School of Science and Technology

CSD3999
Software Development Project

Autumn/Winter term
2018/2019

Date: 26/04/2019
Supervisor: *Michael Heeney*
Student Name: *Christiaan Robert Michael Burrett-Bijlstra*
Student ID Number: *M00608813*
Campus: *Hendon*
Title: *Developing a software-based solution using artificial intelligence to effectively process and edit large video files to the requirements of a professional video editing company.*

School of Science and Technology

Student Name: *Christiaan Robert Michael Burrett-Bijlstra*

Student ID No: *M00608813*

Module number: *CSD3999*

I hereby confirm that the work presented here in this report and in all other associated material is wholly my own work. I confirm that the report has been submitted to TURNITIN and that the TURNITIN results are on CD attached to this report. I agree to assessment for plagiarism.

Signature:.....

Date: 26/04/2019

Abstract

In this report, we will be discussing a proposed software solution for a video editing company by the name of Regent's IT Solutions. Regent's IT Solutions requested a software-based solution for determining the momentum of a camera from it's recorded video alone. Regent's IT Solutions would use this information in conjunction with other custom software to automatically produce videos for the real estate industry. We discuss the research we have conducted within the field of computer vision. We will also discuss in detail the process we have gone through in developing the neural network based solution. The solution we have produced makes use of optical flow information gathered from video sequences that were provided to us by Regent's IT Solutions. We used this extracted optical flow information to distinguish a clear difference between when the camera is moving and when the camera is stationary. We then used this information to train our neural network to be able to accurately detect the difference between moving video sequences and stationary video sequences. We will go into further detail about the development of our neural network and the possible shortcomings our solution may have.

Contents page:

Describing the problem

An essential part of developing a successful and efficient solution is to understand the problem accurately and entirely. Regents IT Solutions (RITS) is a small company which has diversified into many fields of the technological industry, namely the field of video editing and processing [1]. The majority of their clients request services for basic video editing that does not require a high skill level or much creativity. These types of videos are repetitive and follow the same basic principles; the clients request that when the videographer is walking or moving the editor speeds up the video footage to 3 times the regular speed. When the videographer in question is shooting a panning or a still frame shot the editor does not adjust the video except for basic colour correction and lighting adjustments. The editing process for one of these videos can often be a long and tedious process taking between 30 to 40 minutes from start to completion. Resulting in the loss of time for Regents' editors and thus resulting in the loss of profit margins for RITS. Due to the repetitive nature of these video editing requests, we were intrigued to find a solution for the possible automation of this specific editing process. By automating this process it would allow RITS to reduce its number of professional video editors and their working hours, increasing their profit margins for this particular product dramatically.

Conceptualization of Solution

The initial conceptualization of the solution is the foundation of what will eventually be our finalized solution. We will start by first evaluating where in the whole process our solution will exist, what parts it will take over and what parts it might be able to take over in the future. By first conceptualizing it's most basic form we can subsequently extrapolate and optimize our solution. In figure 1.1 you will see a basic flow diagram of the process created with the information provided to us by RITS.

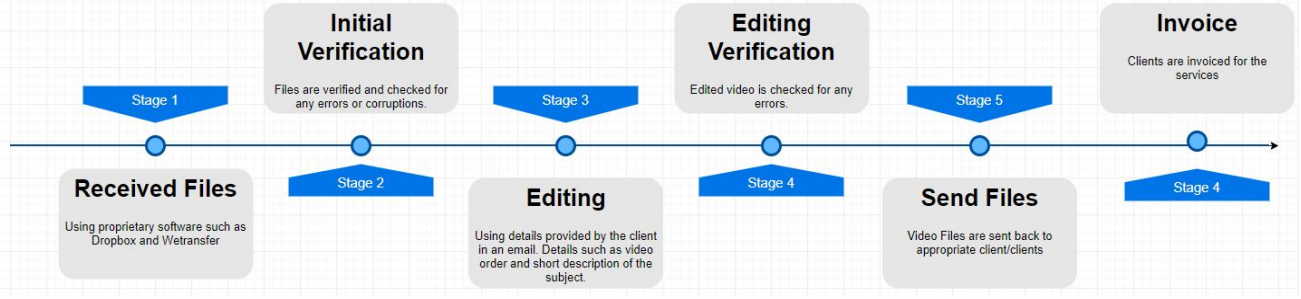


Figure 1 - Diagram of stages in RITS' video editing service

As you can see there are multiple steps that make up the production of the final product. RITS has made it clear to us that the editing done in stage 3 is by far the most time-intensive task. Therefore we will focus our solution around this stage with possible future extrapolation to the other stages. From the requirements provided to us by RITS we can already begin to speculate and conceptualize a possible solution. During the editing process, the professional video editor determines whether the videographer is walking or if he is panning. Therefore, we will need to be able to reproduce this observation within our software. I believe we can break our solution process down into 3 separate stages:

1. Gathering the data.
2. Interpreting the data.
3. Processing the video based on the data.

We currently believe that the most effective way of interpreting our data would be to use artificial intelligence. We will do this using pattern recognition. Pattern recognition is a separate area within the field of machine learning that focuses on the interpretation and recognition of specific patterns in data[13]. We believe that by gathering our data reliably and accurately we will be able to reproduce specific patterns in our data. For example, we might be able to recognize a specific pattern in our data which represents the forward momentum of the videographer. This interpreted data would then be passed to the processing stage of our software. This is where the video will be edited to the required specifications using our interpreted data.

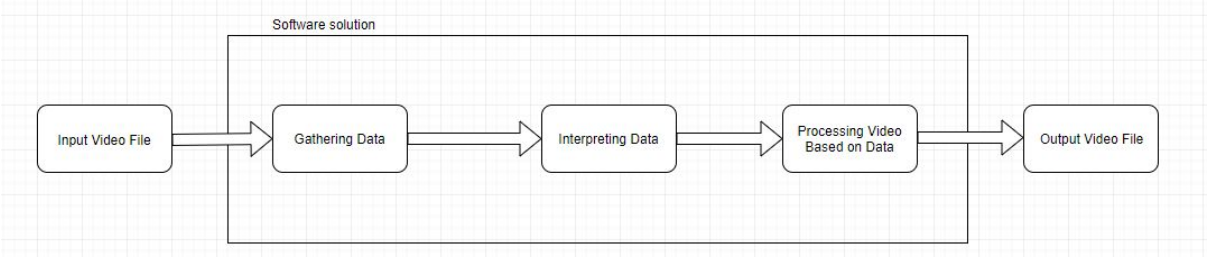


Figure 2 - Flow diagram of the proposed solution

Background and Literature Review

Due to the specificity of our project and the fact that our project is being developed for a specific use case namely the footage of a property. After conducting an extensive amount of research we have however discovered some academic and scientific documentation on subjects that could be relevant to our project. Below we will review the literature and discuss their possible relevance to our project.

Research is a vital aspect to any problem-solving exercise for without the appropriate knowledge we might not develop the most efficient and suitable solution. For our project, we will use an iterative cycle model. In figure 3 you can see a clear visual representation of the Iterative cycle model. The way we will make use of this model in our project, is to iterate through software versions and verify each of these iterations individually. We hope that by using this model we will achieve a more robust and effective software solution to our problem due to the fact that each iteration will be thoroughly verified and evaluated before a new iteration is developed, allowing us to learn from the flaws and progressions in previous iterations.

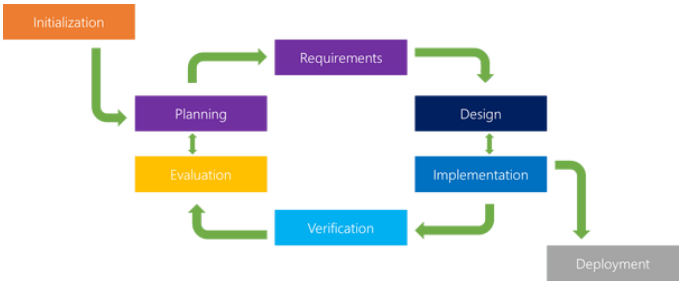


Figure 3 - Iterative Cycle Model

The research described below has been focused on the primary step of our solution, namely data gathering. The accuracy and consistency of the data we gather will absolutely vital to us achieving an effective solution. Therefore, it is of paramount importance that we exhaust all the possible methods for our data gathering. We have

already hypothesised making use of a variety of data gathering methods which would be normalized and compared to obtain a more accurate overall dataset.

Due to my previous intrigue and experiences in real-time visual applications, I was aware of a suitable library for our project called OpenCV, which is an abbreviation of Open Source Computer Vision Library. OpenCV is an open source library that specialises in computer vision and machine learning. It is designed to provide a common infrastructure for software related to computer vision and machine perception[2]. Due to its incredible assortment of algorithms, it can simplify the production of video analysis software dramatically. After conducting a comparative analysis between our problem and the algorithms provided to us by openCV we have concluded that there are a number of different algorithms that could be used to develop our solution.

OpenCV includes multiple corner detection algorithms the most relevant being the Shi-Tomasi corner detection algorithm[3] and the Harris corner detection algorithm. Corner detection is a widely used technique to find unique, easily identifiable and trackable points in an image. After considering both algorithms we have not yet concluded if either will be used in the production of our software. However, our initial research has shown the Shi-Tomasi corner detection algorithm to be more successful in achieving the desired results. This is also verified by a research paper published on this specific topic [5]. One must, however, consider the possible bias involved in their research due to the study being conducted by Jianbo Shi and Carlo Tomashi themselves. Furthermore, the differences measured in results between the two algorithms seem incredibly minute. We must therefore also consider the inevitable computational cost of both of the algorithms, to possibly find a balance between accuracy and computational cost. We have searched for studies conducted on the impact of these algorithms on computer hardware however to this date there have been no studies published on this topic. We will, therefore, have to measure the impact on computer hardware and the computational costs of these algorithms ourselves.

After conducting further research in the field of visual data analysis I have found that the concept of optical flow could be relevant to the development of our solution. Optical flow is a proposed pattern of objects, surfaces and edges in a visual scene brought about by the relative motion between the observer and a visual scene. The concept was first introduced by James J. Gibson in 1940. Since then researchers have conducted further studies showcasing the apparent importance of optical flow. It has been hypothesised that humans can perceive the direction of self-motion using optical flow. During a study in 1988 conducted by Warren, W. H. and Hannon, D. J. they found this to be true [6].

So one could reasonably assume that by using the Shi-Tomasi corner detection algorithm and calculating the optical flow over a duration of time we should be able to gather data that will conclude the direction of movement of the videographer. This is exactly what the Lucas-Kanade Method does. The Lucas-Kanade Method analyses the optical flow of the given video by receives a set of points and comparatively analysing them to a previous set of points[7]. So by combining these two algorithms, we should theoretically be able to accurately track the videographers movements. However, there is a caveat in our previous assumption; namely that the majority of research on these methods have only been conducted with a stationary camera tracking moving objects within their visual frame. Therefore we need to conduct further research and testing to conclude that the Lucas-Kanade Method is an effective method for our solution. In figure 4 we are showing casing the results of the first iteration of our prototype, we will reference this version as Prototype version 1.0. You can clearly see that the optical flow has been accurately represented using the Lucas-Kanade Method in conjunction with the Shi-Tomasi corner detection algorithm.

Figure 4 - Optical flow in prototype 1.0

As we stated previously in our report we plan to make use of artificial intelligence within our system. Artificial intelligence is currently at the forefront of innovation within the technological industry. We have looked at multiple different ways we could use artificial intelligence to effectively interpret our data. A method that has shown it could possibly be useful in our situation, is the process of supervised learning[13]. This process involves the use of a large amount of predefined test data and manually labelling specific parts of that data as either “Momentum”, “Panning” or “Still-shot”. The A.I. will then attempt to recognize specific patterns within these labelled areas of the data. By processing an increasing amount of predefined labelled data the Pattern recognition system it will become increasingly accurate. In figure 5 you can see a graph which represents the optical flow over a short duration of a video file provided to us by RITS. As we stated previously, these results are gathered using our current prototype (version 1.0) uses the Lucas-Kanade Method in conjunction with the Shi-Tomasi corner detection algorithm to give us these results. As you might be able to see we can visually identify the parts where the videographer is moving by the large delta in optical flow.

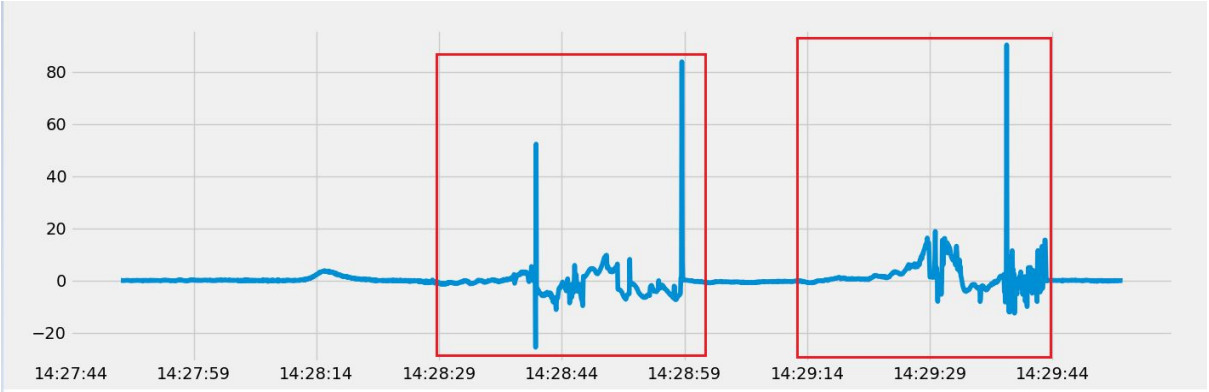


Figure 5 - Change of optical flow graph: X-Axis represents time, Y-Axis represents change in optical flow. Red areas indicated areas where we have manually observed momentum of the videographer from the video.

We have also conducted research in the field of video processing hardware and its impact on processing speeds. It is a well-established fact that video processing requires a large amount of computational power[8]. Formal research into this field shows us there is a clear time-based advantage in using a high clock speed CPU over a lower clock speed CPU due to the calculation complexities involved[8]. Additional research has shown us that other components in our system will have an effect on our processing performance. Figure 5 shows us a considerable performance increase in using high-end GPU's[9]. However, there are other factors to consider besides performance before concluding on our hardware. We must consider the volume of videos to be processed and the optimal desired time for each video to be processed. Based on these variables we could possibly conduct an investigation to establish which hardware has the optimal ratio between price power efficiency and processing power. We must consider this to be a long-term solution for RITS, therefore, if they were to invest in power inefficient hardware it could lead to unnecessary costs. Furthermore, there is a commonly occurring trend in hardware which shows that there is an extreme exponential rate between price and performance. So we must also evaluate whether acquiring the fastest possible hardware is the most cost-effective strategy for RITS.

Feature Tracking

We are using OpenCV's "good features to track" function to find easily definable features in our frame. This will allow us to calculate the difference in location of these features between frames and thus find the optical flow. OpenCV has many different functions that use different algorithms to find these features. Therefore we decided research was required to determine whether or not "good features to track" is the most applicable function for our project. We came across a paper written by Jianbo Shi and Carlo Tomasi. The paper is titled "Good Features to Track"[14]. This paper discussed the algorithm used by the function in openCV that was developed by Jianbo Shi and Carlo Tomasi. The report discussed how the Shit-Tomasi algorithm introduces a monitoring method on top of pre-existing corner detection algorithms. Their paper describes the concept of "good features" and "bad features" that regular feature tracking algorithms employ. Ordinarily, a feature tracking algorithm would determine a "good feature" that it should maintain tracking by its dissimilarity to the same feature in the previous frame. If a dissimilarity is too high then the feature will no longer be tracked. However, they describe a method that would involve accounting for deformations of each specific feature which in turn would conclude that even though the feature has deformed due to changes in the camera angle it still remains a "good feature".

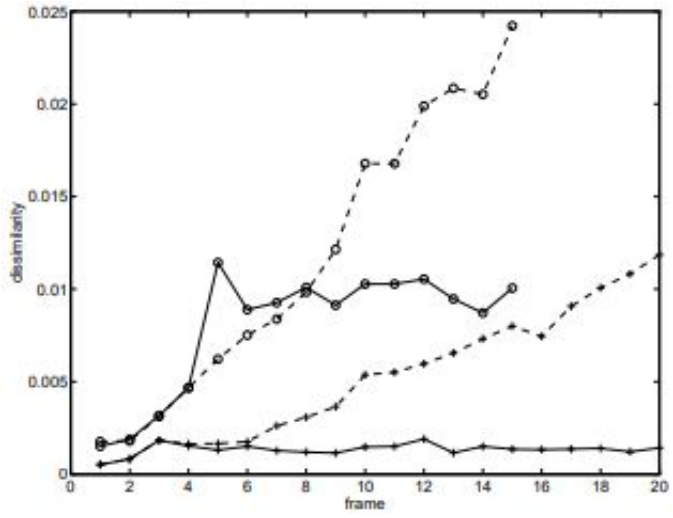


Figure 6 - Graph of dissimilarity per frame from the "Good Features to Track report"[14]

In figure 6 the dashed line with crosses is the line which represents the dissimilarity of the features per frame without accounting for deformation. The solid line with crosses is the dissimilarity whilst accounting for deformation. As you can clearly see the dissimilarity remains relatively constant when the deformation is accounted for and thus the algorithm would continue to recognize this as a good feature to track. The paper then continues to describe further experiments they have conducted on sample footage.

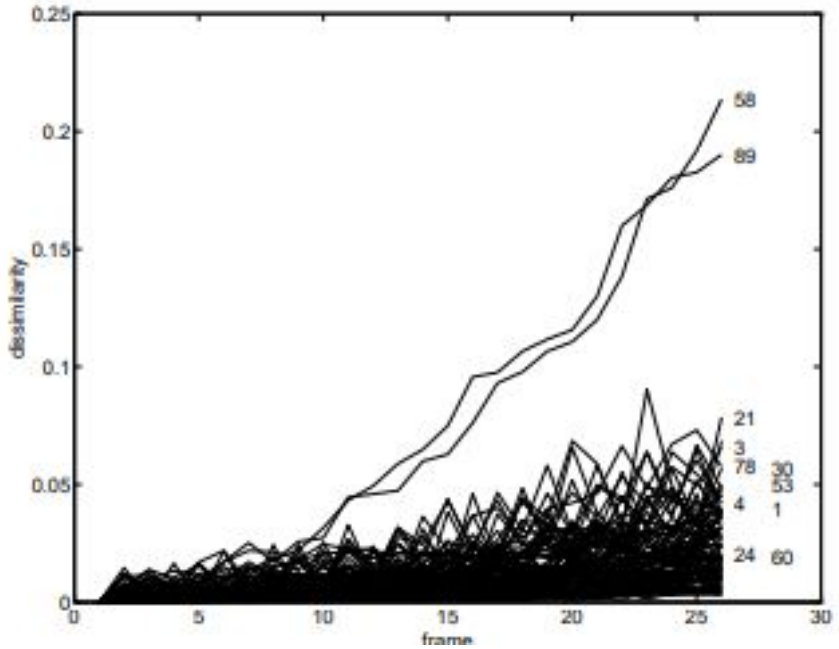


Figure 7 - Dissimilarity on sample footage [14]

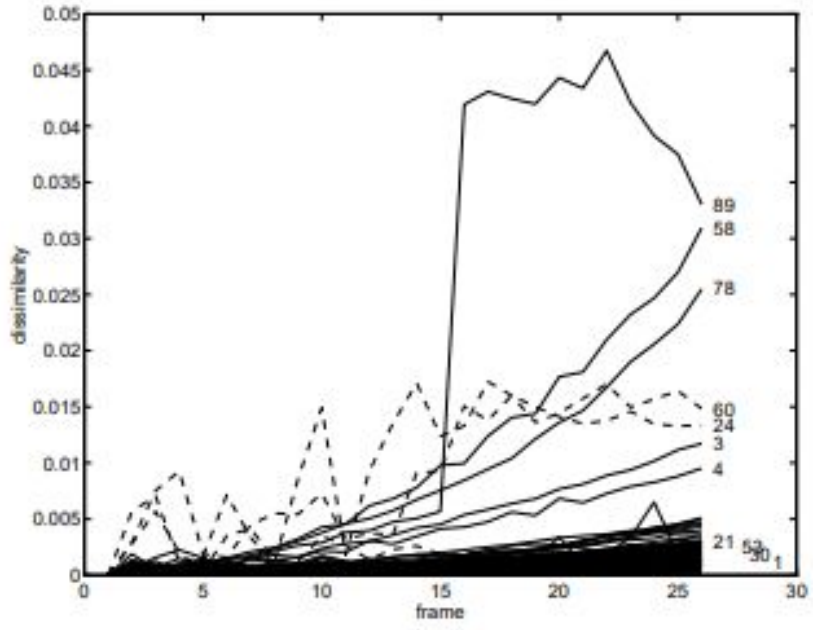


Figure 8 - Dissimilarity on sample footage [14]

Figure 7 and 8 show us the dissimilarity of each feature per frame. Both figures\ were conducted on the same sample footage with the only differentiating factor being that figure 8 has implemented the monitoring functionality proposed by the authors into the calculating algorithm of dissimilarity. As you can see the results from figure 7

display a far more erratic pattern and therefore it might be hard to differentiate between the good features to track and the bad features to stop tracking. However, when the algorithm takes into account the deformation and distortion movement of the camera's perspective can cause we see a far more clear picture. We can therefore easily identify the features which we should not continue to track, resulting in a far more accurate feature set. They go on to conclude that although their algorithm will not always be able to remove all of the bad features it will certainly reduce the number of bad features in the overall calculation. A logical concern that might be theorised however is the increased computational complexity of the algorithm when implementing the proposed monitoring equation into the algorithm. However, the Authors claim that the monitoring functionality is "inexpensive and sound". Taking this into account we concluded that this algorithm would be able to reliably determine the good features to track in our project.

Optical Flow Calculations

As we discussed in our initial report the Lucas Kanade method is an algorithm that is commonly used to calculate the overall optical flow of a video sample. It does this by running its calculation on a set of deliverable features for each frame. In the paper titled "Optical Flow Measurement using Lucas Kanade Method" [15], Authors Dhara Patel and Saurabh Upadhyay discuss the relevance and accuracy of the Lucas Kanade method for estimating motion based on optical flow. In their paper, they discuss different methods and approaches for the calculation of optical flow. It approaches the overall optical flow calculation with the assumption that optical flow is calculated not over the entire image frame but purely in the vicinity of the pixel which is in question. This allows the Lucas Kanade method to be extremely efficient in calculation times whilst still maintaining a high level of accuracy. The Authors conclude that due to its fast computation time and accuracy they would use the Lucas Kanade method in feature motion estimation analysis. We, therefore, concluded that it would be appropriate for us to use it in our project. By supplying the Lucas Kanade function with accuracy "good features" by using the previously

discussed Good Features to Track algorithm we feel confident we can get accurate optical flow information and thus eventually estimate the motion of the videographer.

Further related research

We also continued our research into the subjects that could be considered somewhat significant for us. Within the field of computer vision, there is a subject named “egomotion”. This word, of course, derived from the Latin words “*ego*” and “*motionem*”. “*Ego*” defines the concept of oneself and “*motionem*” describes the act of moving. Therefor egomotion describes the concept of self-motion. Egomotion is also often referred to as Visual odometry. Interestingly enough during our research, we have found that often computer systems and methods will be derived from the functionality that our bodies have already implemented. For example in the paper “The Representation of Egomotion in the Human Brain” by Authors Matthew B. Wall and Andrew T. Smith [16]. They state how significant the brain's ability to detect egomotion is. They also describe the retinas ability to generate an expanding pattern of optical flow. They go on to state there are specific areas within our brain that they have managed to pinpoint that actively respond to optical flow stimulus. Although this information might not be directly relevant to our project, I believe that it serves as a proof of concept. The reason I believe this serves as a proof of concept is due to the fact that in computer scientists a large portion of our advanced concepts and algorithms are inspired by the human body. Some of the most technologically advanced topics in computer science employ methods used in the body. For example Artificial intelligence and the use of neural networks[18] and cybersecurity and the use of immune systems[18].

Further research into robotic visual odometry has also yielded some relevant findings. In the paper “The Use of Optical Flow for Road Navigation” there is an analysis of the use of accurate optical flow data to possibly be used for autonomous road navigation in the future [17]. The paper references the fact that there are currently plenty of vehicles in use that make use of a variety of sensory information

and video analysis techniques including optical flow analysis to help autonomously navigate the road. One of the largest problems introduced by the paper is the fact that on moving vehicles there are a lot of undesired vibrations as it drives over rough or uneven road surfaces. This results in insufficient optical flow information being retrieved from vehicle-mounted cameras. This is a very relevant finding and we, therefore, must consider it when further building our system. As we know our videographer will be moving, this will cause vibrations in the camera and thus result in insufficient and accurate optical flow data. We must look at our system and account for these inevitable vibrations in our footage. However, there are many proposed methods we could use to accurately measure our optical flow data, such as the low complexity algorithm proposed by Florian Raudies and Heiko Neumann [19].

Research into Neural Networks and visual odometry

During the course of our project, we have also conducted an extensive amount of research into the development of convolutional neural networks for use in determining the optical flow of a sequence of images. There are numerous studies and papers that have been conducted in this area and they could have possible implementations in our future prototypes. What we found in most of these research papers however was the fact that their evaluation methods were conducted with the KITTI dataset [25]. This dataset has a vast number of recorded information from a car that had been outfitted with a variety of sensors. In figure 9 you can observe the car outfitted with all of the sensors that have been used to produce this dataset. It is considered a standard benchmark for evaluating monocular and stereoscopic odometry or ego-motion.

Figure 9. Sensor car used to record information for the KITTI dataset [25].

One of the main reasons for the use of this dataset is because of the current trend in autonomous vehicles. As we see companies such as Tesla innovate in the field of autonomous vehicles it has triggered a vast amount of research to be conducted in this field. Initially, this seemed promising due to the fact that accurate ego-motion would be extremely relevant to our project. However, we discovered after extensive testing that most of these methods rely heavily on the assumption that forward momentum is extremely dominant. This assumption would be accurate in a situation such as a vehicle movement, but in our case, the assumption does not always hold. What we found is that some of the algorithms that we tested predicted that there was forward motion from simple camera rotations. I believe these results were produced because on the KITTI dataset the cameras were positioned in a stationary position meaning the cameras themselves were unable to rotate independently from the car. This meant that within the dataset the only way for the camera to have rotation was for the car to have rotation and the vehicles used in the KITTI dataset were unable to rotate on their own access meaning there always had to be a form of translation present where any rotation occurred.

A major advantage imposed by a deep neural network in the analysis of our videos is the fact that it would be able to interpret information from almost any camera without the need for specific calibration of the program. In other conventional methods of visual monocular odometry camera calibration can be critical in achieving accurate results. If we were to use a neural network in our approach to solving this

problem we would have the possibility of expanding our system to be able to interpret alternative types of videos recorded on other types of cameras.

Requirements

After conducting our research we have been able to conduct a requisite analysis. The analysis concluded that because of the considerable computational power needed to process video, it would be beneficial to make use of powerful hardware[8]. As we can see in figure 10 we can see that NVIDIA' Titan V is most effective in processing video using the widely Adobe software Premiere Pro CC 2017. For us to conduct rapid prototyping on our software it would be extremely efficient to have sufficient computational power using equivalent GPU's and CPU's shown in figure 10, thus being able to evaluate and further develop our software with greater time efficiency.

Although we can conclude that our software would benefit from the use of high-performance hardware it is not a necessity. Due to our project being able to make use of multiple threads within the CPU we are able to process at an average rate of 14FPS with our current methods and hardware. Therefore if we take the average video length that RITS has provided us which is around 3 minutes of raw footage we can estimate that we will be able to process on average 1 video every 5 minutes on average hardware. With the current scale of the project, I believe that is an acceptable amount of time for video processing however this could be optimised in the future if required.

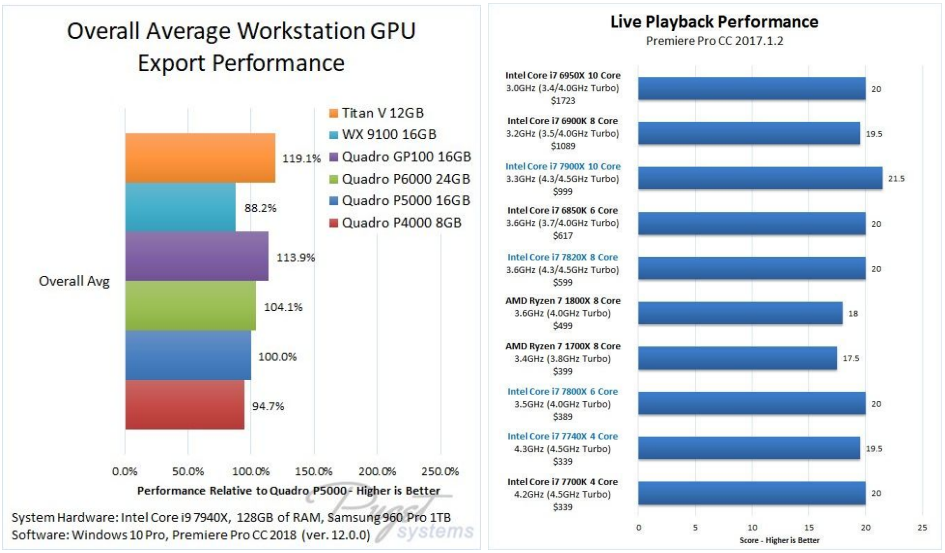


Figure 10 - GPU and CPU benchmarks in video processing using Premiere Pro CC 2017[9][10]

RITS has specified that this requires a form of output with timestamp information on which parts of the video are moving and which are stationary they have stated that they are relatively flexible with how we will represent the output information as a secondary video editing system will be built around our software to further process the videos. Therefore the only requirements that RITS has specified are that our system should be able to take in any file with an MP4 format and that our system outputs timestamp information of when the camera is stationary and when it is moving.

Analysis and design

It is critical that we consider the design of our project and the reasoning behind some of our design choices. Due to the specificity of our project, we do not have a direct competitor to analyse and perhaps base our design choices of. We must, therefore, consider how we wish to present data to users of our software. RITS has specified that they will be looking to deploy a specialised video editing solution after we have completed our project. This means that RITS will use our software in conjunction with other software and case-specific systems.

With the main user of our software being RITS we have been able to conclude the input of the software and the output required of the software. They intend to input files in an MP4 file format and they would like to receive timestamp information about movement in the video. We have considered all design approaches to our system and we have been able to determine that one of the most basic ways to proceed is in a user request format.

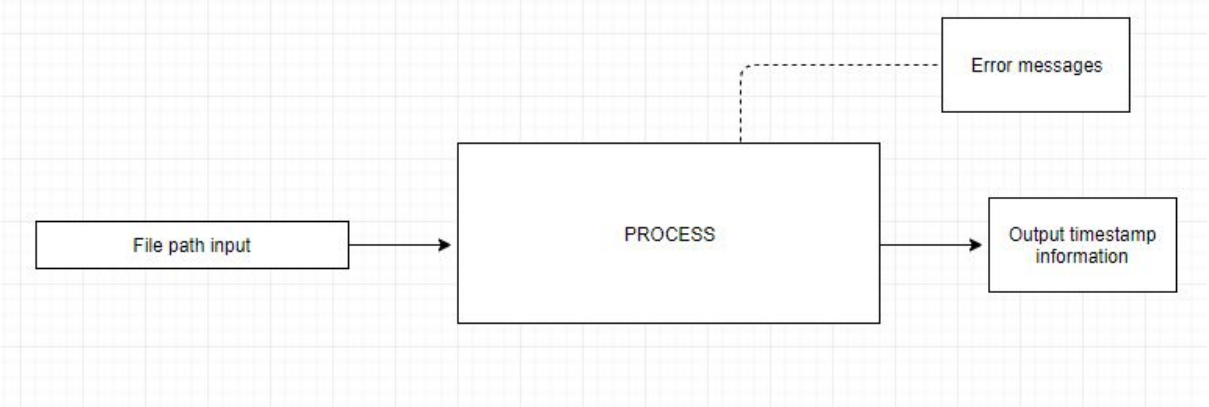


Figure 11. System design concept

In figure 11 you can observe our design concept for RITS. As our software is being developed for professional usage we want our system to be as user-friendly as possible. Therefore we have developed a relatively simple system that should be able to perform its intended purpose with very little maintenance and very little understanding of how our system functions. The system RITS intends to develop should simply provide our software with the file path of the selected videos. These videos will then be processed by our system and when our system has completed its analysis it will output the sections of the video which are considered to be moving and the sections which are considered to be stationary. Although we are not sure of the exact format our output will be, it can easily be changed to any required format. We have advised RITS that perhaps a JSON format would be advantageous as it is widely accepted and easy to interpret. RITS has also informed us that it will employ a system administrator who will handle any potential errors that come through. This means that any error messages produced by our system should not be sent to the user but instead to the system administrator.

We were also tasked with designing an appropriate neural network(NN) for our project. Designing a neural network can be a complicated process as there are a lot of variables that can be adjusted to produce better results. Our software currently makes use of a sequential neural network. This is the structure for our NN:

- Input layer consisting of 30 nodes
- Hidden layer consisting of 60 nodes
- Hidden layer consisting of 60 nodes
- Dropout layer 30%
- Hidden layer consisting of 60 nodes
- Dropout layer 30%
- Hidden layer consisting of 30 nodes
- Output layer consisting of 1 node

```
59  # input layer
60  model.add(Dense(60, input_shape=(30,), activation="relu"))
61
62  # hidden layers
63  model.add(Dense(60, activation="relu"))
64  model.add(Dropout(0.3))
65  model.add(Dense(60, activation="relu"))
66  model.add(Dropout(0.3))
67  model.add(Dense(30, activation="relu"))
68
69  # output layer
70  model.add(Dense(1, activation="softmax"))
71
```

Figure 12. Code of the construction of our neural network

As you can see in figure 12 it is extremely easy to construct the foundation for our NN with the use of Keras. Most of our functions use the “relu” activation function this is also known as the rectifier function.

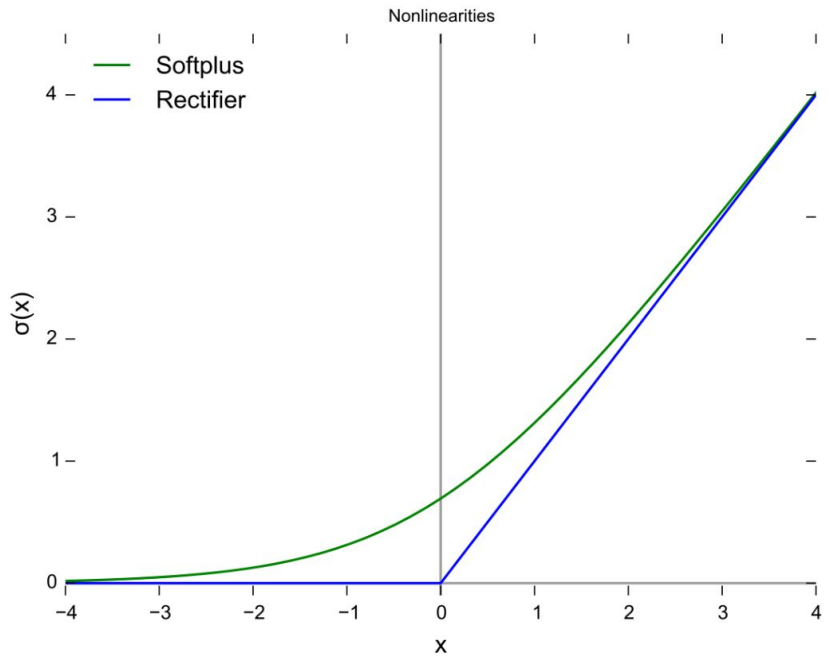


Figure 13. Rectifier function graph

As you can observe in figure 13 the rectifier functions works on the premise that if x is less than 0 it equates to 0 if the variable X is greater than 0 it remains X. This means that the range of the rectifier activation function is $[0, \infty]$. This allows us to avoid gradient vanishing.

After completing the design of our NN we needed to determine how our NN would train in relation to the data. There are three main variables in altering the training behaviour of our neural network:

1. Batch size
2. Epochs
3. Learning rate

The batch size of our neural network is the number of training samples our neural network would use in each training iteration. We concluded that a batch size of 50 was appropriate for our system. An epoch is completed when the neural network has passed over all the training data. We found that 500 epochs were enough for our neural network to achieve the maximum possible accuracy. The learning rate of our

neural network determines how much our neural network adjusts after each backpropagation. We found that a learning rate of 0.01 was suitable for our neural network. We use Backpropagation to train our neural network. This process involves adjusting the weights of each neuron based on the error value we receive after each backpropagation. Because of the fact that our training data is labelled our neural network is able to calculate how inaccurate it was in determining the value of our training data. Because it knows the percentage of error it is able to adjust its weights accordingly.

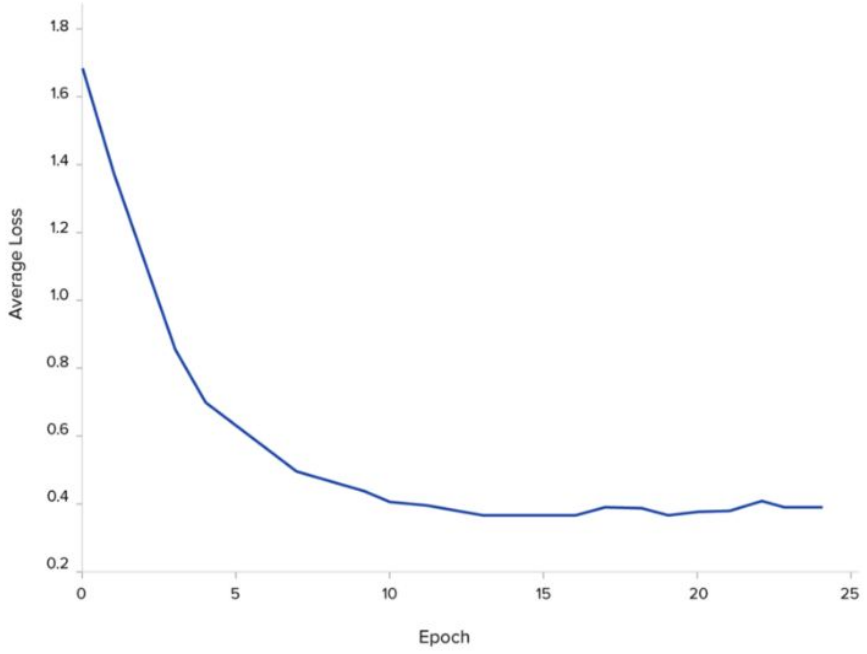


Figure 14. Graph of average loss over epochs

In figure 14 you can observe a good example of how a neural network adjusts its weights using backpropagation to minimize the average loss of its results. The learning rate determines how much we allow our neural network to adjust after each backpropagation. To understand why we do not allow our neural network to train at a rapid pace we must look at stochastic gradient descent.

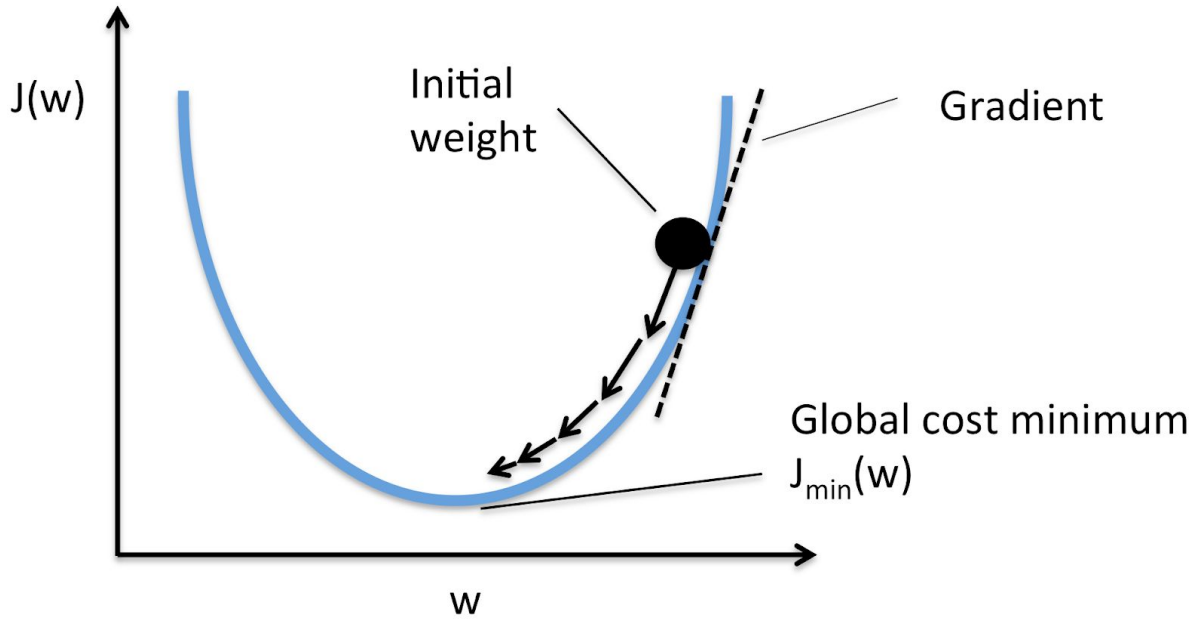


Figure 15. Basic graph of stochastic gradient descent.

In figure 15 you can observe what is called a 2d gradient descent graph. In the basic graph, W is the weight of a neuron and J is error percentage. Each time our neural network performs a backpropagation it knows which direction to change its weights towards. In this graph, the initial weights are quite high and thus the initial errors are also high. Our neural network knows that it should decrease the weight of this neuron to achieve a lower percentage loss. Our neural network does not, however, know by exactly how much it should adjust this weight just the direction in which it should adjust it. If we were to allow our neural network to make a large number of changes to the weights after each backpropagation there could be the possibility of constantly overshooting the global cost minimum. This is the point at which our neural network would achieve maximum accuracy. We concluded that a learning rate of 0.01 allowed us to achieve the global cost minimum relatively quickly with good accuracy. With our neural network, we also wanted to make sure we avoided overfitting our model. Overfitting occurs when our model becomes too accurate at predicting the training data with the result of not being able to accurately predict data from outside of the training set. We can determine if our model is overfitting by using a validation set. This validation set is a similar dataset to the training dataset however it is not used to train our neural network but instead it functions as a test to

see how accurate our neural network can predict data that it has not been trained on. Overfitting is incredibly common when designing neural networks and the most common way to solve it is by acquiring a larger training dataset. Another way to avoid overfitting is by using dropout layers. Dropout layers are added to a model and this layer will randomly select a percentage of nodes that it will ignore. This has the result of increasing a neural networks generalisability. As you can see in our neural network design we have added two dropout layers that dropout 30% of the nodes in that layer.

Implementation and testing

Our goal is to develop the most effective solution for the aforementioned problem. We must, therefore, analyse our solution to determine whether it is effective. However, since our project is so specific by nature, there are no predefined benchmarks or datasets. Therefore, the only way to evaluate our software' effectiveness and efficiency is by comparatively analysing it to our previous iterations. It is, therefore, required of us to maintain an extensive record of all our previous software iterations and their corresponding achieved results. For this is the only way we will be able to evaluate our final software and the progressions we have made.

We will need to conduct appropriate data-interpretation methods to properly evaluate our data. We can reasonably assume there are three most appropriate ways in which we can comparatively evaluate our software. The first being the observed accuracy of our results. This comparison would entail us looking solely at the trueness of our results. The second would be a time-based evaluation. In this comparison, we would observe the time taken to achieve the final result of each iteration and prototype. Since we aim to run these tests on a standardised testbench we can confidently isolate the cause and effect on our results. We can therefore reasonably conclude that the time taken for the operation to be completed will

directly correlate to the computational complexity of our software. Thus we will not need to measure the computational complexity of each iteration. The third way, we assume will be the most appropriate and effective evaluation. It will entail a comparative analysis of the accuracy and the time taken to complete the process. The reasoning for this assumption is on the basis that Regents IT Solutions requires the most effective solution and not necessarily the most accurate nor the quickest. We must stipulate that it is proper to conduct in an observational analysis to measure precision in our results. We will certainly conduct precisional observations by conducting repeat tests on the same software. However, since we expect our current solutional methods to involve algorithmic approaches that will have negligible precisional variation, we do not believe that it will contribute to a meaningful evaluation. However, as stated previously this could be subject to change if we were to observe a fluctuation in our precisional measurements.

Since our project' goal is to develop a solution for a professional video editing company we have other evaluation methods at our disposal. As stated at the start of this report our goal is to develop a software solution that will increase RITS' overall profit margin. We would request that RITS provides us with the necessary data to evaluate the change in profit margin for their video editing service. Considering that they have already disclosed that the editing stage of the service is the most cost-intensive, we should expect to see a considerable increase in profit margin. The reliability and accuracy of our system will greatly affect this. If our system requires a video editor to make corrections frequently the cost per video will increase. Therefore we will also evaluate our project based on the percentage increase in profit margin.

Data gathering is an incredibly important aspect of our project. Due to the fact we plan to use a neural network to analyse our data it is paramount that our data is both consistent and accurate. To produce our initial dataset we decided to gather optical flow information from our video. We did this using OpenCV's goodFeaturesToTrack method to determine the corners of the image to track and then determining the optical flow based on those features with Lukas Kanades algorithm within the

OpenCV library. Initially, we felt very confident with the data we were receiving we were determining the overall optical flow of the image and returning the change in positions of the corners. We decided to compile our dataset using the average deltas in X and Y coordinates of all the points for each frame. However, after extensive testing with this method, we swiftly came to the conclusion that by including the X coordinates in our dataset we were increasing both the complexity of our dataset and the size of our dataset dramatically. We realized that the Delta in the average X coordinates between frames in our video analysis would not be a differentiating factor when trying to determine whether a shot was a stationary or moving shot. We swiftly resolved this error by removing all the delta X coordinates from our dataset. Figure 16 shows a graph which was produced in the initial testing of our dataset. We have selected 3 random 30 frame samples to show a plot on the graph. There are 30 frames on the X-axis of the graph each represents a frame in the sample. The lines are plotted by plotting the Delta Y between each frame on the Y-axis. As you can clearly see there is a distinguishable difference between moving shots and stationary shots.

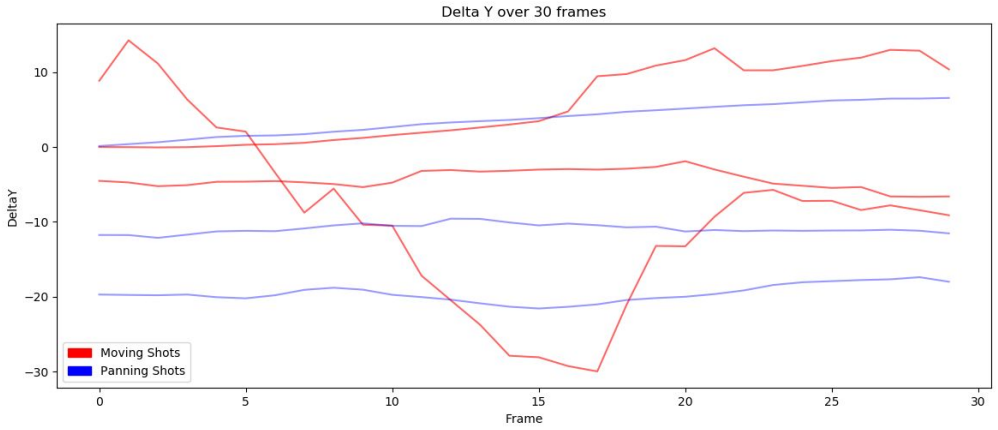


Figure 16 - a graph showing the 30 frames and the average Delta Y relative to the previous frame.

Once we had gathered our initial samples and we were satisfied with our results we decided to scale up our dataset dramatically by gathering 3000 seconds worth of

footage of both stationary shots and panning shots. By increasing our data size this dramatically we soon came to the conclusion that because the data analysis time was increased from 1 minute to over 3 hours, we need a robust way to permanently store our dataset on the hard drive of our computer permanently rather than store it on the ram periodically per instance of our python program. Once we had processed all our footage and stored it we again plotted a graph this is displayed in figure 17.

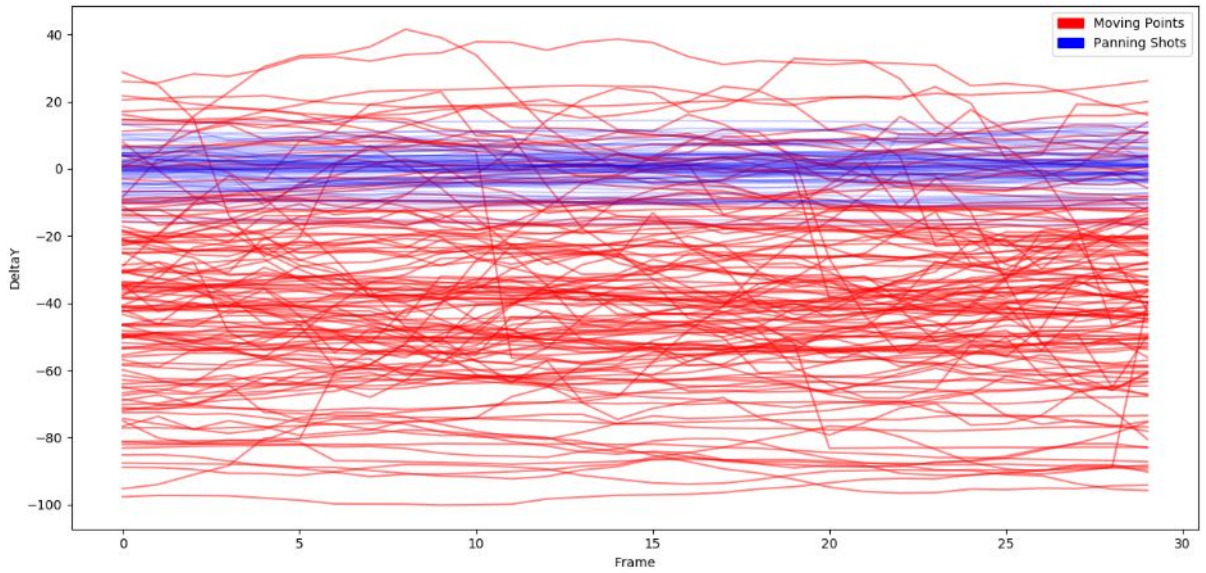


Figure 17 - an Increased dataset on the same graph

As you can see even on an incredibly large dataset there is still a clear and observable differential between moving and panning shots. Due to the fact that the camera moves up and down whilst the videographer is moving the change in Delta Y is increased dramatically compared to when the videographer is filming in a stationary position.

Optimization of Data Gathering

Because we had again achieved our goal of showing the observable trend in a large dataset we concluded that to conduct further testing we needed to optimize our video analysis code. The reason for this was because it was simply suboptimal for our

analysis to take over 6 hours to complete. We determined that because python by default uses a single CPU thread to compute the program we could increase our average frame per second (FPS) of video analysis by running our image conversion and gathering on a separate thread that the initial thread that is being used to make the optical flow computations. We decided on a queue based method, as we believed this would be the most optimal for our program. Figure 18 Shows the layout for our program. As you can see as the program is initiated there would be an immediate divide in operations between the two threads. As the first thread starts processing the video into individual frames and building the queue for the second thread to make use of. This would, therefore, reduce the load on the second thread dramatically. As both threads would start processing concurrently there was a chance for the second thread to want to pull from the queue without the precence of any frames in the queue as none had been processed yet. This would result in an error, therefore, we concluded that it was beneficial to have an initial delay in operations of our second thread. Allowing the first thread to build a sufficiently sized queue before the optical flow calculations began.

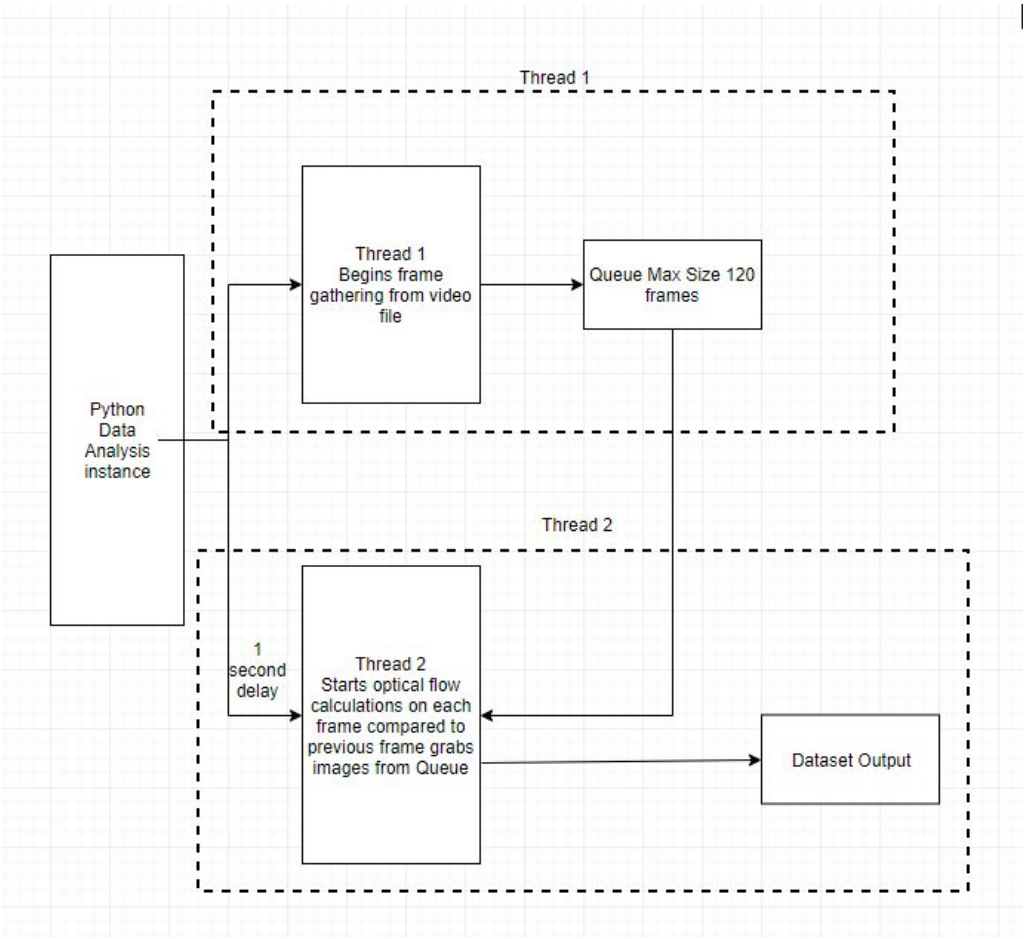
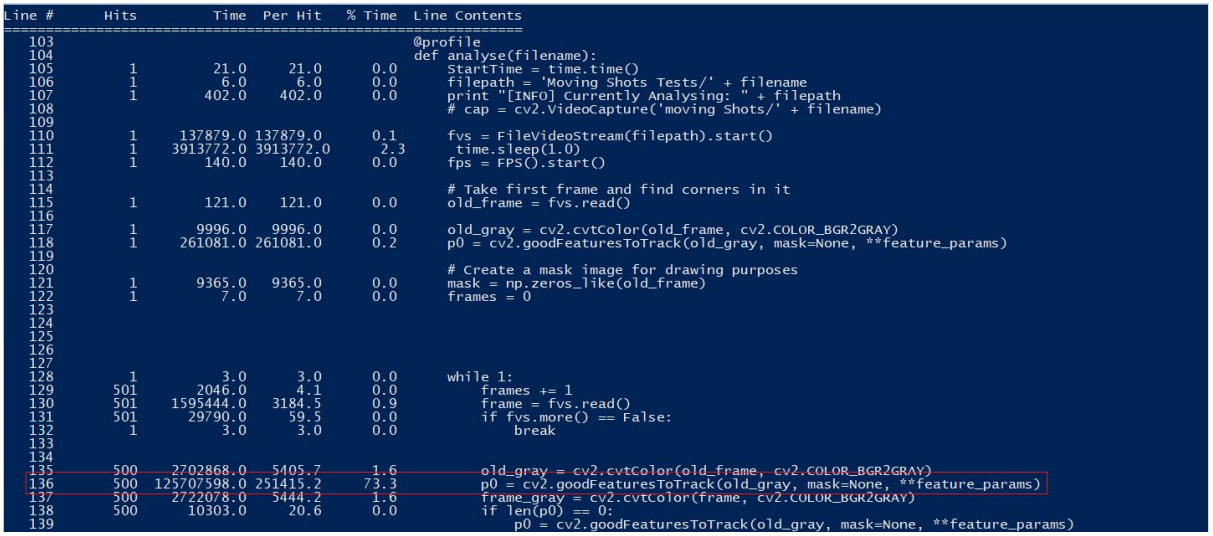


Figure 18 - Layout of our program

We initially thought that this would increase our processing FPS rate by over 140%. Our average processing FPS before splitting the program into separate threads was around 5 FPS this meant that to process 1 minute of footage it would take 12 minutes. We estimated that after our optimization we would increase our average processing FPS to 14 FPS. However what we found was that our average processing speed to increased significantly less than we expected, to only 8.5FPS. Although this was still a 70% increase in efficiency it was less than we had expected. Because of this we decided to conduct further research into why this might be the case. The most effective way to analyse our code was by running a profiler on it. By doing so we discovered that there was one particular line that was consuming the most of the processing time. As you can see in Figure 19 line 136 has taken up 73% of all the processing time of the most computational intensive function of our program.



```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   103                                           @profile
   104                                           def analyse(filename):
   105         1         21.0     21.0      0.0      StartTime = time.time()
   106         1          6.0      6.0      0.0      filepath = 'Moving Shots Tests/' + filename
   107         1        402.0    402.0      0.0      print "[INFO] Currently Analysing: " + filepath
   108                                               # cap = cv2.VideoCapture('moving Shots/' + filename)
   109
   110         1     137879.0 137879.0      0.1      fvs = FileVideoStream(filepath).start()
   111         1    3913772.0 3913772.0      2.3       time.sleep(1.0)
   112         1        140.0    140.0      0.0      fps = FPS().start()
   113
   114                                               # Take first frame and find corners in it
   115         1        121.0    121.0      0.0      old_frame = fvs.read()
   116
   117         1       9996.0   9996.0      0.0      old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
   118         1     261081.0 261081.0      0.2      p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
   119
   120                                               # Create a mask image for drawing purposes
   121         1       9365.0   9365.0      0.0      mask = np.zeros_like(old_frame)
   122         1          7.0      7.0      0.0      frames = 0
   123
   124
   125
   126
   127
   128         1          3.0      3.0      0.0      while 1:
   129       501       2046.0      4.1      0.0          frames += 1
   130       501    1595444.0   3184.5      0.9          frame = fvs.read()
   131       501      29790.0     59.5      0.0          if fvs.more() == False:
   132         1          3.0      3.0      0.0              break
   133
   134
   135       500    2702868.0   5405.7      1.6          old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
   136       500  125707598.0 251415.2     73.3          p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
   137       500    2722078.0   5444.2      1.6          frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
   138       500      10303.0     20.6      0.0          if len(p0) == 0:
   139                                                       p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
```

Figure 19 - Line Profiler of data gathering code

The function of line 136 is to find an array of good features to track within an image. After conducting research into the exact algorithm used by OpenCV to find these points. We concluded that the processing time of that function was due to the fact it was running an inefficient operation for CPU's to calculate. We believe that because the function is running a calculation that is graphically intense it could be beneficial to calculate this function in the GPU. However, we must conduct further analysis and testing to conclude this theory.

Data Formatting

Due to the fact aim to use a neural network to distinguish between stationary video and moving video footage, there are some steps we need to take before our data is ready to be analysed. Although we have not yet started development on the neural network we must consider that gathering a sound dataset is one of the most important and essential tasks in data analysis. It is therefore of paramount importance we have an efficient and accurate method of processing our data. We need to develop a dataset that includes sample footage that is labelled as either moving or stationary. This process is also referred to as labelling. The reason we have to conduct this operation to our dataset is so that the neural network will be

able to train itself based on the samples of information we will provide and comparing them to their corresponding labels. Therefore what we have implemented is a method of analysing moving footage separately to stationary footage. We supply each function with different sample footage of stationary samples and moving samples accordingly. Once the optical flow has been calculated and the average Delta Y per frame has been recorded we store this in a list. We decided to split our data into sets of 30 frames which equates to half a second of sample footage. This figure may change in the future however this is subject to later-stage testing. List structure 1 shown below shows how the data is structured each X representing a figure of delta Y per frame.

List Structure 1: [X1, X2, X3, …….. X30]

Next, we create a list that will contain all our other 30 frame sample lists List Structure 2 below shows the structure of our list containing all the 30 second samples. We then save all this information to the disk naming the files “moving data” and “panning data” accordingly.

List Structure 2:[[X1, X2, X3 …….. X30], [Y1, Y2, Y3 ……..Y30], …… [Z1, Z2, Z3 ……..Y30]]

Next, we import both our “moving data” and “panning data” and for each 30-second sample we append either a 0 representing a moving label or a 1 representing a panning label to a new list containing both the sample and the label as seen in List Structure 3.

List Structure 3: [[X1, X2, X3, …….. X30], 1]

These lists are then appended to a larger list containing all the lists with the 30 samples and their corresponding label. You can see this in List Structure 4 below. List structure 5 is an example of a list containing the dataset for the moving samples as you can see the labels are set to 0.

List Structure 4: [[[X1, X2, X3 …….. X30], 1], [[Y1, Y2, Y3 …….. Y30], 1], …… [[Z1, Z2, Z3 …….. Z30], 1]]]

List Structure 5: [[[X1, X2, X3 …….. X30], 0], [[Y1, Y2, Y3 …….. Y30], 0], …… [[Z1, Z2, Z3 …….. Z30], 0]]]

The final stage of our data processing is to combine both the moving and the stationary final lists (List Structure 4 and List Structure 5) and then randomizing the order of the list. The reason for the index randomization of the final list is due to the fact that we believe our neural network will be able to train itself more accurately if it has a randomized assortment of moving and panning shots in the dataset rather than first having all the stationary samples followed by all the moving samples.

First experiment with artificial intelligence

Due to very little published research being published on the niche area of my use case we have no way to know the effectiveness of our method. We, therefore, decided to run preliminary tests before we continue down this path of development. Therefore we decided to to use an artificial intelligence framework named Keras[20]. Keras allows for the use of artificial neural networks with relative ease in the Python programming language. On their website, they describe one of the main benefits of their framework: "*Allows for easy and fast prototyping*"[19]. Keras is a powerful library and can make use of multiple backends, we decided to make use of Tensorflow as it is incredibly popular and well documented[21][23]. On keras' website, they also state "***Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.***"[22] Therefore we found that this framework would allow us to build a simple sequential neural network to test our data against real training data. By using keras we were quickly able to set up a neural network with our limited research and knowledge of neural networks. We

must emphasize that the objective of this experiment was to see whether we could get a basic neural network to distinguish between the two different types of video which we present to it in numerical list form. Once we had gathered our data and stored it in pickle files[24], we ran our basic neural network. What we found is that the neural network trained itself to be 66.678% accurate on our test data. Although initially, we were not very satisfied with such a relatively low percentage. We soon realised that there were an incredible amount of improvements to make regarding both our neural network and our dataset. Therefore we came to the concluded the validity of our current approach and we will make further progress using this approach. One immediate improvement we could make is to gather more data, as we theorize that this will increase our accuracy dramatically.

Demonstration

To fully comprehend how the software functions we will now demonstrate how we trained our neural network with certain sections of code as a reference. In figure 20 you can observe a flow chart of how the datasets are gathered, prepared and then exported. As we have previously discussed it is fundamentally important that our data gathering methods are robust and consistent as our neural network will be trained on this data.

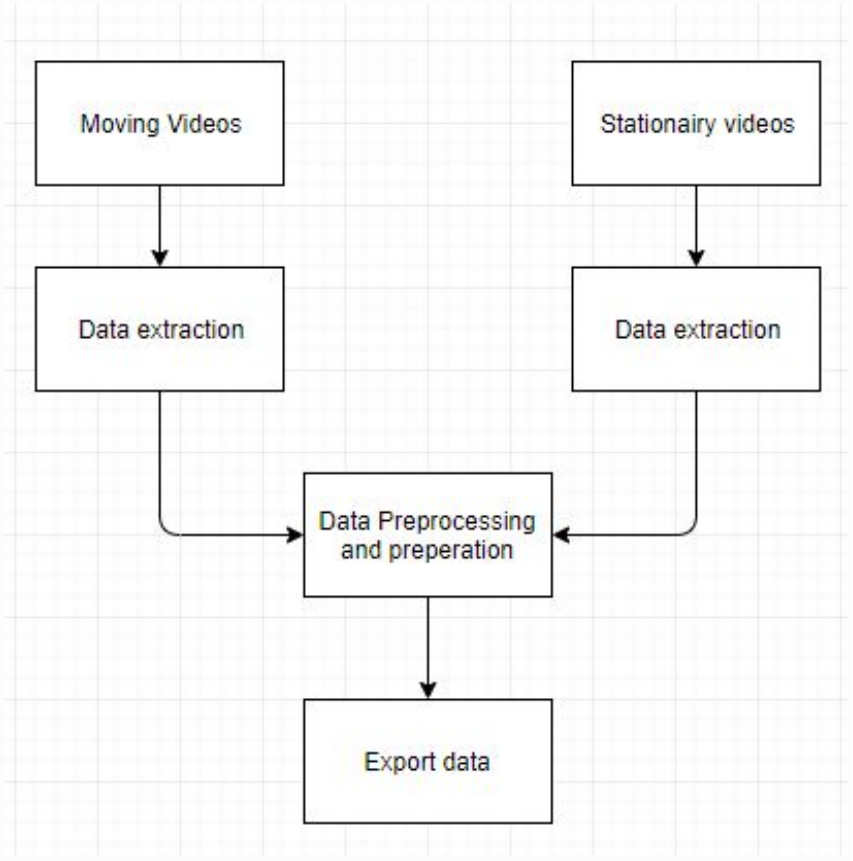


Figure 20. Flow Diagram of data gathering and preparation

Before we run our analysis program we first compile all the relevant video footage into two directories. These directories are intuitively labelled “Moving Shots” and “Stationary Shots”. We have two scripts which each use two threads whilst processing the videos. In figure 21 you can see the main loop that our script uses to extract the features between each section and then calculate the optical flow. In line 154 you can see that we calculate the delta in the Y coordinate of each particular

feature. On line 158 you can observe us calculating the average change of the Y coordinate over each of the extracted features. This average delta Y is then stored in a list named “currentPatternArr”. Once our loop has passed over 30 frames we append our list to a two-dimensional array named “patternAr”. This array holds all the delta Y values for each of the seconds of our clips.

```
127     while 1:
128         frames += 1
129         frame = fvs.read()
130         if fvs.more() == False:
131             break
132 
133 
134         old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
135         p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
136         frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
137         if len(p0) == 0:
138             p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
139 
140         # calculate optical flow
141         p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)
142         # print p1
143         # Select good points
144         good_new = p1[st == 1]
145         good_old = p0[st == 1]
146 
147         # draw the tracks
148         frameDeltaYArr = []
149         for i, (new, old) in enumerate(zip(good_new, good_old)):
150             a, b = new.ravel()
151             c, d = old.ravel()
152             mask = cv2.line(mask, (a, b), (c, d), color[i].tolist(), 2)
153             frame = cv2.circle(frame, (a, b), 5, color[i].tolist(), -1)
154             deltaY = b - d
155             frameDeltaYArr.append(deltaY)
156 
157         try:
158             deltaY = reduce(lambda x, y: x + y, frameDeltaYArr) / len(frameDeltaYArr)
159         except:
160             deltaY = 0
161         global currentPatternAr
162         currentPatternAr.append(deltaY)
163         if (frames == 30):
164             patternAr.append(currentPatternAr)
165             currentPatternAr = []
166             frames = 0
```

Figure 21. While loop for feature detection and dataset creation.

Once we have looped through all the videos within each folder we export our data into a pickle file so that our data can be stored and maintained as a list. This reduces computational complexity as it means we will not need to parse our data from a text or CSV file. In figure 22 you can observe us printing to the console. These lines are

printed after each video sequence has been processed. It allows us and the system administrator to make sure the process is running without errors and how long the process is expected to take. Because these are some of the most time-consuming operations it is vital that we receive feedback from our system to ensure that our system is running nominally. We print 4 lines of critical information to the observer. Firstly we inform the user that the system has completed processing a video. The system then prints out a sample from the pattern array of that sequence, this is so the user is able to determine if the data is valid and as expected. The system then proceeds to print out the average speed at which the software is able to process the video. The metric for this speed is frames per second or FPS. The user is then presented with the total amount of time taken for the software to complete that particular sequence. In figure 23 you can observe what this would look like for the user in a real data gathering scenario.

```
175
176         fps.stop()
177         print("[INFO] Completed Video")
178         print("[INFO] Datasets Gathered first 30 frames:" + str(patternAr[1]))
179         print("[INFO] Average FPS: {:.2f}".format(fps.fps()))
180         print("[INFO] Time Taken for this video:" + str(time.time() - starttime))
```

Figure 22. Printing information for the system administrator to observe

```
[INFO] Currently Analysing: Moving Shots/taal - Hulshorststraat 160 Den Haag.mp4
[INFO] Completed Video
[INFO] Datasets Gathered first 30 frames:[12.855199178059896, 16.109039306640625, 17.3867454
[INFO] Average FPS: 11.63
[INFO] Time Taken for this video:10492.398
```

Figure 23. Real console information from the data gathering process.

During the training of this neural network, we have had to run this process a number of times. This is why we chose to use multi-threading in this section of the process. This allowed us to almost double our processing time per video sequence. Even with this increase in performance the average processing time for our datasets was almost 3 hours. This is why we chose to store all our recorded datasets and create backups of our datasets in chronological order. This allowed us to keep track of our datasets and also allowed us to iterate through them to test different datasets.

In figure 24 you can see the last lines of the script which stores the arrays and also informs the user that the backup has been created. It also informs the user that the process is complete and the total amount of time taken since the start of the program.

```
currentDT = datetime.datetime.now()
backupTime = currentDT.strftime('%d-%m-%Y-%H-%M-%S')
backup = "patternbackups/PatternBackupMoving - " + backupTime
pickle.dump(oldpatterns, open(backup, "wb"))
print "[INFO] Backup saved"
pickle.dump(patternAr, open("datasetMoving.p", "wb"))
print "[INFO] Process Completed"
print "[INFO] Total Time Elapsed: " + str(time.time() - totalstarttime)
```

Figure 24. Last lines of dataset collection scripts.

Once our data has been gathered we move into a separate script that will preprocess all our data for the neural network. One of the most fundamentally important aspects of this preprocessing is the adding of labels to the data. These labels allow the neural network to determine what the outcome should be for each of the training datasets. In figure 25 you can see that our script first imports the data from the pickle files produced by our data gathering scripts. In figure 26 you can observe how we create a new array with each element of the datasets having a label attached to it. For the moving datasets, we have used the label “0” and for the stationary datasets we have used the label “1”. The usage of single integers as labels allows for a more compact data structure.

```

RawPanningDataset = pickle.load(open("datasetPanning3.p", "rb"))
RawMovingDataset = pickle.load(open("datasetMoving3.p", "rb"))
```

Figure 25. Datasets stored as pickle files are opened.

```
20
21  def processData():
22      for x in range(len(RawPanningDataset)):
23          appendage = [RawPanningDataset[x], 1]
24          ProcessedPanningDataset.append(appendage)
25      for x in range(len(RawMovingDataset)):
26          appendage = [RawMovingDataset[x], 0]
27          ProcessedMovingDataset.append(appendage)
28
```

Figure 26. Data processing and labelling

Once both of the datasets have been labelled they have to be combined into one large array. In figure 27 you can observe how we append both datasets to a new array named “FinalProcessedDataset”. Once these datasets have been combined we then proceed to shuffle the entire array. We shuffle the array as this promotes more effective training within the neural network. Once we have completed this we split our arrays again into two separate lists of the labels and the delta y per frames. You can observe this in figure 28. Both of these arrays will have corresponding indices. For example, the label for the data in index 242 will correspond to label 242 in the label list. We do this as the neural network expects us to input two lists, one with the data and one with the corresponding labels.

```
33
34  for x in range(len(ProcessedPanningDataset)):
35      FinalProcessedDataset.append(ProcessedPanningDataset[x])
36
37  for x in range(len(ProcessedMovingDataset)):
38      FinalProcessedDataset.append(ProcessedMovingDataset[x])
39
40
41  random.shuffle(FinalProcessedDataset)
```

Figure 27. Both arrays are combined and shuffled.

```
44  for x in range(len(FinalProcessedDataset)):
45      FinalProcessedLabels1.append(FinalProcessedDataset[x][1])
46      FinalProcessedData1.append(FinalProcessedDataset[x][0])
```

Figure 28. Shuffle array is split into two arrays.

In figure 29 you can observe the data being stored as in JSON format this is because we have to execute our neural network in a python version 3.6. JSON format is a universally accepted format that allows us to robustly transfer data from one python version to another without the possibility of data corruption.

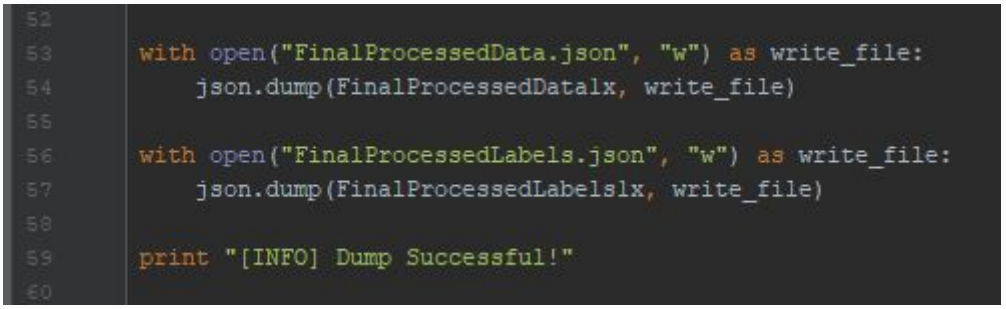


Figure 29. Exporting of the arrays and printing success confirmation

The next stage of our process is to train and test our neural network. We do this all within one script. As you can observe in figure 30 we import the datasets. The test datasets consist of very similar data to our training datasets but these datasets only include approximately 50 seconds of footage. These testing datasets are used to test our neural network after it has trained itself with the training datasets. In figure 31 we observe the last verification of the data before we import it into our neural network. This final verification step is in place to ensure the robustness of the operation and to avoid any possible errors by having elements in our dataset that might vary in size. In figure 32 you can observe our data being converted into a numpy array this is because the neural network requires it to be of this data structure. You can also observe in figure 32 how we specify the shape and size of our neural network.

```
source = open("FinalProcessedData.json", 'r').read()
ProcessedData = json.loads(source)

source2 = open("FinalProcessedLabels.json", 'r').read()
ProcessedLabels = json.loads(source2)

source = open("FinalProcessedTestData.json", 'r').read()
ProcessedTestData = json.loads(source)

source2 = open("FinalProcessedTestLabels.json", 'r').read()
ProcessedTestLabels = json.loads(source2)
```

Figure 30. Importing of the datasets into the neural network script

```
# Removing any elements of the arrays that are over 30 in length
for x in ProcessedData:
    if len(x) != 30:
        print(len(x))
        ProcessedLabels.pop(ProcessedData.index(x))
        ProcessedData.remove(x)
```

Figure 31. Verification of data.

```
# convert datasets to numpy arrays
trainX = np.array(ProcessedData, dtype="float")
trainY = np.array(ProcessedLabels)

testX = np.array(ProcessedTestData, dtype="float")
testY = np.array(ProcessedTestLabels)


model = Sequential()

# input layer
model.add(Dense(60, input_shape=(30,), activation="relu"))

# hidden layers
model.add(Dense(60, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(60, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(30, activation="relu"))

# output layer
model.add(Dense(1, activation="softmax"))
```

Figure 32. Datasets are converted into numpy arrays and neural network model is created.

Once we have created our neural network model it is time to compile it and train it. In figure 33 you can observe how we compile it using “model.compile” this function takes in a variety of variables namely what type of loss function and optimizer we will use. As previously stated we will use Stochastic gradient descent with a learning rate of 0.01. Once we have compiled our model we use “modle.fit” with our training datasets and our test datasets as inputs. We also specify the batch size and the number of epochs as discussed during the design section of this report.

```
74  # Perameters for our NN
75  INIT_LR = 5
76  EPOCHS = 500
77
78  print("[INFO] training network...")
79  opt = SGD(lr=0.01)
80  model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
81
82
83  H = model.fit(trainX, trainY, validation_data=(testX, testY), epochs=EPOCHS, batch_size=25, shuffle=True)
```

Figure 33. Compilation and fitting of our model

Evaluation

The neural network we have built and trained with this limited amount of training data currently produces an accuracy metric of 66.678%. This means that it can determine whether a frame is moving or stationary with 66% accuracy. The way in which our program would operate currently is it takes as input a video file, this video file is then analysed frame by frame. The required data is then extracted from the frames and this is then passed to the neural network. The neural network then analyses each second of video and determines whether that second is either a stationary second or a moving second. This information is then outputted either in the form of printing to screen or printing to a text file. The program can currently analyse videos at an average speed of 11fps. As we previously discussed RITS has advised us that the average length for raw videos files will be 3.5 minutes. This means that our software will be able to analyse an average video in under 10 minutes.

Although the performance of our neural network is relatively poor in accuracy the speed and design of the system are effective. This is due to the removal of complex mathematical functions that would be required without a neural network in place. This means that our software is able to categorize each second in extremely quickly.

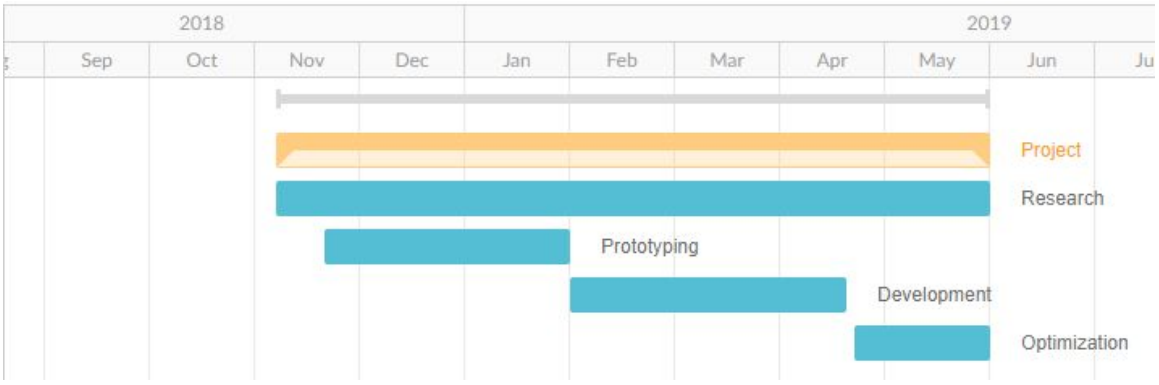


Fig. 34: Gantt chart produced in November 2018

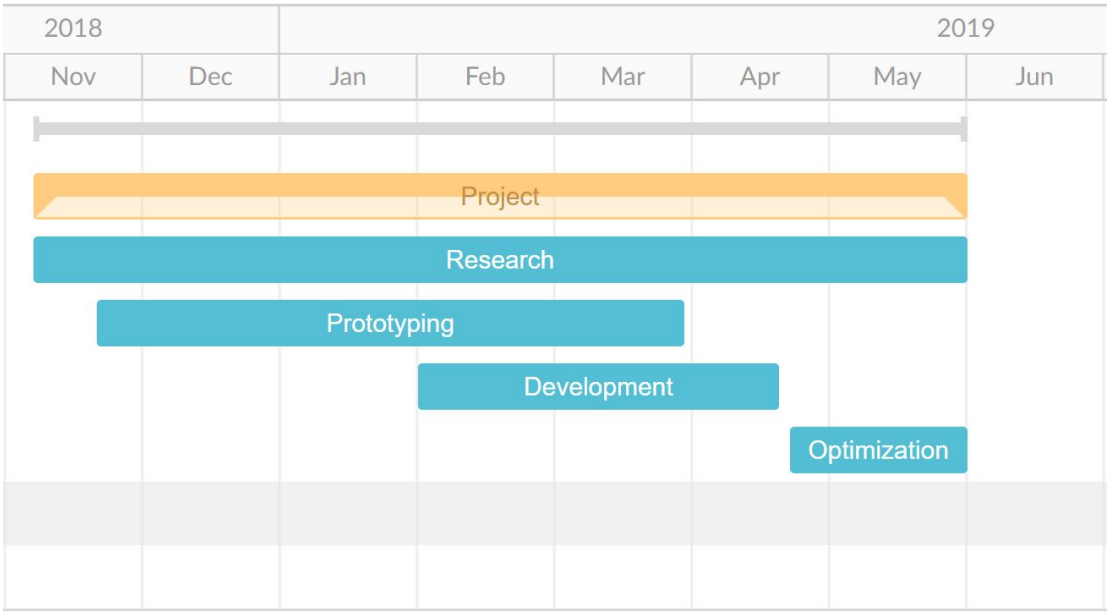


Fig. 35: Gantt chart produced in January 2019

In figure 34 you can observe the Gantt chart that was proposed at the start of the project with figure 35 displaying a revised Gantt chart that was produced the third month of the project. As you can see in the second Gantt chart there was an increase in the prototyping stage of the project that would run in parallel with the development stage of the project. This was due to the extensive amount of testing that needed to be conducted to produce such a complicated and novel software package for our specific use case. The management of the project in regards to the Gantt chart in figure 35 was accurate and all milestones were reached on time. However, with our current method and timeframe, we were not able to produce a neural network with greater accuracy than 66%.

Conclusion

To conclude this report we will critically evaluate both the development process and our software package as a whole. During the initial stage of research and prototyping of the project, we came across a methodology that produced promising results with relatively low computing complexity. This meant that theoretically, our method would allow us to produce software that was accurate and fast enough to be used as professional software for RITS. With this in mind, we extrapolated on this research and tried to produce software using this method alone. However, after further testing, we concluded that our method was incredibly fast however it was not producing results accurate enough to be used professionally.

Looking back at our initial approach we believe that there could have certainly been improvements in how we approached this project. We believe that the complexity of this problem would not allow us to produce a professional software package in this timeframe with the limited amount of human resources available. What we have been able to do however is demonstrate to RITS the capabilities that neural networks have in this field.

The software package we have produced functions as a great foundation for further development within the company. We have proposed that with a larger time frame and with more resources we could produce a neural network that may be slower but that will have increased accuracy. Although we predict that this neural network will have a significantly larger computational complexity.

Our current approach of extracting optical flow information from the video sequences relies on the Lucas Kanade method of detecting features on the image which we should track. This gave us a relatively small array of optical flow information with an average size of 32 X and Y coordinates per frame. We have however hypothesised a new approach which we have offered to work on further for RITS. This approach would consist of extracting the optical flow information for every single pixel within the image sequences. This is also known as dense optical flow.

Dense optical flow would be extremely computationally complex and we predict that our neural network would need to be able to digest a four-dimensional array of information to accurately predict this. For a full HD image per second, we would produce an array of size 30x1920x1080x2 this would equate to 124,416,000 float values per second of footage. We have suggested that there would be ways in which we could dramatically decrease the size of the dataset by optimising our matrices. However, this would need to be tested which would take a large amount of time and resources. In figure 36 you can see an example of this dense optical flow being visualized. You can see the direction of motion of the camera by the trail of the green lines attached to each pixel.

Figure 36. Example of visualised dense optical flow

Although we have not been able to produce a neural network that is able to produce accurate enough results for professional usage I believe we have provided RITS with a proof of concept and the necessary research to produce a neural network that is accurate enough to be used in a professional environment.

References

[1] Regentsitsolutions.co.uk. (2018). *Regent's IT Solutions*. [online] Available at: http://www.regentsitsolutions.co.uk/ [Accessed 9 Nov. 2018].

[2]Opencv.org. (2018). *About - OpenCV library*. [online] Available at: https://opencv.org/about.html [Accessed 9 Nov. 2018].

[3]Docs.opencv.org. (2018). *Shi-Tomasi Corner Detector & Good Features to Track — OpenCV 3.0.0-dev documentation*. [online] Available at: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html [Accessed 9 Nov. 2018].

[4]Docs.opencv.org. (2018). *Harris Corner Detection — OpenCV 3.0.0-dev documentation*. [online] Available at: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html [Accessed 9 Nov. 2018].

[5] Tomasi, C. and Shi, J. (1994). Good Features to Track. *IEEE*, CVPR94, p.8.

[6] Warren, W. H., & Hannon, D. J. (1988). Direction of self-motion is perceived from optical flow. *Nature, 336*(6195), 162-163.

[7] Patel, D. and Upadhyay, S. (2013). Optical Flow Measurement using Lucas Kanade Method. *International Journal of Computer Applications*, 61(10), pp.6-10.

[8] Ranganathan, P., Adve, S. and Jouppi, N. (1999). Performance of image and video processing with general-purpose processors and media ISA extensions. *ACM SIGARCH Computer Architecture News*, 27(2), pp.124-135.

[9] Puget Systems. (2018). *Premiere Pro CC 2017.1.2 CPU Comparison: Skylake-X, Kaby Lake-X, Broadwell-E, Kaby Lake, Ryzen 7*. [online] Available at: https://www.pugetsystems.com/labs/articles/Premiere-Pro-CC-2017-1-2-CPU-Comparison-Skylake-X-Kaby-Lake-X-Broadwell-E-Kaby-Lake-Ryzen-7-969/#Conclusion [Accessed 9 Nov. 2018].

[10] Puget Systems. (2018). *Premiere Pro CC 2018 Workstation GPU Performance*. [online] Available at: https://www.pugetsystems.com/labs/articles/Premiere-Pro-CC-2018-Workstation-GPU-Performance-1116/ [Accessed 9 Nov. 2018].

[11]Ssd.userbenchmark.com. (2018). *SSD User Benchmarks - 964 Solid State Drives Compared*. [online] Available at: https://ssd.userbenchmark.com/ [Accessed 9 Nov. 2018].

[12]Hdd.userbenchmark.com. (2018). *HDD User Benchmarks - 1010 Hard Drives Compared*. [online] Available at: https://hdd.userbenchmark.com/ [Accessed 9 Nov. 2018].

[13]Artificial Intelligence Blog. (2018). *Pattern Recognition*. [online] Available at: https://www.artificial-intelligence.blog/terminology/pattern-recognition [Accessed 9 Nov. 2018].

[14]Docs.opencv.org. (2018). *Shi-Tomasi Corner Detector & Good Features to Track — OpenCV 3.0.0-dev documentation*. [online] Available at: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html [Accessed 9 Nov. 2018].

[15]D. Patel and S. Upadhyay, "Optical Flow Measurement using Lucas Kanade Method", *International Journal of Computer Applications*, vol. 61, no. 10, pp. 6-10, 2013. Available: 10.5120/9962-4611.

[16]M. Wall and A. Smith, "The Representation of Egomotion in the Human Brain", *Current Biology*, vol. 18, no. 3, pp. 191-194, 2008. Available: 10.1016/j.cub.2007.12.053.

[17]A. Giachetti, M. Campani and V. Torre, "The use of optical flow for road navigation", *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 34-48, 1998. Available: 10.1109/70.660838.

[18]S. Armen, "Analysis of biologically inspired artificial neural networks", *Frontiers in Systems Neuroscience*, vol. 3, 2009. Available: 10.3389/conf.neuro.06.2009.03.062.

[19]Wlodarczak, Peter. (2017). Cyber Immunity - A Bio-Inspired Cyber Defense System. 199-208. 10.1007/978-3-319-56154-7_19.

[20]Florian Raudies and Heiko Neumann(2018). An efficient linear method for the estimation of ego-motion from optical flow

[8]https://keras.io/

[21]"About Tensorflow" - https://www.tensorflow.org/about [Accessed 27 Dec. 2018]

[22]"Why use keras" - https://keras.io/why-use-keras/ [Accessed 27 Dec. 2018]

[23]"Module: tf.keras" - https://www.tensorflow.org/api_docs/python/tf/keras [Accessed 27 Dec. 2018]

[24]"Pickle - Python object serialization", https://docs.python.org/3/library/pickle.html [Accessed 27 Dec. 2018]

[25] Cvlibs.net. (2019). *The KITTI Vision Benchmark Suite*. [online] Available at: http://www.cvlibs.net/datasets/kitti/ [Accessed 17 Apr. 2019].

Appendix A: User guide to the software

In this appendix, we will discuss how to use and test the software that I have attached on the USB drive. On the USB drive, you will find two primary directories.

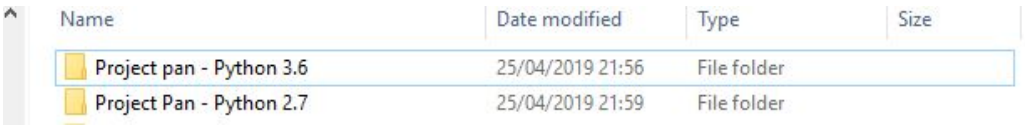


Figure A.1 - USB directories

As you can see due to incompatibilities with some python packages we have had to split our software to use python 2.7 and python 3.6. Most of the operations occur in the python 2.7 scripts and the neural network is run on python 3.6. Due to the size restrictions on our USB, we have had to remove some of the training video sequences, Therefore, your results may vary. In the file “Project Pan - Python 2.7” you will find a variety of files however there are three main ones that are essential to the project. “PanningPatternStorage.py”, “MovingPatternStorage.py” and “DatasetProcessing_Keras_AI.py”. PanningPatternStorage.py takes all the sequences in the “panning shots” folder and analyses them. Next, you will want to run MovingPatternStorage.py and this will analyse all the files in the “moving shots” directory. Once these two scripts have been completed you can run “DatasetProcessing_Keras_AI.py” which will process all the data for the neural network. There is a requirements.txt file within this directory that will have all the python dependencies necessary to run this software.

Moving on to the directory “Project pan - Python 3.6” the scripts in this directory require python 3.6 to run. In this directory, you will also find a requirements.txt file with all the dependencies to run the code. Once you have copied the JSON files that you generated with the “DatasetProcessing_Keras_AI.py” from the “Project pan - Python 2.7” directory you can run “Keras_AI.py” located within the “PanningDetector” directory. This will then train the neural network and provide you with the accuracy of the neural network.

Appendix B: Supervisor Log

NAME: *Christiaan Burrett-Bijlstra*

STUDENT NUMBER: *M00608813*

Date	Comments	Supervisors signature
15/11/2018	Discussion regarding the IPP	Michael Heeney
18/1/2019	Discussion regarding the literature review	Michael Heeney
26/04/2019	Discussed final submission and report structure	Michael Heeney

Apart from these formal meetings we have also conversed informally regarding the project during class sessions.