 × 3	9 × 9 × 1024
Convolutional layer15	1 × 1	9 × 9 × 512
Convolutional layer16	3 × 3	9 × 9 × 1024
Convolutional layer17	1 × 1	9 × 9 × 512
Convolutional layer18	3 × 3	9 × 9 × 1024
Convolutional layer19	3 × 3	9 × 9 × 1024
Convolutional layer20	3 × 3	9 × 9 × 1024
Concat		18 × 18 × 512
Convolutional layer21	1 × 1	18 × 18 × 64
Flattened	2 × 2	9 × 9 × 256
Contact		9 × 9 × 1280
Convolutional layer22	3 × 3	9 × 9 × 1024
Convolutional layer23	1 × 1	9 × 9 × 60 (Network Output)

Figure 13: YOLOv2 Network Architecture

The flow of the inputs and outputs is best shown in the sequence diagram as figure 14 below (Appendix 4). The inputs get passed into the network which then passes it to the first layer. The layer class will then set all the inputs of the nodes. Once the CalculateOutputs() function is called for the network, it is called on each layer starting with the first one. This layer will then calculate the outputs for each node, called the activation function and pass the outputs back to the network class. These outputs are set as the inputs of the next layer and so on until CaluclateOutputs() has been called on each layer. Finally, the cost is calculated and then the BackPropagation() function is called for each layer starting with the last layer to update the weights and biases.

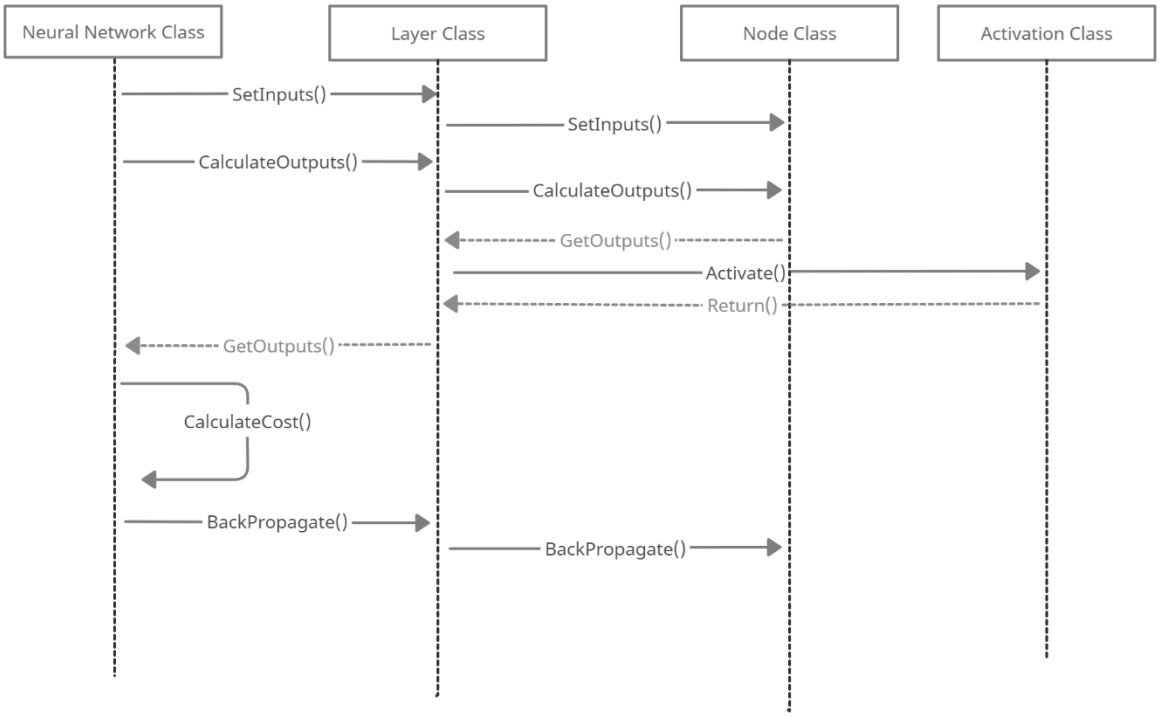


Figure 14: UML Sequence Diagram

The architecture has also been designed so that these different layers can be created separately and added to the network for the network to handle the passing off the output of one layer as the input of the next layer. On top of this, the layers all have different output sizes which is another reason why they are in vector arrays because the layer output sizes may not be pre-defined by the user and therefore will have to be dynamically increased/decreased based on the number of inputs, weights and biases.

Furthermore, the network is designed to store the layers as pointers in a vector array. Pointers are used here so that the classes that inherit from the layer class can be recognised as layers to be placed into the vector array. The main reason there is a base layer class and then the different layer types inherit from this class is because they all have the same functions, but they all work in different ways. For example, the method for calculating the output of a convolutional layer will be different to that of a pooling layer but they both need a calculate outputs function.

Darknet uses standard arrays for the storage of the inputs, weights and biases. This uses memcpy to modify the size of the arrays dynamically, but this artefact will not use standard arrays. Alternatively, the neural network class will consist of vector arrays instead of standard

arrays. A vector array has been chosen because it allows the use of dynamic memory resizing which means the network can be initialized and then the layers can be pushed into the network.

The handling of the activation functions is done using a utility class that defines a set of static inline functions. This is done in a similar way to Darknet because all the activation function needs to do is take in a float and return an output float. Using static functions in a helper class means that an instance of the activation functions does not need to be made since only one function will be needed at a time. The appropriate function will then be called by the layer classes.

6.3.3 Unit Testing

As explained in the scope, it is clear that the artefact is not possible to produce and test in the allotted time frame. Therefore, unit testing will be used to test the artefact. This is because unit testing will be able to test the artefact without it being fully finished and can test the different aspects of the neural network library to make sure that what has been produced will work.

A major advantage of unit testing is that it helps to identify bugs and errors in code so that they can be fixed (Minhas, 2021). It does this by breaking down the source code by testing specific functions/methods to determine whether they meet the expected behaviour. A well-designed unit testing system will test successful outcomes, edge cases and failure cases. This is to make sure all outcomes are covered and there are no undefined behaviours. This is useful for a neural network library because it will contain a lot of maths that will need to be tested with edge cases and with inputs that produce the expected output. For example, the activation functions can be tested to confirm the equations are correct.

Another advantage of the use of unit testing for debugging is that it can be used as part of the iterative design process to test that the developed feature works fully before moving on to the next feature. As a result, a robust artefact will be created with full confidence that there are little to no errors. This is a good method for developing a neural network library because features can only be added to the library once the current features are fully tested. In doing so it avoids developing a library that is too buggy and can only be used in a few situations.

The last major advantage of unit testing is that it helps to produce code with clear inputs and outputs. As a result of this, it makes the code more modular so that it can be built upon or integrated into other projects. One of the most important aspects of a library is that it needs to be modular so that it can be used for a variety of different projects/situations. The idea of creating a neural network library is for the user to be able to create different types of neural networks and not just one type of model. Unit tests allow the library to be tested for different use cases and means that it can be built upon to produce more advanced neural network models that will eventually help with the end goal of the project.

These are the reasons for using unit testing for the neural network library and due to the iterative design process and unit testing, the implementation of the artefact will likely turn out differently from the initial design.

7.0 Implementation

As stated in the design, the inspiration for the artefact of the neural network library comes from Darknet (Bochkovskiy, et al., 2020). Darknet is a neural network library specifically designed for object detection and was made for the creation of YOLO models. As found in the research, a YOLO type model would be best for the object detection of objects in real-time and therefore this is the reason for using Darknet as an example of the end goal of the artefact. However, the depth and complexity of Darknet are too high for the given time frame so only certain aspects have been influenced by Darknet's design. One of these aspects is how the activation functions are used.

7.1 Activation Functions

Activation functions are implemented at the end of a layer after the inputs have been processed using the weights/biases. The role of the activation function is to determine how much of a signal to pass to the next layer based on the outputs of the layer. The functions are typically non-linear and take in an input and produce an output after passing it through the activation function (Weidman, 2019).

The activation functions are all static inline functions. Using static inline before the functions tells the compiler that these functions should only be accessed within the activation function header and cpp files. There is also an activate function which takes an ACTIVATION enum and an input. This function is simply a switch state based upon the enum passed in that calls the corresponding activation function and returns the result. This is so that this function can be called by the layers and this function can then call the static inline functions. On top of this, there is an activate array function that takes in a vector array of inputs and returns a vector array of outputs after applying the softmax function.

The implementation of the activation functions only differs from the initial design in one aspect. As stated previously in the design, the layer classes will handle which activation function to call but this has been changed in the implementation to be handled by the activate function. This is because it is unnecessary duplication of code to have the switch statement in all the different layer classes.

There are seven different activation functions in the artefact. These were chosen based because they were used in Darknet and therefore are used in object detection. However,

Darknet has more than seven activation functions implemented. Due to the time constraint, it would not be feasible to implement all the activation functions. Consequently. the seven functions were chosen based on which are the most used activations in neural networks.

The process of implementing the activation functions consisted of looking up the mathematical equations and converting them to code. Since these equations are not too complicated, they were implemented easily. They were created as a class object so the activation type was passed into the constructor but that was changed to be a single activate function as mentioned earlier.

7.2 Layer Class

The layer class is implemented as a base class for all the different types of layers. This class consists of the variables and functions needed in most of the layers. The variables are the inputs, outputs, weights, and biases which are all vector arrays of float for the reason discussed earlier in the design. There are also variables for the number of nodes, activation type, layer type and sets of inputs.

This class also contains several virtual functions. The use of virtual functions allows them to be overridden by the inherited classes. On top of this, the inherited class can have these overridden functions with the same name but with different methods specific to the type of layer. For example, the calculate output's function can be overridden by the connected and convolutional layers but the way these functions will have different methods of calculating the outputs.

In addition to this, using layers classes that inherit from the base class means that the different layers can be stored in the network class as an array of layer objects. The use of overridden functions means that these functions can be called by the network class when iterating through this array of layers. The layer type variable is used to differentiate the different layer types for use when iterating through the array of layers. There are three different layer types.

As shown in the UML diagram for the initial design, the layer classes were planned to store a set of nodes which would handle the calculation of each node. This was changed when implementing the base layer class and the inherited classes so that the calculating of all the outputs are handled in the layer classes. The reason for this change is that the addition of a

node class was found to be an unnecessary abstraction and was more difficult to debug when the unit testing failed on the calculation of the outputs.

The filter class was implemented in a similar way to that in the initial design. This is because the weights are stored in the filters and therefore these filters can be applied to the inputs to produce the outputs. However, the implementation of the filter class differs from the design by only using the filters are a means of storing the weights instead of calculating the outputs using the filter class. The reason this was changed in the implementation is similar to the removal of the node class in that it was simpler to debug the process of calculating the outputs when it was all calculated in the layer classes.

There are additional functions added to the layer class which differ from the initial design. These include three functions to get the bias costs, weight costs and input costs which were added to retrieve the corresponding variables so that they can be used by the network class to pass the next layer and update the biases and weights once backpropagation has been completed on the whole network. Another additional function is the update weights and biases function. In the design, this function was planned to be done in the back propagation function, but this was implemented into a separate function because if there are multiple batches of inputs then backpropagation will need to occur for each input in the batch before the weights and biases are updated. More on how the backpropagation was implemented later in the section titled backpropagation.

7.2.1 Fully Connected Layer Class

The connected layer is a layer that consists of output nodes with the equation:

$$output = inputs * weights + bias.$$

This is the most used layer in neural networks and is the final layer in a YOLO based object detector. This is because it is typically used for the classification of objects in an image. A visualization of a fully connected layer is shown below in figure 15 (Ramsundar & Zadeh, 2018).

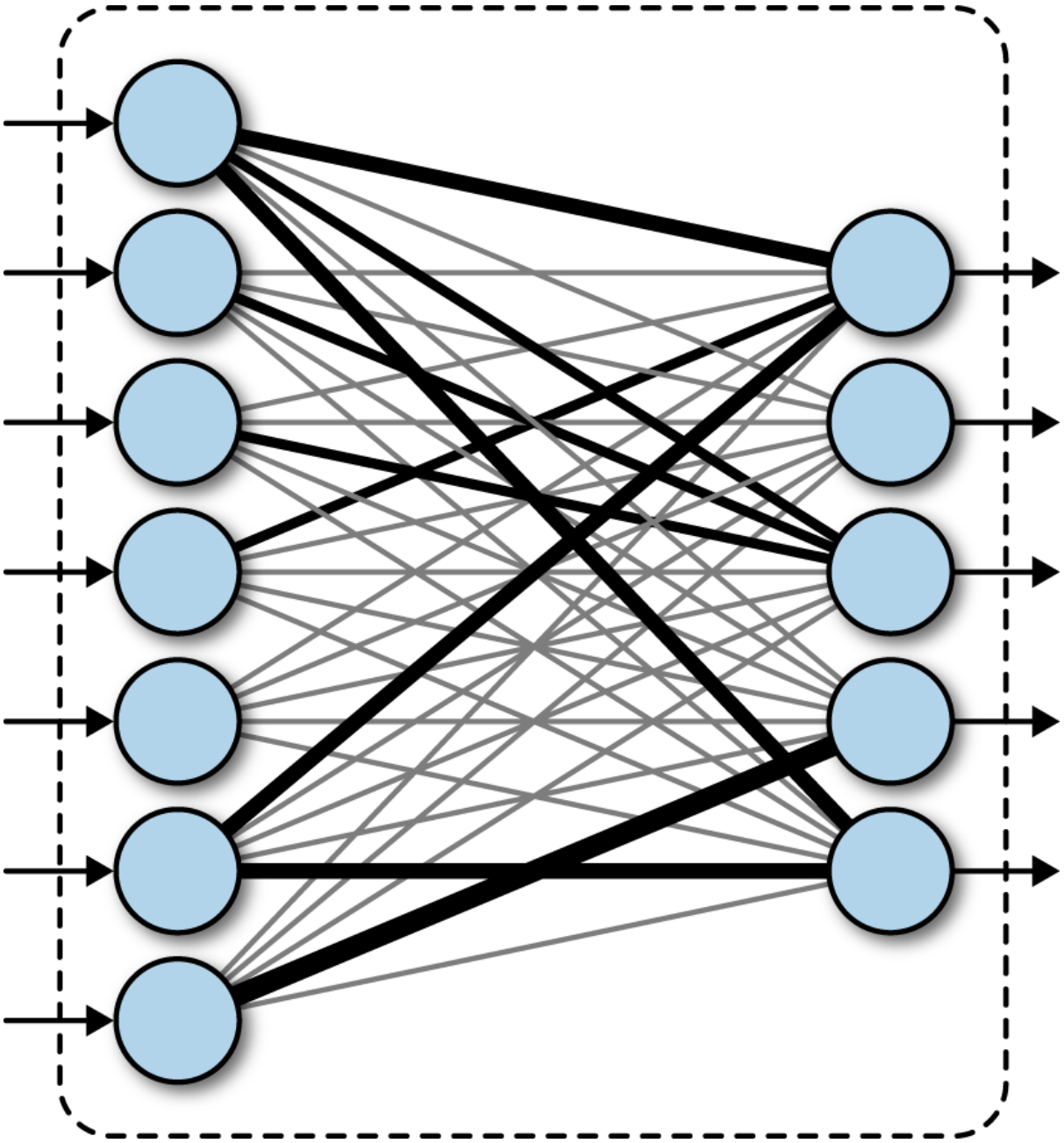


Figure 15: Visualization of a Connected Layer

The constructor for the connected layer class requires the inputs, weights, biases, number of output nodes and activation type to be passed in. The corresponding class variables are set to these values passed in.

The calculate outputs function is an overridden class which iterates for each output node to calculate the output for that node. Firstly, the inputs are iterated through to multiply the input with its corresponding weight. Once this has been done, the bias for that node is added and finally the activate function is called to get the final output for that node. These output values are pushed into the vector array of outputs.

There is a set inputs function which takes in a vector array of floats and sets the inputs for that layer. This has been implemented for neural networks that have multiple layers so that when the outputs are calculated for the previous layer then they can be set as the inputs of the next layer. The set inputs function allows the handling of outputs/inputs to be controlled by the network class.

The implementation differs from the initial design because there was an additional function implemented called calculate input costs. This function calculates the input costs for the layer so that they can be passed into the next layer for backpropagation. This function is only called at the end of the back propagation function so it can be included in that function as planned in the design, but it has been separated to differentiate the calculation of the weight costs and the input costs.

The process of implementing these functions for the connected layer class was not too difficult and the logic behind calculating the outputs is not difficult to implement in code. An aspect of the implementation of the fully connected layer that is done well is that there are no constraints on the number of nodes/weights/inputs that can be in the layer. This means that a variety of different configurations of the layer can be used so that the library can be used to create networks for a multitude of different purposes.

However, there was an issue when calculating the outputs which were indexing through the weights so that the correct input was multiplied by the correct weight. That's why the number of weights per node variable is calculated to find the correct index using the equation:

$$Index = weight\ index + (node\ index * number\ of\ weights\ per\ node).$$

7.2.2 Convolutional Layer Class

The second type of layer is called the convolutional layer. This layer uses filters to find specific shapes and lines within an image that is passed through. Filters are 3x3 or 5x5 matrices that act as the weights for the layer. All the values in the filter are multiplied by the corresponding values in the image and added together to get an output. This filter will then move along by a stride length across the whole image to get an output image (Brownlee, 2019).

For images with colour, the input image will have three channels (RGB) and therefore the filters will also have three channels. When calculating the output for multiple channels, the process is the same and applied to each channel except the outputs for each channel are added together to get an output image with one channel. An example of this is shown in figure 16 below (Saha, 2018).

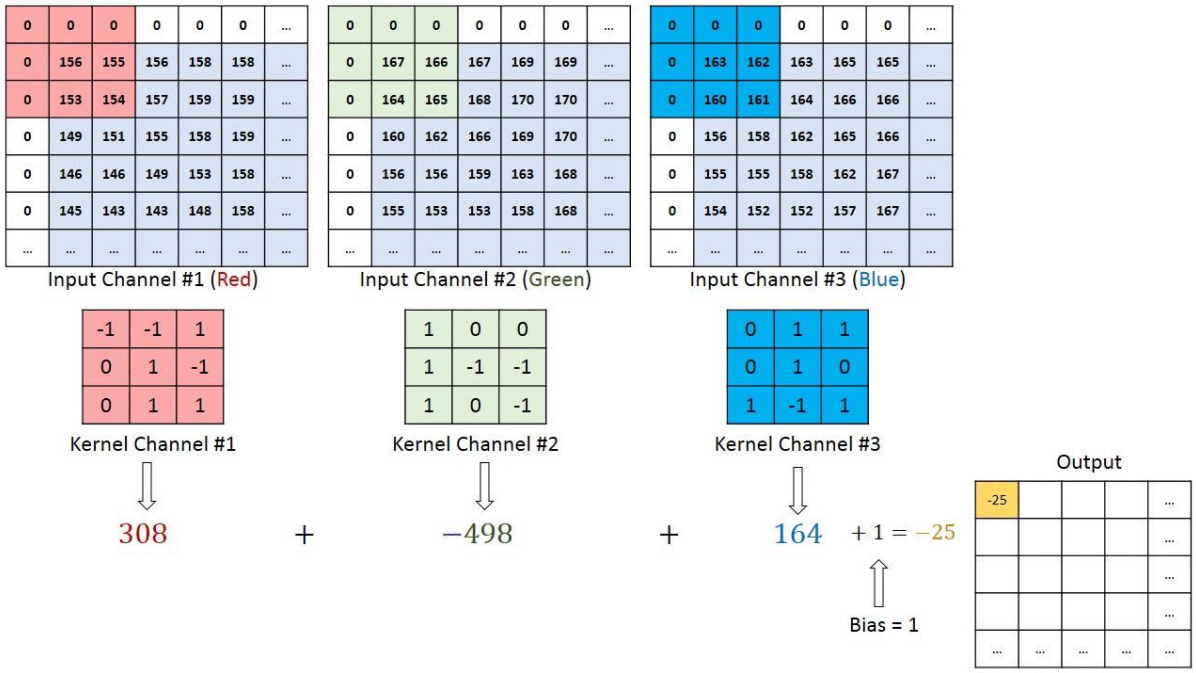


Figure 16: Visualization of a 3D convolutional layer

Another aspect of convolutional layers that is also present in figure 16 is the row and column of the value 0. This is referred to as padding and is often used to modify the size of the output image. For example, if the input is 5x5 and the filter is 3x3 with a stride of 1 this results in an output image of 3x3. If padding is added to the input image, then the output image size can be increased. There are two types of padding: valid padding and same padding. Valid padding is when a single row and column of 0s is added to the start of the image to get an output image with a size smaller than the inputs image. Same padding adds 0s all around the edges of the image to get an output image of the same size as the input. If valid padding was applied to the above example, then the output image will be of size 4x4 but if same padding was applied then the size will be 5x5 (Saha, 2018).

The convolutional layer class is another inherited class from the layer class. This class overrides the calculate outputs, set inputs and get outputs functions from the base class. There is an additional function called get filters which returns the filters for the layer.

Alongside the inherited variables the following class integer variables have been added: input height, input width, number of channels, number of filters, stride height and width. Also, two 3D vectors array has been implemented for the input and output image. These have been implemented in a three-dimensional (3D) vector array of floats because as shown previously, the theory has the filter moving across a multidimensional image. Therefore, converting a one-dimensional (1D) array of inputs into a 3D array helps to visualize the calculation of the outputs.

A filter struct was implemented to store all the variables needed for each filter. The variables are the height, width, number of channels and a 3D vector array of floats to store the values of the filter. This has been implemented in a struct because a convolutional layer will likely have multiple filters of different sizes and values. The filter struct has a constructor that takes in the height, width, channels and values as a vector array of floats. The values for the filters are then looped through to push these values into the 3D vector array. A 3D vector array is used because the input images are likely to be an image with multiple channels so converting the filters into 3D vector arrays makes the calculation of the outputs easier the visualize from the code if the inputs and filters are in the same dimensions.

The constructor for the convolutional layer takes in the input height, width, channels, a vector array of filters, stride height, width, a boolean for whether to add padding or not and the activation type. The corresponding class variables are set in the constructor and the filters are pushed into the array of filters. The reason these are pushed individually instead of setting the array is so that each filter can be checked using an assertion to make sure the number of channels in each filter is the same as the input. Even though the filter can be of different sizes, they all must have the same number of channels as the input image. As well as this, the values are pushed onto the weights array so that the number of weights can be retrieved by the network class.

The set inputs function takes in a 1D array of floats and converts this into a 3D array of floats referred to as the input image. Before the conversion, there is an assertion to check the input height, width and channels are consistent with the size of the array passed in. For the parsing of the inputs, two local variables called input channel and input row are created. The input row is a 1D vector array and the input channel is a two-dimensional 2D vector array. A triple nested for loop is implemented to iterate through the channels, heights and widths to firstly

push each input value into the input row. Then each input row is pushed into the input channel and finally, each input channel is pushed into the final input image.

Padding is also added to the input image in the set inputs function. If the has padding Boolean variable is set to true, then true padding is applied so that rows and columns of 0s are added to the edges of the image. This is done by adding a padding row before and after every input channel and adding a 0 before and after every row when parsing the input image. Finally, 2 is added to the input width and height to reflect the addition of padding.

The calculate outputs functions iterates through each filter and applies that filter to the input image to create what is referred to as an activation map which is part of the final output image. The output image is a 3-dimensional vector where each channel is a different activation map or output after applying a filter. If there are 6 filters in the layer, then the output image will have 6 channels.

Firstly, two variables are created called the max height and width which are calculated by using the input heigh and width minus the height and width of the filter minus one. This is done to find the end point of the input image, so the filter does not iterate too far along the image. This is because the filters are iterated using the top left of the filter. Then a double nested for loop has been implemented to iterate through the input image from 0 to the max height and width using the stride height and width to increase the top left x and y positions. For each position a local variable for the output float is set to 0.0f then a filter is applied by iterating through the height and width of the filter to get the filter value and the corresponding input image value and multiplying them together. The output value for that iteration is the total of the multiplication between the inputs and filter values for the size of the filter. The sum of the outputs is for all channels that the filter iterates over so the final output image will have one channel for each filter used in the layer. This output image is sometimes referred to as the activation map so if there are six filters then the activation map will have six channels regardless of the number of channels the input image has. Finally, these outputs are passed into the activation function and then pushed onto the output image and the outputs vector array.

The reason for having a 3D output image and a 1D outputs array is so that when debugging the outputs, they can be clearly visualized in a 3D array. This is because it is easier to see how

each output value relates to the input image. The 1D outputs array is used to pass into the set inputs function of the next layer in the network. This is to have a consistent set inputs function that takes in a 1D array for each layer.

The convolutional layer class only differs from the initial design with the additions of extra 3D vector array variables for the input and output and the variables for the input loss and weight loss. Just like the connected layer, the get functions for the input costs and weight costs were added for the network class to retrieve these variables so that they can be passed into the previous layer during backpropagation.

The implementation of the convolutional layer was more complex to implement than the connected layer because of the increased number of dimensions to calculate. The first challenge was the parsing of the weights into a 3D array. Originally, the weights were implemented in a 4D array but that was too convoluted to iterate through and didn't account for different sizes of filters. This was changed to a filter class that stored the height, width, channels, and weights for each filter as mentioned above. Once the change was made, the calculation of the outputs was easier to calculate because each filter can now be iterated through and applied to the inputs.

Another challenge was adding the padding to the input image. When padding is added to the input image, the width and height of the input image increase. Therefore, this needs to be accounted for when iterating through the new input image and was an issue when the unit tests for the convolutional layer failed. The cause of this was found to be when the inputs were passed through the constructor and then the set inputs function, the size of the input image would be padded twice. Therefore the passing of the inputs was removed from the constructor and so the set inputs function has to be called for the inputs to be passed in.

Overall, the implementation of the convolutional layer is done in a naïve way. It works for all configurations, but the calculation of the outputs uses 3D arrays which means that the calculation takes longer to complete. It would have been quicker to calculate the outputs using a 1D array to iterate through. More details on the speeds of the calculations will be in the testing and result sections.

7.2.3 Pooling Layer

The third and final layer type is the pooling layer. It is used to reduce the size of the input images to speed up the calculation of the outputs for a CNN. This layer works in a similar way to the convolutional layer in that it uses filters that iterate over the input image to produce output images. In contrast to the convolutional layer, the pooling layer does not have any weights in the filters and instead will either use max pooling or average pooling (Krizhevsky, et al., 2017).

Max pooling will calculate the output for each filter iteration by taking the maximum value as the output for all the values that are covered by the filter. In contrast, average pooling calculates the average of the values that are covered by the filter. These filters are two dimensional and therefore are applied to each channel separately and do not aggregate all the channels into one like the convolutional filters do. For example, if the input image is of size 28x28x6 and the filter size is 3x3, then the output will be of size 8x8x6 which is a significant decrease in size (Sewak, et al., 2018).

For the implementation, the pooling layer class constructor takes in the input height and width, inputs array, filter height and width, stride height and width and a bool to determine whether the pooling layer is using max or average pooling. There is a class variable for all these variables and they get set in the constructor. Similarly to the convolutional layer, the pooling layer also converts the 1D array of inputs into a 3D array but this is done in the constructor as well as the set inputs function.

The calculate outputs function starts by calculating the max height and width for the iterations of the filter. These are simply the input height or width minus the filter height or width. Then the input image is iterated through to get the top-left position for the filter. For each iteration, either the max pool or average pool function is called based upon the bool passed into the constructor.

The average pool and max pool functions both take in three integers that are the top left corner position of the filter for this iteration. In these functions, an output float is initialized. For the max pool function, this output float is initialized to negative infinity and for the average pool function, it is initialized to 0.

Then, the max pool function will iterate through the height and width of the filter to add that to the top left position passed in to get the input image value at each position. If that value is higher than the current output value then the output value is set to that. After the filter has been iterated through, the output value is returned. In contrast, the average pool function will sum up all the values when iterating through the filter and then divide that sum with the filter height multiplied by the filter width to get the mean average for that filter iteration. This mean average is then returned. Once these functions have been called, the returned value is pushed onto the outputs array and the 3D output image.

The implementation of the pooling class was changed from the initial design in that the two additional functions for the max and average pooling. In the design, this was planned to all be in the calculate outputs function but this was changed to make the debugging of the outputs easier. By implementing the pooling calculations into functions, the top-left position for each iteration can be checked alongside the output for that iteration.

The implementation of the pooling layer is similar to that of the convolutional layer in that it was a naïve approach that can be optimized by iterating through the inputs as a one-dimensional array rather than the 3D input image array. However, because it uses the input and output images, it can be unit testing in the same way that the convolutional layer is. As a result, it made it easier to figure out the necessary unit tests to fully test the pooling layer after the convolutional layer was fully tested.

7.3 Backpropagation

The one thing that hasn't been discussed with the layers is backpropagation. The purpose of backpropagation is to adjust the weights and biases of the layers to get the expected outputs for the given inputs. The method that is typically used to calculate the amount to adjust the weights is called stochastic gradient descent (SGD). The objective of SGD is to minimize the loss between the expected output and the output calculated after the inputs have been fed through the network (Bottou, 2012).

The loss is minimized by increasing or decreasing the weights and biases by a small amount to find the optimal values for these weights and biases. One of the common methods of calculating the loss is using the mean-squared error (MSE). MSE is calculated using the following equation: $Error = \frac{1}{n}\sum_{i=1}^{n}(y - g(x))^2$. Where n is the number of sets of inputs per

batch, y is the expected outputs and g(x) is the calculated outputs. A batch is a randomized set of inputs with their respective expected outputs (Ketkar, 2017).

The loss can be minimized by decreasing or increasing the weights and biases by a value. This value can be calculated by finding the derivative of the cost over the weight: $\frac{dCost}{dWeight}$. The chain rule states that this calculation can be broken down into this equation: $\frac{dCost}{dWeight} = \frac{dCost}{dZ0} * \frac{dZ0}{dWeight0}$. During the process of backpropagation, the layers are evaluated in the opposite order in which the outputs are calculated. So when referring to the previous layer, that is the previous layer going back to front in the network. The methods of calculating these values differ slightly depending on the layer type.

7.3.1 Backpropagation for the Connected Layer

The first part of this equation is the $\frac{dCost}{dZ0}$ which is the derivative of the cost over the output calculated by a single neuron and can be calculated using the equation: $\frac{dCost}{dZ0} = \frac{dCost}{dO0} * \frac{dO0}{dZ0}$. Backpropagation starts with the final layer in the network and works backwards so this equation represents the difference between the output and expected output multiplied by the output multiplied by one minus the output. This also calculates the cost for the bias associated with that neuron which can be written as $\frac{dCost}{dBias}$ (Wolfe, 2017).

However, this only works for the final layer in the network and so the equation for $\frac{dCost}{dO0}$ differs from the previous layers in the network. The new equation is calculated using the sum of the previous layer's bias costs multiplied by the weights that are relevant to this neuron. This is referred to as the input costs for the previous layer and will need to be passed into the previous layers to get the bias and weight costs (Wolfe, 2017).

The second part of the main equation is $\frac{dZ0}{dWeight0}$ which is the derivative of the output over the weight. This is simply just the input that the weight is multiplied by. Therefore, to get the cost of the weight, the cost of the bias needs to be multiplied by the input for that weight. In summary, for backpropagation in the network the following values need to be calculated: cost of the bias, cost of the weight and the cost of the inputs for that layer (Wolfe, 2017).

The implementation of backpropagation for the connected layer is calculated in a two-step process. Firstly, the weight, bias and input costs are calculated and then these values are applied to the weights and biases. This process is done in three functions: backpropagate, calculate input costs and update weights and biases.

There are two variations of the backpropagate function: one which takes in a vector array of floats that represent the expected outputs and one which takes a vector array of floats are the previous layers inputs costs. The first variation is used when the connected layer is the last layer in the network and so the second variation is used when there are layers that are not the final one in the network.

Both functions start by calculating the number of weights and the number of weights per node. Then each node is iterated through, and the bias cost is calculated. The calculation of the bias cost differs slightly between the variations. For the last layer, the output for the node is subtracted by the expected output for that node and that is multiplied by the $output * (1 - output)$ to get the bias cost. When the layer is not the last one, the output subtracted by the expected output is replaced with the previous layer costs that are passed in. This bias cost is then pushed into a vector array.

The weight costs are then calculated by iterating through all the weights for that node and multiplying each input by the bias cost to get the weight cost for each weight. These are pushed back into a vector array of weights costs. Finally, the input costs are calculated in a separate function. Each input is iterated through and the cost of this input is calculated by iterating through the number of weights and multiplying the weight by the bias cost. These values are summed together for each weight that connects to the input to get the final cost for that input. Just like the weights and biases costs, these values are pushed into a vector array of input costs.

The update weights and biases function takes in two vector arrays of floats that are the weights costs and biases costs for this layer. Then each weight and bias are iterated separately to subtract the current weight or bias value by the passed in costs multiplied by the learning rate which is also passed in: $weight = weight - (learning\ rate * weight\ cost)$.

The backpropagation was initially planned in the design to have all calculations in one function but this was separated into two main functions in which one function calculates the costs and

the other updates the weights and biases. This change was made after testing the backpropagation with multiple inputs. When testing with multiple sets of inputs the costs will need to be averaged to get an average cost for all the sets of inputs. This was found to be an issue with one function because the costs for each set of inputs need to be aggregated and then applied once the average was found. Separating the calculation of costs and updating the weights and biases, allows the average costs to be calculated outside of the class and then passed into the update function once the average has been calculated.

The implementation of the backpropagation for the connected layer was the most difficult part of the artefact to implement. This was because of the amount of maths equations to understand and convert into code. It was hard to figure out what the derivations were and how to calculate each one using the variables already implemented for the calculation of the outputs. For example, it took a while to figure out that equations such as $\frac{dCost}{dZ0} * \frac{dZ0}{dWeight0}$ just meant the $bias\ cost * input$. Once these conversions from maths to variables were made, backpropagation for the connected layer was easy to implement.

Overall, the implementation for backpropagation in the connected layer was completed and fully tested in time and works as expected regardless of the initial difficulty in understanding the maths behind it. However, some optimizations can be made to improve the implementation of backpropagation. These include methods of adapting the learning rate to avoid hitting incorrect optimal weight and bias values and speed up the rate at which the correct weight and bias values are reached.

7.3.2 Backpropagation for the Convolutional Layer

Backpropagation differs slightly for the convolutional layer in the way that the costs for the bias, weights and inputs are calculated. Since the weights are in filters which are applied to the inputs and there are no biases in the convolutional layer, the cost of the bias does not need to be calculated. The loss for each filter can be calculated using the equation: $\frac{dCost}{dFilter} = \frac{dCost}{dO0} * \frac{dO0}{dFilter}$ where, as previously mentioned, $\frac{dCost}{dO0}$ is the cost of the inputs from the previous layer and $\frac{dO0}{dFilter}$ is the filter. Because the cost of the inputs and the filters are in three dimensions, the loss from each filter can be calculated using a convolution between the cost of the inputs and the filter. This is illustrated in figure 17 below (Solai, 2018).

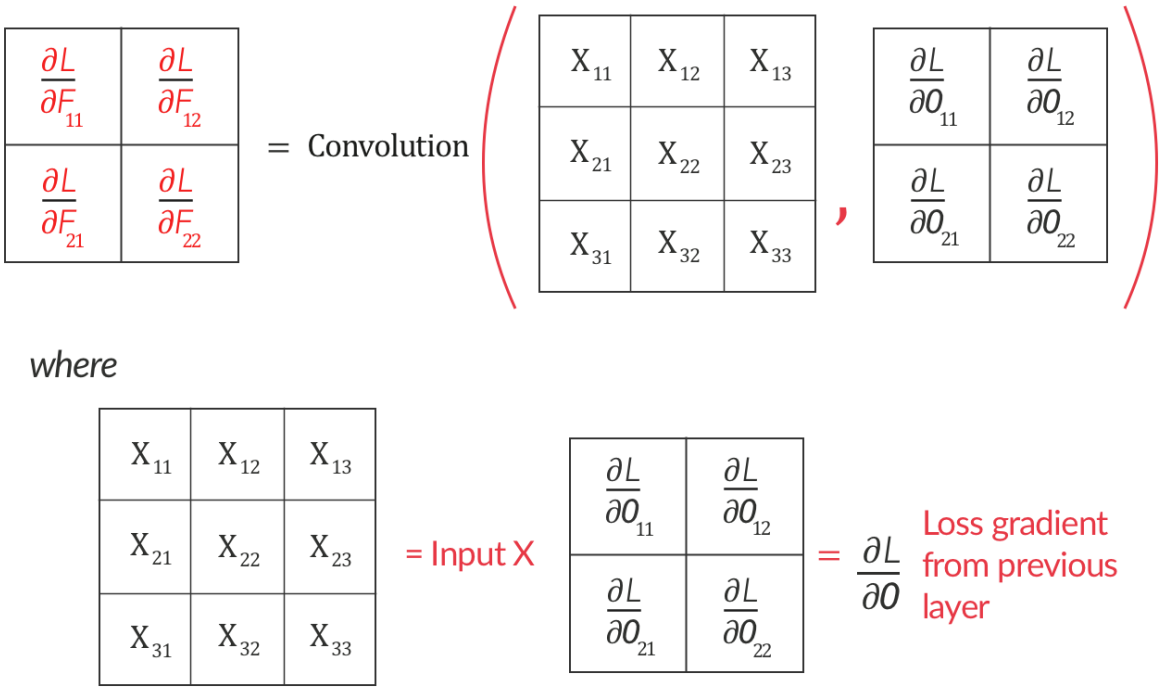


Figure 17: Illustration of the calculation of the loss for a filter

The loss for the inputs is calculated using the following equation: $\sum_{k=1}^{n} \frac{dCost}{dO0} * \frac{dO0}{dZ0}$. As previously mentioned, $\frac{dCOst}{dO0}$ is the loss from the previous layer but since there is no bias for this layer, $\frac{dO0}{dZ0}$ is the filter rotated 180 degrees. Therefore, to get the loss for the inputs, the rotated filters need to be multiplied by the loss from the previous layer which is a convolution between the two. This process is shown in figure 18 below (Solai, 2018).

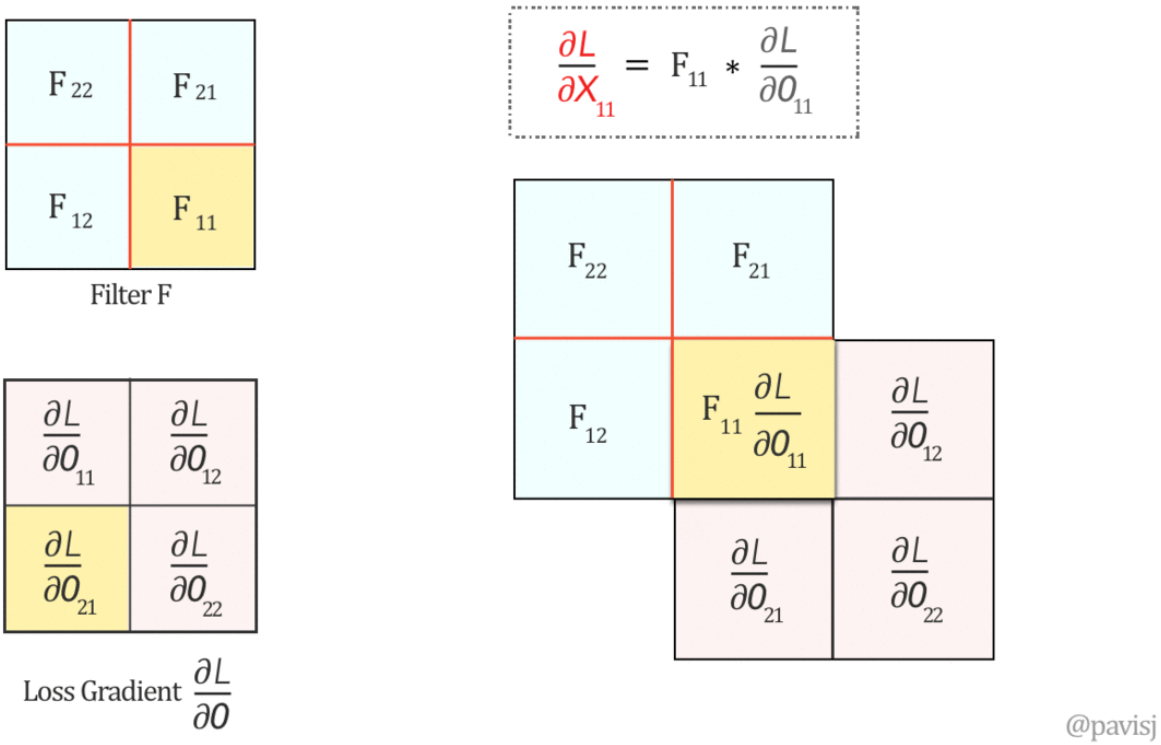


Figure 18: Process of calculating the loss for the inputs of a convolutional layer

Just like the connected layer, the implementation of backpropagation for the convolutional layer occurs in two functions: backpropagate and update weights and biases. However, it differs by only having one variation of the backpropagation layer with takes in a vector array of floats for the loss of the previous layer. This is because the convolutional layer is not typically the final layer. After all, the outputs are in three dimensions and would be fed into a connected layer to produce outputs in a format which can be interpreted.

Firstly, the loss of the previous layer that is passed in is converted to a 3D vector array called a loss image. Then the number of filters is iterated through because each channel of the loss image is the output of a filter convolution. The max height and widths are calculated so that the input image can be iterated through. The loss for the weights is calculated by iterating through the input height and width similarly to calculating the outputs but instead the output is calculated by summing the loss image values multiplied by the input image values for each filter iteration to get the loss values for that filter. The loss values for each filter are pushed into a 1D vector array and stored in each filter as a 3D array of floats.

The calculation of the costs for the inputs is calculated by first making a copy of the filter and then transposing these filters by rotating them 180 degrees. Then these transposed filters are

applied to the loss image from the previous layer. For this calculation, the start x and y positions are calculated using one minus the filter height and width. This is because, as shown in figure 19, the start positions for the convolution are not at 0,0 like the process of calculating the outputs. To account for this, there is a check before calculating the multiplication to check that the indexes for the x and y positions are within the bounds of the loss image. If they are not, then the output is not calculated. To get the final output for each input, the sum of the filter values multiplied by the loss image values is calculated for all the filters to get a sum of all filters. These values are all stored in a 1D array of floats called loss input.

In the update weights and biases function, the filter values are updated by iterating through each value in each filter and subtracting the weight costs multiplied by the learning rate which are passed into the function. Since this function is overridden from the base layer class, the bias costs are passed in but are ignored for the convolutional layer because there are no biases to update.

Just like the connected layer, the only change that was made to the backpropagation was the separating of the calculating the costs and updating the weight and biases functions.

The concept of backpropagation for the convolutional layer was not difficult to understand once the realization that the costs were calculated using convolutions between the filters, loss of image from the previous layer and the input image. The implementation was completely working and tested a lot quicker than originally anticipated. However, during the testing of the backpropagation, it was found that padding was not accounted for so some of the unit tests failed. This was easily fixed by increasing the max height and width when calculating the input costs.

7.3.3 Backpropagation for the Pooling Layer

Unlike the convolutional and connected layers, the pooling layer does not have any weights or biases so the only derivative to calculate is the loss for the inputs. For the max-pooling layer, the maximum input is the output for that filter iteration. Therefore the derivative of that output is itself and so this number is the loss for the input at the same position in the input image. However, this leaves the other values that were not taken as the output need to be derived. These values that were not the max are set to 0 since the max value position takes

100% of the output. The process of backpropagation for the pooling layer is shown in figure 19 below (Kravets, 2021).

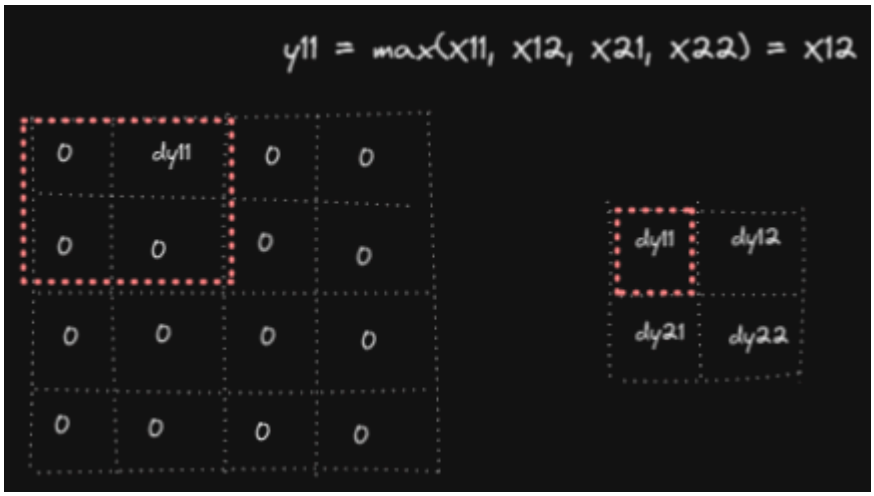


Figure 19: Illustration of backpropagation of the pooling layer

Since only the input costs need to be calculated, the implementation of backpropagation for the pooling layer was quick to complete and test. This process is once again done in the backpropagate function that takes in the loss from the previous layer. Firstly, the loss input image for this layer is initialized to be the same size as the input image. Then the input image is iterated through to get the top left values as if calculating the output. A separate function called backpropagate max pool is called that takes in the top left values, the channel index and the value from the previous layer costs that are in the same position as the output is for that filter iteration. This function gets the output for that iteration from the output image and then the filter is iterated through and if the input in one of the positions is equal to the max number for that filter iteration then the value in that position in the loss input image is set to the max, else it is set to 0. Once the costs for the loss input image have been calculated then they are all pushed into a 1D array called loss input.

The implementation of backpropagation for the pooling layer has an additional function called backpropagate max pool that was not initially planned in the design. This function was created in the eventuality of having a max pool and average pool function. However, the average pool function was not implemented in time and therefore the extra function was not necessarily needed. But the separation of the max pool backpropagation is helpful for the debugging of the backpropagation because it makes it easier to see how each output is calculated. So overall, the implementation of backpropagation for the pooling layer works

well for max-pooling but it is not completed because the implementation for an average pooling layer was not done in time.

7.4 Network Class

The purpose of the network class is to handle the passing and calculation of the inputs and outputs from layer to layer. It also handles the training of the network by calculating the loss and aggregating and passing the relevant costs to each layer in the network.

The network itself is a vector array of layer pointers where each layer is added to the network by calling the add layer function. The inputs for the first layer can be set by calling the set inputs function which directly calls the set inputs function for the first layer in the network. The output for the network can be calculated by calling the calculate outputs function. This function first asserts to check there are layers in the network array then the current outputs are initialized as the inputs of the first layer. After this, each layer is iterated through to set the inputs as the current outputs, call the layer's calculate output function and then set the current outputs as the output for that layer. Finally, the current outputs of the final layer are returned.

There is a backpropagate function that handles the backpropagation of the whole network for a set of inputs. It takes in the expected outputs for the set passes that to the backpropagation function for each layer in the network going from back to front. Then the weight and bias costs are retrieved from each layer and appended to an array with all the weight costs and an array with all the biases costs for the whole network. Finally, the value of the costs for this set is added to the running total of the cost values for all the sets in the batch which are stored in two vector arrays called biases costs and weights costs.

A function called train network has been implemented to handle the overall training of a network that has multiple sets of inputs and expected outputs in the batch. It takes in two vector arrays: one for the inputs for that batch and one for the corresponding expected outputs for each input. Firstly, the number of inputs and expected outputs per set are calculated. Then an input index and output index are iterated increasing them by the number of inputs and outputs per set for each iteration. So for each set, the inputs and expected outputs are set from the vector arrays passed in and then the inputs are set, the outputs are calculated and the backpropagation function is called. After these functions have been called,

the outputs for that set are pushed back into a class vector array of floats for all the outputs from all the sets of inputs.

The outputs are then used with the expected outputs to calculate the loss for the training batch using the equation: $\frac{\sum(expected\ outputs - outputs)^2}{number\ of\ outputs}$. This is called the mean-squared error (MSE) and represents the loss for the network as a whole. MSE was chosen because it is the most commonly used error calculation for regression neural networks and cross-entropy is used for classification networks (Reed & Marks, 1999). However, cross-entropy was not implemented in time so MSE is used for all types of neural networks.

Finally, the bias and weight costs are divided by the number of sets of inputs to get the average costs for all the sets in the batch. Then the update weight and biases function is called to pass these costs into the relevant layers to adjust the weights and biases.

There were a couple of changes made to the neural network class that differentiates it from the initial design. The first change was the separation of the backpropagation function to have the update weights and biases separate from the calculation of the costs. This was done so that the costs can be summed and averaged for all the sets in the batch before updating the weights. Also, the train network class was added to tie together the backpropagate and update functions. This was only needed for the implementation of SGD which requires a different batch of inputs and expected outputs for each iteration of the training of the network.

Originally, only gradient descent was planned to be used but after testing the network with a training set that had a large number of sets of inputs, it was found that the network would take too long to train. SGD optimises the times by creating a smaller set of inputs called a batch in which the inputs in the batch are chosen at random. This works because the costs for a smaller subset are likely to be representative of the costs for the whole set of inputs and speed up the training because fewer inputs are being trained on (Bottou, 2010).

In reflection, the implementation of the network class works well for the training of most neural networks but cross-entropy needs to be added for better error calculation of classification neural networks. Also, training of the networks works for each iteration but the handling of each iteration should be done by this class and it currently is handled outside of

this class. So another function needs to be added in the future that simply calls the train network function for each iteration of the training so that the randomization of the batches is done within the class. Ideally, this function will only need to be called once so that it makes it easier for the user to train the network because they don't have to do the batch preparation themselves.

7.5 Neural Network Creation and Training

A neural network can be created using the library by first creating the separate layers and passing them into a neural network object using the add layer function. For the creation of each layer, the inputs, weights, biases, and activation type will need to be passed into the constructor. This works well for networks which have predefined weights and biases but there is no option for the weights and biases to be initialized by the layers and will have to be done by the user. This is good for pre-trained networks but this is an issue for networks that need to be trained because the initialization of weights and biases are crucial to how well a network trains. Some of the methods of initialization that could be included in the layer classes are the He initialization and Xavier initialization (Yadav, 2018).

The process of training a network requires a loop in which the train network function is called for each iteration and the inputs and expected outputs for that iteration need to be passed in. As previously mentioned, the process of training a network can be made easier for the user if the iterations are handled by the network class. This is so that the handling of inputs for each batch can be handled by the network.

8.0 Testing

As previously mentioned in the design and implementation, part of the testing strategy for the artefact is unit testing the different aspects of the library. The purpose of the unit testing is to test all the possible outcomes of the functions using test inputs and the expected outputs. An example of this sort of test is the test for the activation functions. There are three tests for each activation function which takes in three inputs: a negative number, a positive number and the number zero. Using these numbers means that the activation functions can be tested for different situations and accounts for outliers therefore if the three tests pass then the activation functions will work for all situations. There are also unit tests that test the failure of certain functions when the expected output is an assertion. These are referred to as death tests and have been used to test that functions such as the set inputs functions fail when the incorrect number of inputs were passed in.

8.1 The implementation of the Unit Testing

Unit testing has been implemented using the Google Test adaptor for Visual Studio. There is a multitude of different unit testing applications. One of the reasons why Google test was used is because it has integration with Visual Studio. This makes the process of creating and linking the tests to the library easier to do within Visual Studio and it allows the unit tests to be created in a separate project so that the unit testing can be run separately from the main project. As well as this, the integration of the Google tests in Visual Studio means that the unit tests can be debugged using the visual studio debugging tools. This means that when the unit tests failed, they were easily debugged to fix the error.

Another reason it was chosen is that classes that inherit from a base test class can be created to initialize the variables for a set of tests. For example, there is a connected layer class which initializes the inputs, weights, biases and the layer pointer. This means that all the connected layer tests can use the same variables without the need to re-initialize them for each test. On top of this, this allows the tests to be separated based upon which class they are testing so it makes it clearer to see which test has failed and the class that test has failed on. This is helpful considering that each class will have similar tests. For example, there is a constructor test for each of the layer classes so separating the tests based on the layer helps to define which layer's constructor may have failed.

The final reason Google unit testing was chosen is because Google testing has a function called expect float which accounts for the floating-point error that can occur. This is useful for the testing of the artefact because all the maths is calculated using floats for the inputs and outputs. The expect float function accounts for the floating-point error by rounding the output and expect outputs to six decimal points before evaluating them to check if they are equal.

8.2 Example Network Tests

Another testing method that was used for the artefact was the creation and testing of two different neural networks: one which uses connected layers to emulate an XOR gate and one which uses a convolutional, pooling and connected layer for the classification of images of digits from 0 to 9. The success of the two networks will be determined based on the loss decreasing for each iteration of the training and the testing of the network once it has been trained.

The network that emulates the XOR gate is an example of a regression neural network that tests the network's suitability for creating and training a simple two-layer network using the library. As previously mentioned above, the success of the network will be determined by tracking the loss after each iteration and testing the network after it has been trained. The loss of the network is tracked by getting the loss using the get cost function in the network class and pushing it into a vector array of floats. This array of floats is passed into a function that uses the loss values to draw a graph showing the loss over the iterations. This function uses a Github repository to draw the graph and output it to a file (Johansen, 2021). The network is then tested using the same inputs as the network was trained with.

The classification neural network is a CNN consisting of a convolutional, pooling and a connected layer that takes images of hand-drawn digits and predicts the digits that are drawn in each image. The data used for training this network are selected at random from a larger dataset and added to a batch of size 150 images for each iteration. Once trained, the network is tested against 100 images from a different set of images selected at random from the testing dataset. Both of these datasets comprise images from the MNIST dataset which is the most commonly used dataset for classification (LeCun, et al., 1998). The MNIST dataset

images are parsed using a function which creates a 1D vector array of values between 0 and 255 that represent the greyscale pixel values of each pixel in an image (Wicht, 2019).

9.0 Results

9.1.0 Unit Tests Results

Unit tests were used as part of the iterative design process so when the unit tests failed, the code was modified until the unit tests passed. As a result, in its current state, most of the unit tests pass because the issues causing the failure of these tests have been fixed before moving on to other parts of the implementation. However, the softmax activation function unit test is the only test that failed. This is because the softmax function was added last to calculate the outputs for the classification CNN. Therefore, there was not enough time to fix the issues with the softmax function. As shown below in figure 20, the outputs of the softmax function are not correct to six decimal places. However, they are all correct to five decimal places which were deemed satisfactory for the current solution but will need to be fixed in the future.

```
[ RUN      ] ActivationsTests.SoftmaxActivation
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.016590023
  expectedOutputs[i]
    Which is: 0.016589999
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.40699518
  expectedOutputs[i]
    Which is: 0.406995
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.011120625
  expectedOutputs[i]
    Which is: 0.011121
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.020880206
  expectedOutputs[i]
    Which is: 0.020880001
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.44979942
  expectedOutputs[i]
    Which is: 0.449799
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.020263102
  expectedOutputs[i]
    Which is: 0.020262999
C:\Users\k013044i\Documents\GitHub\NN-Library\NN-UnitTesting\test.cpp(74): error: Expected equality of these values:
  outputs[i]
    Which is: 0.074351333
  expectedOutputs[i]
    Which is: 0.074349999
[  FAILED  ] ActivationsTests.SoftmaxActivation (3 ms)
```

Figure 10: Output from the softmax unit tests

9.2.0 Neural Network Results

After training the XOR gate neural network for 500,000 iterations, the loss decreases for each iteration in an exponential curve as shown in figure 21 below. This loss graph is similar to what

is expected of a typical loss graph because the loss does not vary as it decreases and follows an exponential curve that decreases heavily at the start and decreases less and less over the iterations. This shows that the neural network is training correctly the get closer to the expected outputs.

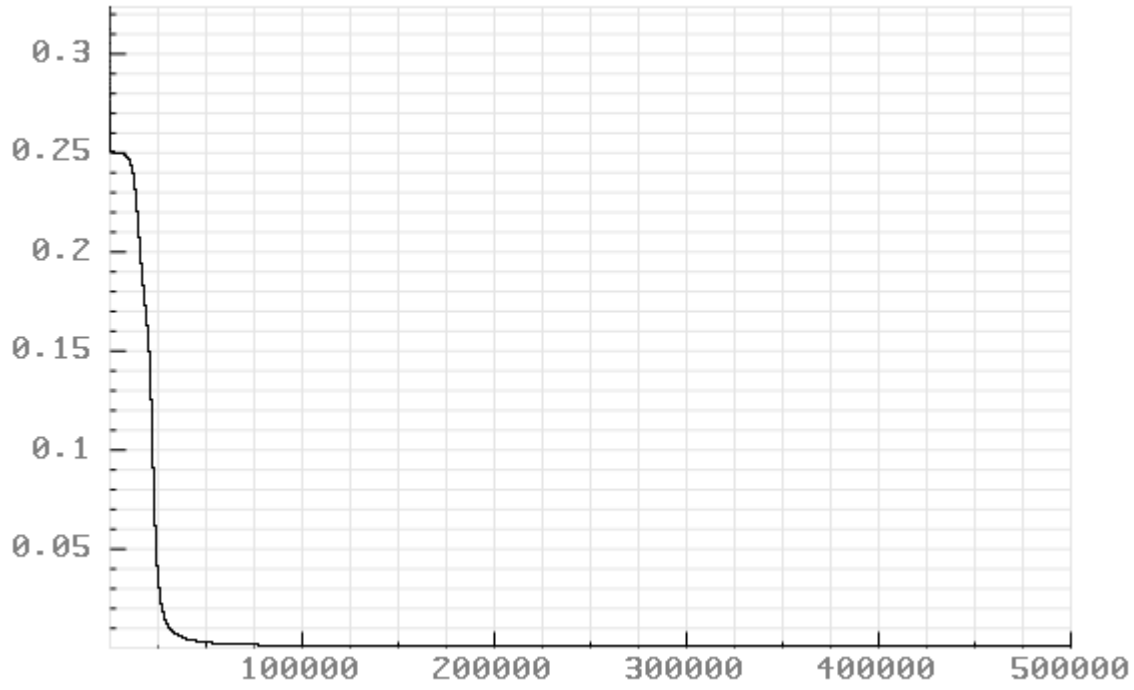


Figure 11: Cost output of the XOR Neural Network

After the network was trained, the outputs are shown in figure 22 below. The expected outputs are 0, 1, 1, 0 and so the outputs of the network are off by two decimal places. This is a positive result because these outputs show that the network is trending towards the expected outputs and therefore if the network is trained for more iterations, the outputs will be closer to the expected outputs. The training of the neural network took 82 seconds to run in release mode on a CPU. Even though this did not take a long time to train, the process can be sped up drastically by running it on a GPU using CUDA cores.

```
Output = 0.0117277

Output = 0.990054

Output = 0.990015

Output = 0.0102406

Time taken to train: 82 seconds
```

Figure 12: Output of the network after being trained using 500000 iterations

The classification neural network was trained for 1000 iterations and the loss graph is shown in figure 23 below. The trend of the loss is like that of the previous neural network because it follows an exponential curve. However, it differs because the variation in the loss values increases as the number of iterations increases. This is not expected because the loss should not be varying to that extent but overall the trend of the loss is good because it is still decreasing. Because of this variation in the loss, it is difficult to say whether the loss will keep decreasing if the number of iterations increases. This can be improved in the future by implementing cross-entropy to calculate the loss so that the correct loss values are calculated for classification neural networks.

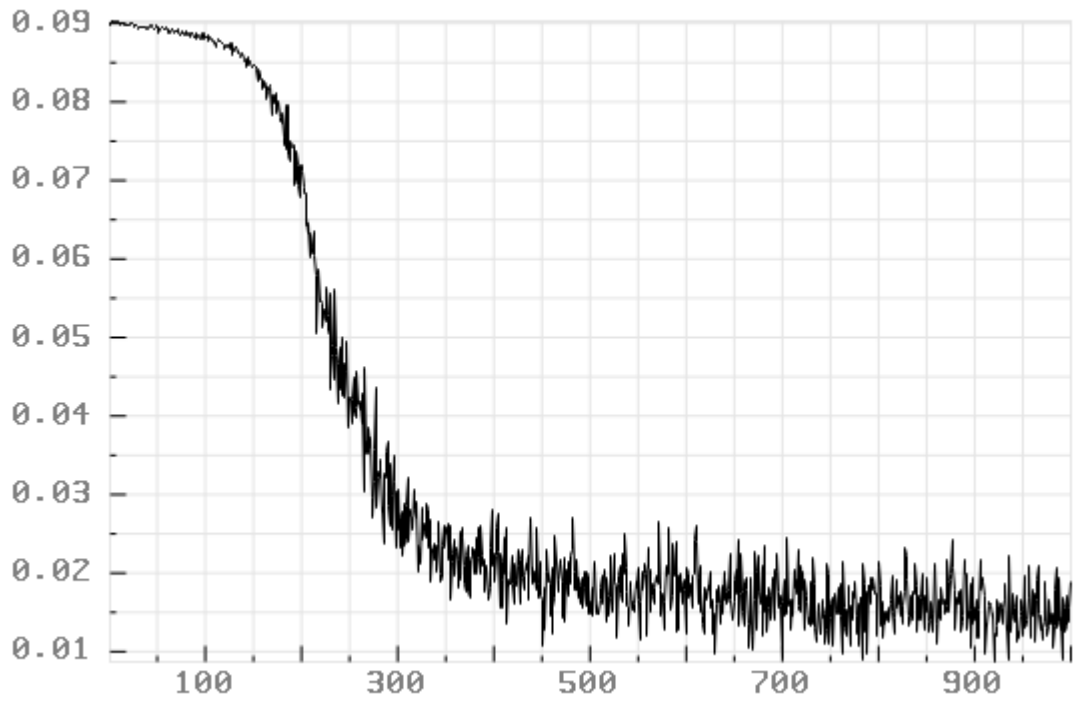


Figure 13:Loss graph of the CNN for 1000 iterations

The output of the trained CNN is shown in figure 24 below. After 1000 test inputs were fed through the network, it predicted 905 correct which gives the network accuracy of 91%. This is a good result because this accuracy suggests that the network has been correctly trained to predict the digits in the images. Consequently, this shows good progress towards making an object detector because classification is part of object detection. However, the network took 73 seconds to train using 1000 iterations which is slow in comparison to the time it would take to train this network on a GPU. Therefore, an object detector network would take a lot longer to train because there will be more layers in an object detector. So before, implementing the functions needed for object detection, the library will have to be modified to allow these networks to be trained on a GPU using the CUDA cores.

```
Number of correct tests = 905/1000
Time taken to train: 73 seconds
```

Figure 14: Output of the trained classification CNN

10.0 Critical Evaluation

10.1 The success of the project

To determine the success of the project as a whole, the criteria used are the objectives that were defined at the start of the project. The first objective is to research the current state of object detection. This was completed by conducting research into the background of object detection to see how the techniques have changed and improved over time. The research into the background was completed thoroughly to ensure that the most prominent innovations in object detection were mentioned. Therefore, the first objective was successfully met because it gives an understanding of how the current state of object detection has come to be.

The second objective is to research and analyse the different methods of object detection to find the best method for this project. This was completed by first researching the two object detectors: R-CNN and YOLO. Then these object detectors were evaluated using the criteria needed specifically for this project. The research into these two detectors comprised of detailing the methods of these detectors to get an understanding of how they work whilst also highlighting the differences in these methods. The key part of this research is these differences because they helped with the evaluation of these methods. The evaluation of these methods looked at comparing the two methods in three aspects: accuracy, speed and utility. These criteria were chosen because they are the best ways to determine the most suitable object detector for this project. Upon reflection, the research into the two object detectors was very good at finding the best method of object detection for this project because these methods were compared with the end purpose in mind. Therefore, the second objective was successfully met using the analysis of R-CNN and YOLO.

The third objective is to create and train an object detector that uses both real and synthetic data. After researching the methods of creating and training the object detectors, it was found that this was not possible to implement an object detector in the given time frame as explained further in the scope section of this report. Therefore, this objective had to be reworked to give a scope that was small enough to complete in the given time frame. Consequently, the objective of the artefact was to implement a neural network library capable of producing and training a neural network that can classify images into categories.

This objective was completed as shown in the example network that was trained using the MNIST dataset. This network had a 91% accuracy of classifying drawn digits from 0 to 9 which means it is possible to create and train a neural network using the artefact library that can classify images into categories. Classification is a part of object detection so this shows that, if given more time, the library can be built upon to create and train an object detector. Overall, the artefact does not fully complete the third objective but is a good base to build upon to fully complete it.

The fourth objective is to improve the accuracy and speed of the object detector to work in real-time. The current state of the artefact was found, in the testing phase, to take too long to train for it to work in real-time. This shows that optimizations will need to be made to the library to speed up the time it takes to train and test the networks made using the library. So the fourth objective was not met.

The fifth objective is to implement the object detector in a synthetic environment to test its viability. This objective was not met because the artefact is not in a state to create an object detector and therefore it cannot be tested in a synthetic environment. However, given more time, the library can be built upon to complete this objective.

The time management of this project is illustrated in the Gantt planner (see appendix 1). The weeks surrounded by a thick border show the planned breaks in which no work should be done in those weeks. Even though the majority of tasks were done in the time frames given, there were a couple of instances where the tasks overran. The biggest examples of these are the write up of implementation and the testing which was started two weeks after the planned weeks. This happened because implementation had to be finished before it could be fully tested and written up. As a result, these tasks overran into weeks 28 and 29 which were planned as a break. Regardless, all the tasks were done before the evaluation was planned to start and so the evaluation took one week less than expected.

In conclusion, an adequate amount of research has been done into potential solutions to this project. However, the artefact produced does not provide a solution to the original solution but is a good start considering the allotted time frame given for this project.

10.2 Potential changes if this project was repeated

If this project was repeated, the first aspect of this project that would be changed is to adjust the objectives of this project to reflect something that would be possible to implement in the given time frame. This is so that the research is done to find a solution to the objectives that will be more relevant to the artefact implemented. Also, the artefact implemented will likely be able to solve the adjusted objectives if the target objectives are adjusted with the allotted time frame considered. The adjusted objectives will likely be to research into and implement a classification neural network library which can be used to classify objects that are seen from a helicopter camera.

If the project was repeated with the goal of building a similar artefact, the main change that would be made to the artefact is the use of standard float arrays instead of vector arrays. The use of standard arrays would have sped up the process of training and testing a neural network. This is because standard arrays are quicker to access and modify the values in the array which is something that is done often with the calculation of the outputs and the backpropagation of neural networks.

10.3 Extensions that could be made to this project

The first feature that is recommended to extend the artefact is the addition of cross-entropy for the calculation of the loss for classification neural networks. This is recommended because the current calculation for the loss, MSE, is used for regressions neural networks but it is not a good calculation for the loss of classification neural networks. The reason why cross-entropy is better than MSE is that it helps to train the network faster and improves the generalization of the classes to categorize (Bishop, 2006).

Another feature that could be added to the library is momentum to the training process. Momentum is used to modify the calculation of the weight and biases costs to use an exponential weighted average of the costs instead of just the average of the costs. By doing this, it converges faster than just using gradient descent and therefore trains at a steadier rate. This would be a solution to the variation of the loss shown in the results section. A visualization of this comparison is shown in figure 25 below (Michelucci, 2018).

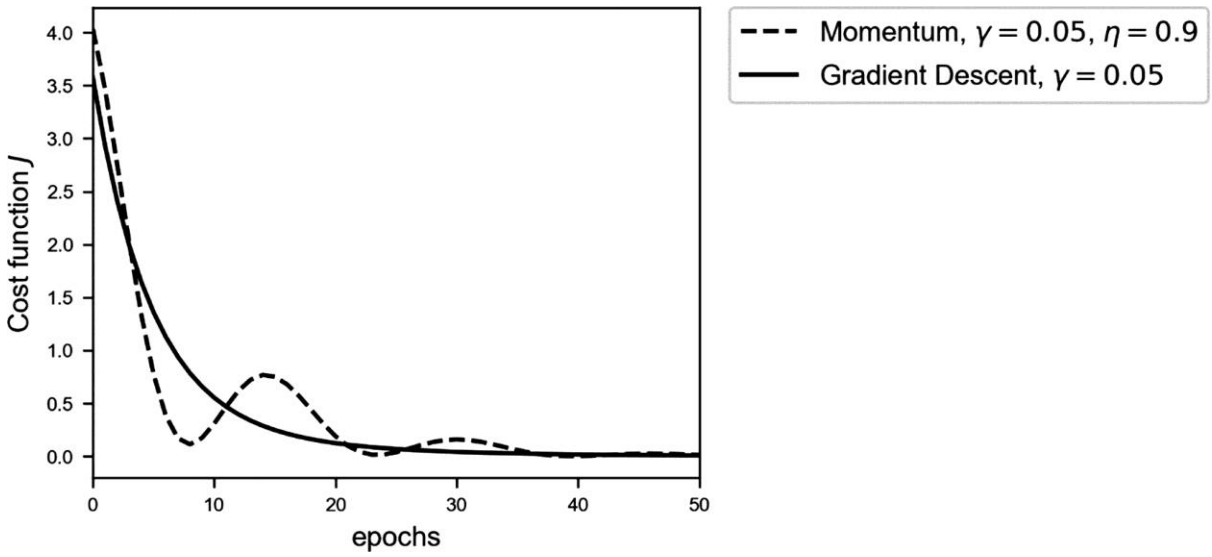


Figure 15: Visualization of the comparison between gradient descent with and without momentum

A final feature that could be added to the library is the initialization of the weights in the layer classes. The initialization of the weights can make the difference when it comes to the network converging and therefore must be initialized properly for the network to train correctly. Currently, the weights must be initialized and passed into the layers but this should be changed so that the user does not initialize the weights and is instead initialized in the constructor using the following two algorithms: Xavier and He weight initialization. Xavier initialization is used when the activation function is sigmoid or tanh for the layer. He initialization is used when ReLU as the activation function for the layer (Goodfellow, et al., 2016).

10.4 Future Work

The project could be worked on further as part of a master's degree to build upon the library to make object detection neural networks. These neural networks could then be trained using a mix of synthetic and real data to see if synthetic data is viable as an alternative for real data when training object detection neural networks. The trained networks could be tested in a synthetic environment to test their viability in real-time. The research for this project will have to look at the techniques used in YOLO that help to speed up object detection and potential methods that improve the accuracy of the networks.

Bibliography

Albawi, S., Mohammed, T. A. & Al-Zawi, S., 2017. *Understanding of a Convolutional Neural Network.* Antalya, IEEE.

Artelnics, 2021. *OpenNN Github.* [Online]
Available at: https://github.com/Artelnics/OpenNN
[Accessed 6 January 2022].

Bishop, C. M., 2006. *Pattern recognition and machine learning.* 1st ed. New York: Springer.

Bochkovskiy, A., Wang, C.-Y. & Liao, H.-Y. M., 2020. *YOLOv4: Optimal Speed and Accuracy of Object Detection.* s.l.:s.n.

Bottou, L., 2010. *Large-Scale Machine Learning with Stochastic Gradient Descent.* s.l., Physica-Verlag HD.

Bottou, L., 2012. *Stochastic Gradient Descent Tricks.* 1st ed. Berlin: Springer.

Brownlee, J., 2019. *How Do Convolutional Layers Work in Deep Learning Neural Networks?.* [Online]
Available at: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/
[Accessed 3 April 2022].

Chen, Y., Li, R. & Li, R., 2021. HRCP : High-ratio channel pruning for real-time object detection on resource-limited platform. *Neurocomputing (Amsterdam),* 463(1), pp. 155-167.

Dalal, N. & Triggs, B., 2005. *Histograms of oriented gradients for human detection.* San Diego, IEEE.

Du, J., 2018. Understanding of Object Detection Based on CNN Family and YOLO. *Journal of Physics,* 1004(1), p. 12029.

Erickson, J., Lyytinen, K. & Siau, K., 2005. Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. *Journal of database management,* 16(4), pp. 88-100.

Evelin, Suclinton, Kwa, R. V. & Anggreainy, M. S., 2021. *Developing an Application to Hire a Private Tutor Using Scrum Method.* Bandung City, IEEE.

Felzenszwalb, P. F., Girshick, R. B. & McAllester, D., 2010. *Cascade object detection with deformable part models.* San Francisco, IEEE.

Felzenszwalb, P. & Huttenlocher, D., 2004. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision,* 59(2), pp. 167-181.

Felzenszwalb, P., McAllester, D. & Ramanan, D., 2008. *A discriminatively trained, multiscale, deformable part model.* Anchorage, IEEE.

Girshick, R., 2015. *Fast R-CNN.* Santiago, IEEE.

Girshick, R., Donahue, J., Darrell, T. & Malik, J., 2014. *Rich feature hierarchies for accurate object detection and semantic segmentation.* Columbus, IEEE.

Girshick, R., Donahue, J., Darrell, T. & Malik, J., 2014. *Rich feature hierarchies for accurate object detection and semantic segmentation.* Columbus, IEEE.

Goodfellow, I., Bengio, Y. & Courville, A., 2016. *Deep Learning.* 1st ed. Cambridge: MIT Press.

Hakim, H. & Fadhil, A., 2021. *Survey: Convolutional Neural networks in Object Detection.* Babylon-Hilla City, IOP Publishing.

Huber, J., 2020. *Batch normalization in 3 levels of understanding.* [Online]
Available at: https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338#b93c
[Accessed 10 December 2021].

Hui, J., 2018. *mAP (mean Average Precision) for Object Detection.* [Online]
Available at: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173
[Accessed 26 November 2021].

Jiao, L. et al., 2019. A Survey of Deep Learning-Based Object Detection. *IEEE Access,* 7(1), pp. 128837-128868.

Jiao, L. et al., 2021. New Generation Deep Learning for Video Object Detection: A Survey. *IEEE transaction on neural networks and learning systems,* 1(1), pp. 1-21.

Johansen, M., 2021. *pbPlots.* s.l.:Github.

Karlsson, E.-A., Andersson, L.-G. & Leion, P., 2000. *Daily build and feature development in large distributed projects.* Limerick, ACM.

Ketkar, N., 2017. Stochastic Gradient Descent. In: *Deep Learning with Python.* 1st ed. Berkeley: Apress, pp. 113-132.

Khashei, M. & Bijari, M., 2010. An artificial neural network (p, d, q) model for timeseries forecasting. *Expert system with applications,* 37(1), pp. 479-489.

Kravets, A., 2021. *Forward and Backward propagation of Max Pooling Layer in Convolutional Neural Networks.* [Online]
Available at: https://towardsdatascience.com/forward-and-backward-propagation-of-pooling-layers-in-convolutional-neural-networks-11e36d169bec
[Accessed 16 April 2022].

Krizhevsky, A., Sutskever, I. & Hinton, G., 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM,* 60(6), pp. 84-90.

Larman, C., 2005. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development.* 3rd ed. s.l.:Upper Saddle River, N.J. : Prentice Hall PTR, c2005..

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P., 1998. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE,* 86(11), pp. 2278-2324.

Lei, H., Ganjeizadeh, F., Jayachandran, P. K. & Ozcan, P., 2017. A statistical analysis of the effects of Scrum and Kanban on software development projects. *Robotics and Computer-Integrated Manufacturing,* Volume 43, pp. 59-67.

Metherall, J., DiJoseph, P., Conti, N. & Britton, C., 2007. *Software Development Planning and Management System.* United States, Patent No. 735,682.

Michelucci, U., 2018. *Applied Deep Learning A Case-Based Approach to Understanding Deep Neural Networks.* 1st ed. Berkeley: Apress.

Minhas, M. S., 2021. *Unit Testing in Deep Learning.* [Online]
Available at: https://towardsdatascience.com/unit-testing-in-deep-learning-b91d366e4862
[Accessed 22 March 2022].

Musyarofah, Schmidt, V. & Kada, M., 2019. *Object detection of aerial image using mask-region convolutional neural network (mask R-CNN).* West Java, IOP Publishing Ltd.

Peng, Z., Sun, B., Ali, K. & Saenko, K., 2015. *Learning Deep Object Detectors from 3D Models.* s.l., IEEE, pp. 1278-1286.

Petersen, K. & Wohlin, C., 2009. A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. *The Journal of systems and software,* 82(9), pp. 1479-1490.

Ramsundar, B. & Zadeh, R. B., 2018. *TensorFlow for Deep Learning.* 1st ed. Beijing: O'Reilly Media, Inc.

Raskar, P. S. & Shah, S. K., 2021. Real-time object-based video forgery detection using YOLO (V2). *Forensic science international,* 327(1), p. 110979.

Redmon, J., Divvala, S., Girshick, R. & Farhadi, A., 2016. *You Only Look Once: Unified, Real-Time Object Detection.* Las Vegas, IEEE.

Redmon, J., Divvala, S., Girshick, R. & Farhadi, A., 2016. *You Only Look Once: Unified, Real-Time Object Detection.* Las Vegas, IEEE.

Redmon, J. & Farhadi, A., 2017. *YOLO9000: Better, Faster, Stronger.* Honolulu, IEEE.

Redmon, J. & Farhadi, A., 2018. *YOLOv3: An Incremental Improvement,* s.l.: arXiv.

Reed, R. & Marks, R., 1999. Classical Optimization Techniques. In: *Supervised Learning in Feedforward Artificial Neural Networks.* s.l.:MIT Press, pp. 155-156.

Ren, S., He, K., Girshick, R. & Sun, J., 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE transactions on pattern analysis and machine intelligence,* 39(6), pp. 1137-1149.

Rubin, K. S., 2013. *Essential Scrum : a practical guide to the most popular Agile process..* London: Addison-Wesley.

Saha, S., 2018. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* [Online]
Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
[Accessed 3 April 2022].

Saxena, S., 2021. *Binary Cross Entropy/Log Loss for Binary Classification.* [Online]
Available at: https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/
[Accessed 10 December 2021].

Serrador, P. & Pinto, J. K., 2015. Does Agile work? — A quantitative analysis of agile project success. *International journal of project management,* 33(5), pp. 1040-1051.

Sewak, M. a., Karim, R. a. & Pujari, P., 2018. *Practical Convolutional Neural Networks.* 1st ed. s.l.:Packt Publishing.

Solai, P., 2018. *Convolutions and Backpropagations.* [Online]
Available at: https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c
[Accessed 14 April 2022].

Thesing, T., Feldmann, C. & Burchardt, M., 2021. Agile versus Waterfall Project Management: Decision Model for Selecting the Appropriate Approach to a Project. *Procedia Computer Science Volume 181,* pp. 746-756.

Viola, P. & Jones, M., 2001. *Rapid object detection using a boosted cascade of simple features.* Kauai, IEEE.

Wang, Y., Deng, W., Liu, Z. & Wang, J., 2019. Deep learning-based vehicle detection with synthetic image data. *IET intelligent transport systems,* 13(7), pp. 1097-1105.

Weidman, S., 2019. *Deep Learning from Scratch.* 1st ed. s.l.:O'Reilly Media, Inc.

Wicht, B., 2019. *mnist.* s.l.:s.n.

Wolfe, A., 2017. *How to Train Neural Networks With Backpropagation.* [Online]
Available at: https://blog.demofox.org/2017/03/09/how-to-train-neural-networks-with-

backpropagation/
[Accessed 14 April 2022].

Yadav, S., 2018. *Weight Initialization Techniques in Neural Networks.* [Online]
Available at: https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78
[Accessed 18 April 2022].

Ye, T., Zhang, X., Zhang, Y. & Liu, J., 2021. Railway Traffic Object Detection Using Differential Feature Fusion Convolution Neural Network. *IEEE transactions on intelligent transportation systems,* 22(3), pp. 1375-1387.

Zou, Z., Shi, Z., Guo, Y. & Ye, J., 2019. *Object Detection in 20 Years: A Survey.* s.l.:IEEE TPAMI.

Appendices

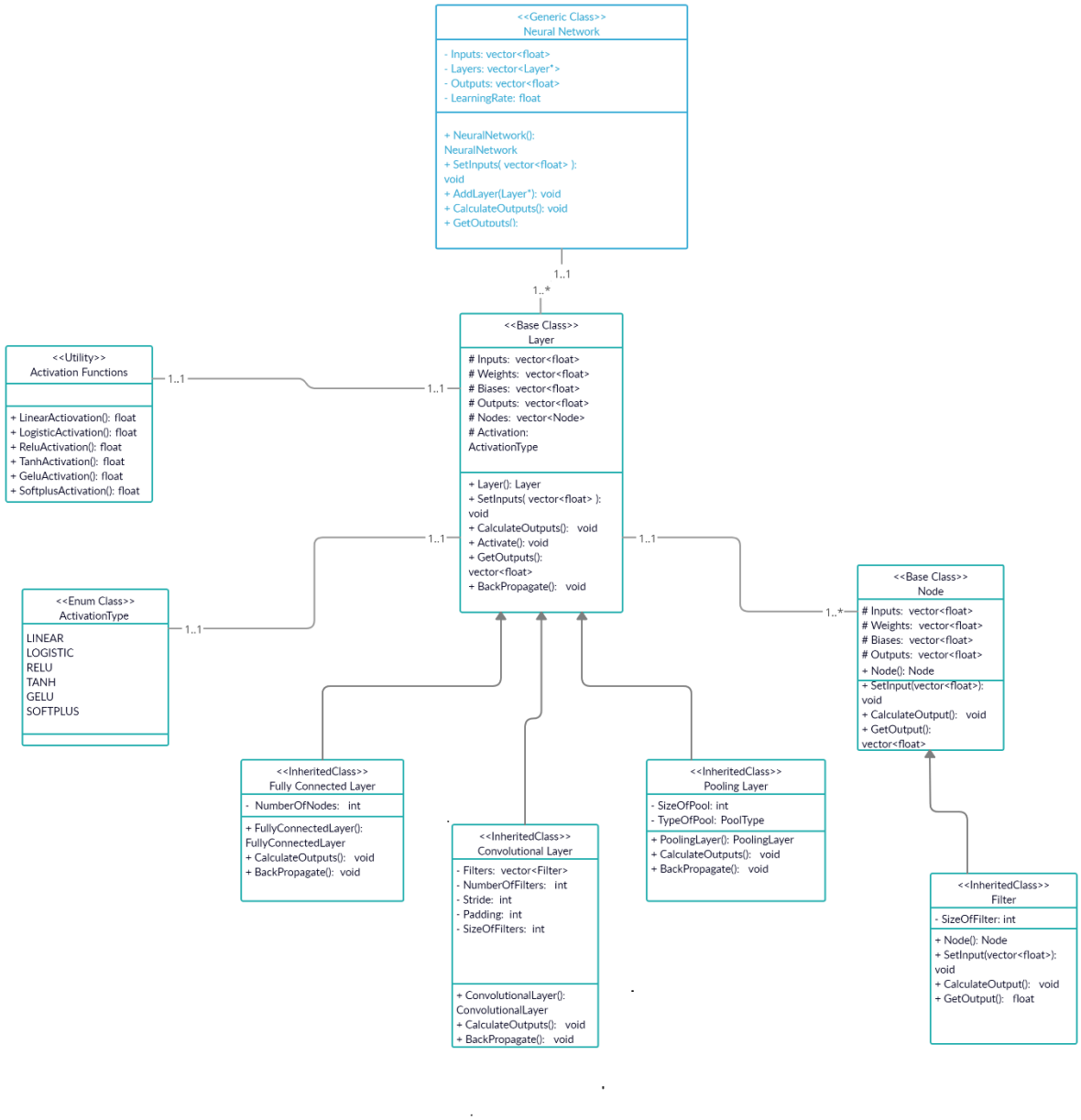
Appendix 1: Gantt Chart

Gantt project planner - Nasser Ksc

Appendix 2: Diary

Research and Artefact Diary.docx

Appendix 3: UML Class Diagram



Appendix 4: UML Sequence Diagram