Final Report

**Project Title:** Building Deep Learning Applications on a Spark Cluster: Object Detection for Self-Driving Cars

**Name:** Kerney MohammadOwais

**UID:** 3035296866
Abstract

Road accidents have become a dangerous everyday happening and their associated death toll is on a constant rise. To alleviate this problem, intensive research is being carried out in the field of self-driving cars in order to eventually replace distracted drivers from roads. However, self-driving cars have a long way to go before they can achieve widespread acceptability. Their underlying technology is still not up to par with the standards of a safety critical application. This project aims to focus on one of the most important underlying components and pressing issues of a self-driving car: its object detection and tracking system. By exploring the use of recent advances in deep learning and cloud computing, the project aims to optimize the process of object detection and tracking in order to prepare self-driving cars to be able to adapt to ever changing road conditions.

Acknowledgements

I would like to thank my supervisor, Dr. CL Wang for his continuous help, guidance and mentorship over the course of this project’s timeline.

I would also like to thank my parents and friends including Mahnoor Ahmed, Mazel Mihardja, Preya Shah and Rhea Kochar for their help with formatting and designing the report.

Abbreviations

GPS: Global Positioning Satellite  
GT: Ground Truth
CNN: Convolutional Neural Network  
LT: Latency
TN: True Negative  
TP: True Positive
FN: False Negative  
FP: False Positive
YOLO: You Only Look Once  
DNN: Deep learning Neural Network
DR: Detection Rate  
TRDR: Tracker Detection Rate

Kerney Mohammad Owais
Contents

1. Introduction  
   1.1 Background  
   1.2 Motivation  
   1.3 Brief introduction of technical concepts  
   1.4 Current Solutions  
   1.5 Project Scope  
   1.6 Project Deliverable  
   1.7 Outline of report  
2. Project Methodology  
   2.1 Design and Procedure  
      2.1.1 A brief rundown of the system design and architecture  
   2.2 Equipment/platform setup  
      2.2.1 Capturing a camera stream and selecting specifications using a front end  
      2.2.2 Setting up the object detection module  
      2.2.3 Integrating the object detection module with an input data stream  
      2.2.4 Configuring the frontend to demonstrate object detection results  
   2.3 Test data and test case design  
3. Results and Outcomes  
   3.1 Discussion of results  
      3.1.1 Configuration of a standalone object detection system  
   3.2 Test data and test case design  
      3.1.2 Distributed object detection using Spark  
         3.1.2.1 Implementation 1: Batch processing using Spark-Core  
         3.1.2.2 Implementation 2: Realtime processing using Spark-Streaming  
   3.2 Testing Results  
      3.2.1 Testing Methodology  
      3.2.2 Results: Implementation 1 (Spark Batch Processing)  
         3.2.2.1 Spark Analysis-Implementation 1  
         3.2.2.2 Analysis-Tiny Yolo with 4 workers vs Tiny Yolo with 2 workers  
      3.2.3 Results: Implementation 2  
         3.2.3.1 Implementation 2- Spark Cluster Analysis
3.2.3.2 Result Analysis- Implementation 2 (Only tested on YOLO at Fast Speed)

Tests on drive time videos played on a monitor
Live webcam stream captured and transmitted as an HTTP link over CS VPN

3.3 Limitations/Difficulties encountered

5. Conclusion

5.1 Summary and Future Direction

References
**Figures**

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Object Detection system for YOLO</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Rundown of system architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Spark Setup</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>UI of system</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementation of standalone system</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Tiny Yolo vs YOLO v2 comparison</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Implementation 1: Spark Batch Processing</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Implementation 2: Spark-Streaming</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Common Objects for In-Lab tests</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Implementation 1 Results with 4 executors</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Implementation 1 Results with 2 executors</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>4 Spark executors with IP addresses</td>
<td>38</td>
</tr>
<tr>
<td>4.0</td>
<td>Spark Timeline and Executors Summary for Tiny (4 executors, Fast)</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Executor Specific Spark Timeline for Tiny (4 executors, Fast)</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Spark Timeline and Executors Summary for Tiny (4 executors, Slow)</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Executor Specific Spark Timeline for Tiny (4 executors, Slow)</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Spark Timeline and Executors Summary for Yolo2 (4 executors, Fast)</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Executor Specific Spark Timeline for Yolo2 (4 executors, Fast)</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Spark Timeline and Executors Summary for Yolo2 (4 executors, Slow)</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>Executor Specific Spark Timeline for Yolo2 (4 executors, Slow)</td>
<td>46</td>
</tr>
<tr>
<td>4.8</td>
<td>Spark Timeline and Executors Summary for Tiny (2 executors, Fast)</td>
<td>48</td>
</tr>
<tr>
<td>4.9</td>
<td>Executor Specific Spark Timeline for Tiny (2 executors, Fast)</td>
<td>49</td>
</tr>
<tr>
<td>5.0</td>
<td>Spark Timeline and Executors Summary for Tiny (2 executors, Slow)</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>Executor Specific Spark Timeline for Tiny (2 executors, Slow)</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Spark Timeline and Executors Summary for Yolo2 (2 executors, Fast)</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Executor Specific Spark Timeline for Yolo2 (2 executors, Fast)</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Spark Timeline and Executors Summary for Yolo2 (2 executors, Slow)</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Executor Specific Spark Timeline for Yolo2 (2 executors, Slow)</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Result for Implementation 2</td>
<td>57</td>
</tr>
<tr>
<td>5.7</td>
<td>DAG for Implementation 2</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Spark Timeline and Executors Summary for Tiny (1 executor, Fast)</td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>Executor Specific Spark Timeline for Tiny (1 executor, Fast, Stage A)</td>
<td>61</td>
</tr>
<tr>
<td>6.0</td>
<td>Executor Specific Spark Timeline for Tiny (1 executor, Fast, Stage B)</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Spark Timeline and Executors Summary for Tiny (2 executors, Fast)</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>Executor Specific Spark Timeline for Tiny (2 executors, Fast, Stage A)</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>Executor Specific Spark Timeline for Tiny (2 executors, Fast, Stage B)</td>
<td>66</td>
</tr>
<tr>
<td>6.4</td>
<td>Spark Timeline and Executors Summary for Tiny (Local Spark, Fast)</td>
<td>67</td>
</tr>
<tr>
<td>6.5</td>
<td>Executor Specific Spark Timeline for Tiny (Local Spark, Fast, Stage A)</td>
<td>68</td>
</tr>
<tr>
<td>6.6</td>
<td>Executor Specific Spark Timeline for Tiny (Local Spark, Fast, Stage B)</td>
<td>69</td>
</tr>
<tr>
<td>6.7</td>
<td>Comparison of Executor Specific Timelines</td>
<td>71</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Background

Injuries sustained from road traffic accidents are the leading cause of death around the world for people aged 15 to 29 and more than 1.25 million people lose their lives due to them every year. If sustained action is not taken, road accidents are predicted to become the 7th largest cause of death by 2030 [1]. To alleviate this problem and make roads safer, extensive technological research is being carried out in the field of self-driving cars and many leading car companies in the world are pouring a vast amount of resources to advance this technology. For example, Ford, one of the leading car manufacturers in the world, has already invested almost a billion US dollars and plans to test its self-driving cars on the streets of Washington DC early next year [2].

Self-driving cars have the potential to reduce the number of road accidents and the associated death toll [3]. Nearly 90% of road accidents happen due to human error [4]. By solely being focused on driving and being ever so ready to react to urgent situations, self-driving cars can avoid the risk of accidents caused by distracted drivers behind the wheel [5]. They can also be set to obey the law, respect road signs, and never lose track of their speed and the official speed limit. However, self-driving cars and their underlying technology is still not advanced enough to be production ready and replace drivers on roads. There has been widespread criticism of the technology and its safety concerns. The next section explores some of the research gaps and existing problems with self-driving cars.
1.2 Motivation

Truly autonomous cars do not require human intervention and need to be able to drive, navigate and make their own driving decisions using various technologies and tools. Roads are rarely predictable and situations that require instant decision making may occur anytime. For example, self-driving cars may have to navigate through heavy rain or snowstorms that could hide or distort the painted lines on roads which are extremely important for self-driving cars to stay in their lanes. Hence, an autonomous driving system needs to be able to adapt to changing conditions and make calculated and quick driving decisions in unfavorable conditions.

Additionally, self-driving cars need to detect, recognize and track objects like other cars, pedestrians and cyclists around them. Since the appearance and movement of these entities is highly unpredictable, the object detection and tracking system needs to be accurate and have an instant feedback time. The smallest of delays in obtaining object tracking data or a lapse in judgement by the object detection system can lead to a delayed reaction by the driving system, potentially leading to an accident. A very popular and prominent example of this is an incident from earlier this year when an Uber self-driving car failed to detect a crossing pedestrian and ended up running over them. The car’s object detection had a delayed response in detecting the pedestrian; classifying them as an unknown object, a vehicle and then a bicycle [6].

The object detection system also needs to be constantly alert about any intruding objects and instantly inform the car’s decision system to react according to the situation. This is the primary area of concern that this project is going to address and attempt to contribute towards solving.
1.3 Brief introduction of technical concepts

The following technical concepts are important components of the project and are mentioned several times in this document:

**Machine Learning:** A branch of artificial intelligence based on the concept of systems learning by identifying patterns in data and making decisions without human intervention.

**Deep learning:** A branch of machine learning focusing on data representations as they appear in the human brain.

**Cloud computing:** The processing of data or computer operations in multiple systems/computers rather than in a single machine.

1.4 Current Solutions

Mainstream ways of object detection, especially in applications which are not safety critical, employ various techniques including background subtraction.

Background subtraction is mainly used for video sequences with relatively static background frames. It works by separating the sequence into foreground and background; with the former containing dynamic objects such as moving cars and people in the context of a drive time video and the latter containing static objects such as roads, buildings etc. Objects are detected by first capturing a reference background scene before an object of interest enters the scene and then subtracting this background reference from the current image frame when the object is finally present. This isolates the object of interest and makes it easy to detect it [7]. For object tracking, some mainstream methods can be divided into feature and learning based methods. Feature based methods make use of characteristics like texture, gradient and color which are extracted first to identify various objects in a video stream [8].

Learning based methods are one of the most advanced object detection and tracking techniques in the present day. Modelling the object tracking problem as a decision-making process, learning
based methods train classifiers to differentiate between objects and their background in an image fragment [7].

Recently, deep learning-based object detection and tracking frameworks are gaining popularity in autonomous driving systems. Using convolutional neural networks (CNN) and Deep Neural Networks (DNN), these frameworks have made great breakthroughs in fast and accurate object detection. They are able to learn semantic and high level features with ease and have algorithms that allow for expressive and robust training. There are two main strategies used to achieve object detection by these networks. One is to follow the traditional object detection pipeline by dividing images into region proposals and then predicting the object category for each region nominated. This technique is used by R-CNN (Regions with CNN) and Fast R-CNN. The other technique is to consider the object detection phenomena to be a classification or regression problem and having a unified framework to achieve final results collectively. This technique is used by neural networks such as MultiBox, YOLO (You Only Look Once) and YOLOv2 [9].

In recent times, the YOLO framework has received massive acclaim. The main strategy employed by YOLO is to divide an input image into a D X D grid and make each grid cell responsible to detect any object having its center in that cell (Refer to Figure 1.1 on the next page) Each cell submits a list of bounding boxes for objects detected and their confidence scores where confidence scores represent the probability of the predicted object being in the box, calculated by:

\[
\Pr(\text{Object}) \times \text{IOU (truth prediction)}
\]

where \( \Pr(\text{Object}) > 0 \) is the likelihood of an object’s existence and \( \text{IOU} \) is the confidence of its prediction.

At the same time, YOLO also calculates conditional class probabilities which can be denoted with:

\[
\Pr(\text{Class}(i) | \text{Object})
\]

for each object identified in order to help classify the object. [9]
Fig 1.1: The process of object detection in YOLO. This figure shows that each image under consideration is divided into a grid and grid cells that contain centers of detected objects with high confidence are used to draw bounding boxes.
While these deep learning techniques are ground breaking events in the research for object detection and classification, error rates for all these systems are still relatively high for the standards of a safety critical application. In addition, the performance of these object detection and tracking systems is mainly computed based on prerecorded datasets and is yet to be widely implemented and tested on actual autonomous car systems in real-time. Not only does the preparation and labelling of pre-recorded datasets take time, testing on pre-recorded datasets also makes it hard to analyze the failure points of an object detection system.

[10].

Additionally, video data obtained from car cameras requires fast processing and streamlining before being fed to an object detection system. With multiple cameras attached to a single autonomous car and hence large and diverse camera streams, the input data is very heavy and requires high computing power and electrical consumption to process. With a linear data processing system to feed data into a deep learning neural network, the object detection process becomes computationally intense and sometimes infeasible. There is also high latency involved when processing such a large amount of image data frame by frame which is not acceptable for a safety critical application. For example, with a linearly implemented system, each frame obtained from each camera stream needs to be fed into a neural network, processed and labelled with bounding boxes for all detected objects. This linear system is indeed very slow and creates a long line of frames waiting to be processed.

For self-driving cars to be able to react to ever changing road conditions in a timely manner, the input data processing requires serious optimization and resource conservation. The process of feeding the input data from multiple cameras into a neural network and getting output results needs innovation and change in order for it to be ready for deployment in autonomous vehicles. Parallelization of tasks, simultaneous processing of frames and distributed processing of the neural network computations for object detection have a great potential to tackle this problem.
1.5 Project Scope

This project explores the use of deep learning combined with cloud computing to try and optimize the speed and accuracy of object detection and tracking for self-driving cars. While exploring the use of existing deep learning-based object detection algorithms, it will attempt to streamline the process of data input and processing by using cloud computing. The idea is to create a distributed stream to perform the object detection computations in a branched manner, hence attempting to reduce processing latency that is encountered during linear processing of data.

The project includes the use of a deep learning neural network to recognize and track objects while employing a cloud-based architecture to handle distributed data input streaming and distributed processing.

The scope of the project includes:

1. using a pre-trained deep learning based neural network to recognize and track objects commonly found on roads
2. streamlining and optimizing the process of data input to the object detection module and distributed execution of neural network computations using cloud computing
3. using real time video stream as testing data to compute the response time and accuracy of the object detection and tracking module as well as computational power required
1.6 Project Deliverable

The project deliverable will be an end to end computer application capable of accepting a camera stream as input. After processing and injecting the stream to an object detection module, the application will return real-time video output outlining detected objects and tracking their movement continuously through the entirety of the stream. The aim of the application is to increase the speed and accuracy of object detection and tracking for drive time video data which, in the bigger picture, would enhance the ability of self-driving cars to make timely and accurate driving decisions.

1.7 Outline of report

The following sections of the report, firstly, go over the detailed methodology of the project. The report briefly describes the system design and architecture and then moves on to environment setup required to run the application. After describing the core parts of the application, the report goes on to mention the testing criteria and methods as well as the expected and achieved results. Next, the report highlights some of the problems and hurdles faced in the development of the system followed by a conclusion to wrap up the content.
2. Project Methodology

2.1 Design and Procedure

2.1.1 A brief rundown of the system design and architecture

The end to end deliverable, shown in Fig. 1.2 on the next page, features a frontend whose primary purpose is to accept video stream as input data. The input data is then forwarded to the backend. Data is processed by the backend and streamed into the deep learning object detection module. The object detection model processes the data in real time and displays detected objects with bounded boxes and name labels through the frontend.

![Diagram of project deliverable](image)

*Fig 1.2: A diagram showing the basic architecture of the project deliverable. A camera stream from the frontend is transmitted to the Spark backend for distributed processing in YOLO before results are returned.*

Kerney Mohammad Owais
2.2 Equipment/platform setup

The platform setup process will be done in 4 main steps:

2.2.1 Capturing a camera stream and selecting specifications using a front end

The object detection system, developed using Java 8, needs a continuous stream of live video data to be operated and tested. For the purpose of this project, a Logitech C922 Pro Stream webcam, capable of capturing 30-60 frames per second, has been selected and mounted using a USB terminal to a 64-bit Linux machine. To capture the video stream, the open source platform of OpenCV 3.4.1 (Open Computer Vision) and JavaCV (Java Computer Vision) will be employed. These platforms provide easy-to-use APIs for not only capturing video streams and images but also dividing them into individual frames for further processing using Java. [11]. A Java front end has been set up with options to start a live stream for processing and selecting options such as the neural network to be used for object detection and the speed/quality settings to configure the neural network with. This front end will act as a user’s control panel to start/test the system and explore the various features that it has. It will also be responsible to display the video stream results i.e. the live camera stream with bounding boxes and name labels for detected objects.

2.2.2 Setting up the object detection module

Object detection and tracking will be achieved by using YOLO (You Only Look Once); a deep learning neural network (DNN) based system. YOLO is an open source platform that uses a single neural network to accept images as input, divide each input image into regions with bounding boxes and predicting the presence of objects, as discussed earlier in the report.

YOLO has proven itself to have one of the fastest response times in its domain. Moreover, it offers the flexibility to change its configurations to achieve a tradeoff between speed and accuracy [12]. This feature is extremely important for the result analysis of this project to
determine the different levels of speed and accuracy achievable and be configured using the application’s frontend.

For the object detection module, YOLO will be deployed on Deep Learning for Java (DL4J). DL4J is a commercial, open and distributed deep-learning library for Java and Scala. Providing integration with Hadoop and Spark, DL4J provides the infrastructure to run deep learning algorithms on multiple machines/GPUs [13].

DL4J has a neural network repository called the DL4J Zoo which has both untrained and pre-trained implementations of YOLO and several other DNNs available for use. For the purpose of this project, pre-trained versions of YOLO have been downloaded and configured.

YOLO is pre-configured to receive image data as input and return output images with each identified object separately labelled. For it to be able to detect objects for a self-driving car system, it needs to be configured to accept continuous image streams as input using its API.

2.2.3 Integrating the object detection module with an input data stream

A major component of the project is its ability to swiftly handle a large and continuous stream of input video data, feed it to the object detection module in a distributed manner and collect object detection results.

To achieve this, the use of an open source data streaming platform called Apache Flink was initially considered. Flink is capable of executing dataflow programs in a parallel and pipelined manner which means that it can collect heaps of video clip data from a destined source and feed it to our object detection system in the form of a continuous pipeline. However, due to lack of online academic and troubleshooting support, few third party libraries as well as a shortage of training materials, the use of Apache Flink was taken out of consideration.

Kerney Mohammad Owais
Instead, the platform selected for this project was Apache Spark. Spark is capable of achieving high performance for both batch and stream processing using a query scheduler, optimizer and physical execution engine. Spark is generally very easy to use and supports applications written in a wide variety of programming languages including Java, Scala, Python, R and SQL. Being a more mature open source project, Spark not only speeds up the process of data input but also reduces the computational resources required as Spark is capable of distributing data processing jobs to multiple computers and processors. Moreover, Spark is capable of storing data that has been processed over time to make it easy to analyze the performance of the deliverable and to test it. [14]

Spark has multiple execution modes with connectivity to other execution platforms such as Apache Hadoop and Mesos. However, for this project, Spark’s standalone cluster mode will be employed.

In standalone mode, Spark uses its own resource manager and allows the creation of a master-worker architecture to distribute the computation jobs to multiple executors which can be located on a single computer or a cluster of machines.

Apache Spark streamlines the data streaming process by distributing its workload over several processors or computers. The main machine that Spark is installed on operates as the master node and any other processors or computers employed to distribute the workload over are called the worker nodes. The Spark resource manager in the master node is responsible for deciding how much workload to outsource to the worker nodes and when to receive the intermediate results from these nodes.

For this project, a Spark standalone cluster has been set up using a 64-bit Linux machine as the master and 4 identical Linux machines as worker nodes on the HKU CS network, as shown in Figure 2.1 on the next page. To configure this cluster, Apache Spark version 2.4.0 was installed on all 5 machines. Next, to allow the master node to communicate with worker nodes over the network without requiring authentication each time, password less SSH (secure shell) protocol was established between the master and all 4 workers. To start the cluster, a master node was
initiated from the main Linux machine and worker nodes were instantiated in the worker machines. These workers were instructed to report to the master node using the main machine’s IP address to bind them to the master.

![Diagram of Spark architecture]

**Fig. 2.1:** A diagram showing the distributed architecture of Spark. The master node is located in a Linux machine and worker nodes are established in computers on the HKU LG104 Lab, connected to the master using SSH.
Spark can be easily plugged into a Java application to support computations performed on DL4J and together these two platforms will make the distributed object detection system for this project. Once the integration of DL4J and Spark is done, the object detection system is ready to work with a data input stream and the next step will be to setup an interface to display the results of the system.

2.2.4 Configuring the frontend to demonstrate object detection results

While YOLO can produce output images with detected objects labelled, just a static output is not good enough for a self-driving car system which needs to continuously detect and track surrounding objects. For a more intuitive output result, the application consists of a Java frontend component to display detected objects and track their coordinates as they move in a running video stream.

2.3 Test data and test case design

- Common testing terms

**System output**: The output result generated by the system being tested

**Ground Truth, GT**: The true data (for example: number of actual objects in an image frame)

**True Negative, TN**: Number of frames where system output and Ground Truth(GT) agree on absence of objects

**True Positive, TP**: Number of frames where system output and GT agree on presence of objects

**False Negative, FN**: Number of frames where GT contains at least one object that system output does not

**False Positive, FP**: Number of frames where system results contain at least one object that GT does not

Kerney Mohammad Owais
The performance of the system will be tested and analyzed based on the following criteria:

1. **Detection based metrics**

These metrics will be used to evaluate the performance of object detection on individual video frames of the input video data. They will not include monitoring of individual objects over the life of the test or multiple video frames. Instead, all the objects in a single video frame will be individually tested to determine the number of matches between the system output and the Ground Truth (actual number of objects in the concerned video frame). The following calculations will be employed:

   ii. **Accuracy**: Measure of the actual performance of the system with regards to correctly detecting and correctly rejecting objects

   \[ \frac{TP+TN}{GT} \]

   i.e. the sum of true positives and true negatives relative to total number of GT objects [8].

2. **Speedup of overall object detection and tracking results**

To measure the overall speedup achieved by the system, the following metrics will be used:

   i. **Time Detection Lag**

   The time detection lag is the delay that the system output has compared to the GT [8].

   ii. **Frame Rate**

   Frame rate is the frequency at which consecutive images called frames appear on a display. It is measured in Frames per second (FPS)
3. Results and Outcomes

3.1 Discussion of results

The results and approach employed in the project can be divided into the following sections:

3.1.1 Configuration of a standalone object detection system

One of the critical parts of the project so far include the setting up of the object detection module as a standalone application and configuring it to accept and process pre-recorded videos as well as a live camera stream and detect objects inside them. This system is capable of outputting a running stream of the input video or live stream with detected objects labelled and surrounded by bounding boxes. There are two neural networks that this object detection system supports, namely Tiny YOLO and YOLOv2.

Tiny YOLO is a lightweight version of the YOLO object detection network suite. It is known to be much faster than other YOLO implementations and can achieve object detection at up to 244 FPS on a GPU-powered computer [15]. On the other hand, Yolov2 is much more accurate than Tiny Yolo, operating at about 40 FPS on a GPU-powered computer.

For the object detection application, pre-trained versions of both Tiny Yolo and Yolov2 have been downloaded and used by adding a maven dependency to the project. They have been trained using the ImageNet dataset which contains a wide range and large quantity of pictures of everyday objects [16].

This standalone object detection module follows a multi-threaded implementation process. While this implementation is just a utility for the ultimate purpose of this project, its understanding is important for a reader to easily comprehend the computation parallelization structures discussed later in the report. Through the use of a Java UI (Refer to Figure 3.1 below), the neural network
of choice (Tiny Yolo or Yolov2) can be selected and a preferred speed can be determined using a dropdown menu with the options being:

- **Fast**: Highest speed, lowest accuracy
- **Medium**: Almost real-time with medium accuracy
- **Slow**: Slowest speed, highest accuracy

![User Interface](image_url)

*Fig. 3.1: This figure shows the user interface of the project. The preferred network (Yolov2 or Tiny Yolo) as well as the preferred speed-accuracy setting can be selected before detection is started.*

The speed settings correspond to variable height, width, grid height and grid width settings which are used to pre-process frames with before injecting them into the neural network to optimize performance to achieve variable levels of speed and accuracy.

At this point, detection can be started by pressing a “Start Detection” option. As soon as this is done, the preferred neural network model is downloaded and stored as a computation graph in the program memory. JavaCV is then summoned by the system to read the input camera stream and divide it into frames continuously using a main thread. Each of the frames read is then pushed to a global stack by this thread one by one.
Another java thread (secondary thread), operating in parallel, then retrieves each frame pushed to the stack by the main thread and then:

1. Pre-processes the frame’s image to fit the required grid dimensions of the Speedsetting and creates an **INDArray** (a special n-dimensional array) for the frame
2. Loads an output layer of the computation graph of the neural network
3. Uses the computation graph and frame’s INDArray to generate an INDArray for results
4. Uses the output layer loaded in (2) to generate a list of detected objects for the frame i.e List<DetectedObject>

In this scenario, **DetectedObject** class represents an object by storing its name, class and coordinates on a frame image (for drawing of bounding boxes later)

The main thread, in turn, retrieves this list of detected objects and uses a marking function to mark the bounding box for each object in the list onto the concerned frame, which is then displayed on the screen (Refer to Figure 3.2 for system layout and 3.3 for result screenshot)

This way, the basis of the object detection module of this project is set and has been tested on a 64-bit Linux machine.

The performance of the current system using Tiny YOLO is very far from optimum for a self-driving car. With a Top-1 accuracy of about 50%, Top-5 accuracy of 83% and a 30 frames per second (FPS) frame rate, the system needs major improvement in order for it to come close to the requirements of a safety critical application. Tested with Yolov2, the Top-1 accuracy is 70% at 20 FPS.
Fig 3.2: A figure showing the implementation of the standalone object detection system. Each extracted frame from a camera stream is processed by the neural network and marked with bounding boxes for each detected object inside it.
Fig. 3.3: This figure shows the comparison of object detection results for Tiny Yolo (top) and Yolov2 (bottom). It is clearly seen that Yolov2 has higher accuracy as compared to Tiny Yolo and detects more objects.
3.1.2 Distributed object detection using Spark

3.1.2.1 Implementation 1: Batch processing using Spark-Core

As mentioned in section 3.1.1, which describes the basic layout of the object detection system used in this project, the processing of frames by the neural network happens in a serial manner. As each frame is captured using a camera, it is preprocessed and fed into the neural network computation graph. Next, a list of detected objects for that frame is collected as output from the computation graph and a marking function is used to mark bounding boxes on the concerned frame based on the list of detected objects found. Finally, this frame is displayed on the screen with the object detection results.

This entire process induces a latency in object detection as each frame has to go through this long process before another frame can be processed. With the neural network operations being computationally intensive and time consuming and the frame preprocessing and post-processing functions being expensive as well, there seemed to be a potential for improvement in the system architecture.

This is where the use of Apache Spark was employed in order to reduce the processing latency and attempt to quicken the speed of object detection. In particular, Spark Core was introduced in the system to observe any changes. Spark Core is the foundation of Apache Spark and provides distributed task dispatching, scheduling, and basic I/O functionalities. Core uses a fundamental data structure called RDD (Resilient Distributed Datasets) that is a logical collection of data partitioned across machines. [14]

In this implementation, the spark cluster is used in the following way. The spark cluster is started by providing the application the IP address of the master node and loading the application jar file with dependencies onto the master node. When the camera input stream starts, 10 frames are collected in one instance and pushed to a stack by the main thread.
The secondary thread then, in turn, pops these 10 frames, pre-processes them and creates an INDArray (array that stores a frame object as a vector) object for each of them using the computation graph. Now, a JavaRDD<INDArray> object is created using Spark’s ‘parallelize’ function that contains the 10 INDarrays. The creation of the RDD object is necessary for Spark to distribute computation over the cluster.

Next, the spark ‘map’ function is used to forward the components of the RDD object to spark worker nodes along with an instance of the output layer of the neural network computation graph. Spark distributes the payload of the RDD object into its worker nodes and each worker node then feeds the INDArray object that it receives into the instance of the computation graph output layer loaded inside it. The output is a list containing the list of detected objects for each frame’s INDArray. This output is returned by the worker nodes to the master, where the results can be stored using the ‘collect’ function. As a detected object itself is not from a serializable class and Spark only allows the use of serializable classes as input and output objects, a serializable wrapper (SerializedDetectedObject) was created for detected objects to carry them back as results from Spark worker nodes.

The result of the distributed computation includes 10 separate lists of detected objects for each of the frames that were distributed into the worker nodes. After unwrapping, these lists are now used to mark the concerned frames with bounding boxes for detected objects and the main thread now takes over to display them one by one.

The following software were used in the development of this system:

- JDK 1.8
- Apache Maven 3.5.2
- OpenCV 3.4.1
- Spark 2.4.0
This implementation and distribution of spark jobs can be seen in the following diagram:

Fig 3.4: This diagram shows the system sequence of object detection with Spark. A batch of frames as well as an instance of the neural network output layer is distributed into worker nodes and object detection results for the batch are collected and displayed at the master node.
3.1.2.2 Implementation 2: Real time processing using Spark-Streaming

Spark streaming is an extension of core Spark API that allows scalable, high-throughput stream processing of live data streams. [14] Streaming data can be gathered from various sources such as Apache Kafka, Flume, Kinesis, or TCP sockets, and can be processed and expressed with algorithms or high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to databases or storage systems, or even broadcasted or displayed live.

As the scope for this project stresses the need to cut down processing time for object detection by harnessing the power of distributed computing, it was a natural choice to use Spark Streaming to analyze how the implementation of YOLO can be applied to a live camera stream on the go.

Component 1: Video Stream Collector

To achieve this, the first challenge was to create a live stream from the frames read through the camera input in one of the compatible input stream formats. After careful research, Apache Kafka was selected as the carrier stream for the frames for numerous reasons. As a stream of frames needs to be stable and prevent loss, Kafka was a natural choice to create a reliable stream.

Kafka as a streaming platform has many capabilities including publishing and subscribing to streams of messages in queues and storing streams with high fault tolerance. To achieve this with efficiency, Kafka needs the help of Apache Zookeeper which is a top level software that acts as a centralized service to provide flexible synchronization.

For the streaming service of this project to be configured, the Zookeeper (version 3.4.8) service needs to be activated and the Kafka server (version 2.11) needs to be running. Next, a Kafka topic is created to transport the stream. The topic is basically the service name that the stream is broadcasted with and can be subscribed to when the stream needs to be captured and collected on another end. Now, the Kafka server is ready and waiting to produce a stream and the Video Stream Collector module is configured to subscribe to the Kafka server instance with appropriate
settings such as maximum transport batch size, message format, topic name and compression type etc.

However, as Kafka is a message transport service at the most basic level, the frame objects of the camera stream have to be serialized and converted into a transportable format. In order to do this, the Mat objects of each camera image frame are converted into a JSON object as soon as they are extracted. For each Mat object, the rows, columns, type, timestamp of collection and byte array is extracted and put inside a JSON object. The byte array is encoded as a base-64 string before being stored in the JSON object.

As soon as the JSON object for each captured frame is ready, it is sent to the Kafka server using the appropriate topic name and the Kafka server then in turn streams the serialized frame JSON as a live streamed message. The Kafka server uses the local file system as a stream buffer to store streaming data in order to prevent loss. This storage is also important since the future processing of this data is a much slower process than frame collection and the data needs a point of storage to queue in.

**Component 2: Video Stream Processor**

Now that a live Kafka stream is all set and being broadcasted over a server, the next step is to create a sink to collect it. To achieve this, a Video Stream Processor module has been set up.

First of all, the processor has configuration settings to subscribe to the same Kafka server and topic that the stream is running on. Once this is done, a Spark-Streaming instance is created and pipelined directly to the Kafka stream. This Spark instance, called SparkSession is configured to distribute computation over multiple worker nodes as it is being handled by the job scheduler of the main node on the master node Linux machine.

To generalize the JSON data being received, the spark streaming instance collects the JSON objects into a Spark Dataset of the type VideoEventData i.e.

```
Dataset<VideoEventData>
```

Kerney Mohammad Owais
This collection is done using Spark’s bean encoder that converts the JSON messages into objects of the Java bean class called VideoEventData. The <VideoEventData> class contains the timestamp, rows, columns, type and data of each frame object and stores them in a serialized manner.

Next, Spark’s ‘groupByKey’ function is used to transform this dataset into a Key Value Grouped dataset, to make it easy to identify incoming data and label it for further processing, if in case multiple cameras are being used to send in the input stream or the incoming stream is being channeled from multiple sources.

Next, to actually process the incoming camera stream from the Kafka stream and detect objects inside it, a Spark Streaming Query is used. The dynamics of this query work in the following way. As each batch of JSON objects is received at the processor and converted into a key, value grouped dataset, Spark Streaming Query distributes this batch into each of the executors of the worker nodes using a ‘mapGroupsWithState’ function. This function divides and injects a part of the batch into each worker node and collects the result from them.

In each worker node then, the Object Detection System based on YOLO (packaged into the same Video Stream Processor component) and discussed earlier is used to:

1. Unpack the VideoEventData object and convert it to a Mat object again
2. Process it using an output layer of the YOLO neural network
3. Mark it with bounding boxes based on detected objects obtained in step 2
4. Convert the processed frame into a new VideoEventData object and return to master node

The returned result which is collected at the master node is unpacked into a Mat object and displayed on the screen for each frame in the processed batch.

This entire process of being subscribed to the Kafka server stream, receiving every batch of camera input frames, transforming them for use at the master node, distributing the object
detection mechanism onto worker nodes, collecting results and displaying them, happens in real-time and continuously goes on until there are no more batches left to be processed from the Kafka stream i.e. the camera stream has been shut down from the Video Stream Collector component.

The following software were used in the implementation of this system:

- JDK 1.8
- Apache Maven 3.5.2
- Apache Zookeeper 3.4.8
- Apache Kafka 2.11-0.10.2.0
- OpenCV 3.4.0
- Apache Spark 2.4.0

This implementation can be visualized in the diagram (Figure 3.2) on the next page.
Fig. 3.5: This figure shows the Spark-Streaming based implementation of the project. A Kafka stream is created for the camera input and distributed into worker nodes. The output from the worker nodes consists of frames marked with bounding boxes for detected objects and it converges to be displayed at the master node.
3.2 Testing Results

3.2.1 Testing Methodology

Three types of tests were carried out:

1. Structured in-lab tests using a webcam directly connected to Spark master node
2. Tests on drive time videos played on a monitor screen and captured by pointing the webcam at the monitor (to simulate drive time videos in real-time)
3. Live webcam stream captured and transmitted as an HTTP Link over CS VPN

**Structured in-lab tests**

**Equipment**

The testing results in this section were obtained by connecting a webcam directly to one of the Linux machines in the HKU CoC Server (machine: student-79). See figure below:

*Fig. 3.6: Common lab objects used to test the object detection system in the structured in-lab tests*
**Camera Stream**

The camera stream was obtained using OpenCV in the main Java program and then used for object detection purposes.

**Testing environment**

Multiple objects (including those found commonly on roads as well as anomalies) were placed in the computer lab where the Linux machine is located. The objects included: Human subjects, a giant cardboard (to simulate walls/pavements), umbrella, chairs, tables, monitors. The lab environment was kept well lit and bright.

**Testing variables**

The system was tested with:

1. A variable number of executors (worker nodes): 2 and 4
2. Both YOLO versions (Tiny YOLO and Yolov2)
3. Two speed settings (Fast with low accuracy and Slow with higher accuracy)

**Testing parameters**

- Frames per second (FPS) of the output stream with object labels
- Accuracy
- Time detection lag
- Spark Cluster Analysis

**Testing Strategies**

To test for the frame rate, the DL4J API was used to collect timestamps of frames as they were received and compared to the time at which they were processed and displayed.

To test for accuracy, the number of total detectable objects were counted in multiple scenarios and each object successfully detected, omitted, or wrongly detected was noted down manually.

To test for time detection lag, various objects were brought and removed from in front of the camera stream and the time it took for the objects to be detected was measured.

**3.2.2 Results: Implementation 1 (Spark Batch Processing)**

The results for the first three testing parameters are reported in the table below and the Spark Cluster Analysis follows afterwards. After the Spark Analysis, the result is analyzed and all testing parameters are compared.
### Fig. 3.7: This figure shows the comparison results for both neural networks using 4 executors

<table>
<thead>
<tr>
<th></th>
<th>Tiny Yolo</th>
<th>Yolo V2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed Setting</strong></td>
<td>Slow</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>FPS</strong></td>
<td>12</td>
<td>7-8</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>60%</td>
<td>90%</td>
</tr>
<tr>
<td><strong>Time Delay</strong></td>
<td>5-6 sec</td>
<td>9-11 sec</td>
</tr>
</tbody>
</table>

### 4 Executors

<table>
<thead>
<tr>
<th></th>
<th>Tiny Yolo</th>
<th>Yolo V2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed Setting</strong></td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td><strong>FPS</strong></td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>50%</td>
<td>75%</td>
</tr>
<tr>
<td><strong>Time Delay</strong></td>
<td>0.5-1 sec</td>
<td>2-3 sec</td>
</tr>
</tbody>
</table>
Fig. 3.8: This figure shows the comparison results for both neural networks using 2 executors.

<table>
<thead>
<tr>
<th></th>
<th>Tiny Yolo</th>
<th>Yolo V2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed Setting</strong></td>
<td>Slow</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>FPS</strong></td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>55-60%</td>
<td>90%</td>
</tr>
<tr>
<td><strong>Time Delay</strong></td>
<td>5-7 sec</td>
<td>11-14 sec</td>
</tr>
</tbody>
</table>

2 Executors

<table>
<thead>
<tr>
<th></th>
<th>Tiny Yolo</th>
<th>Yolo V2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed Setting</strong></td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td><strong>FPS</strong></td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>50%</td>
<td>70%</td>
</tr>
<tr>
<td><strong>Time Delay</strong></td>
<td>1 sec</td>
<td>2-3 sec</td>
</tr>
</tbody>
</table>
3.2.2.1 Spark Analysis-Implementation 1

For implementation 1, the following results include:

1. The spark event timeline to see when each job returns results
2. The executors’ summary to analyse the resources and tasks in each executor
3. The specific DAG for one chosen Stage that has been concluded successfully during processing
4. The event timeline and summary for each executor for the same selected stage as in 3

As far as implementation 1 is concerned, the following functions were performed in a distributed manner using Spark:

- **parallelize** (to create RDD objects from frames)
- **map** (distributing batches of frames to be processed by the neural network)

(Note: Each Spark job corresponds to only one Stage in this implementation)

Some common Spark timeline terms are described as follows:

- **Scheduler Delay**: The time it takes for the Spark master driver to deliver the job to the worker node.
- **Task Deserialization**: The time it takes for a task to be deserialized and prepared for operations.
- **Executor Computation Time**: The time it takes for the actual execution (object detection functions in our case) to finish

The general DAG diagram for this stage (job) is shown below:
Testing with 4 worker nodes

4 worker nodes (executors) were set up on 4 machines as shown below:

1. 10.42.0.88:41895
2. 10.42.0.87:37219
3. 10.42.0.84:39177
4. 10.42.0.86:36883

All 6 cores of each worker were utilized and 6 GB of memory on each executor was occupied. The driver occupied 6 GB of memory on the master node. See figure below:

Fig. 3.9: This figure shows the 4 spark executors used in testing including their IP addresses.
**Network:** Tiny Yolo  |  **Speed:** Fast  |  **Spark workers:** 4

Tiny Yolo -FAST

**General Spark Timeline**

**Executors Summary**

*Fig. 4.0: These figures show the spark timeline and executor summary for Tiny Yolo with 4 executors in Fast mode*
Network: Tiny Yolo

Speed: Fast

Spark workers: 4

Executor Specific Timeline

*Fig. 4.1: This figure shows the executor specific Spark timeline for Tiny Yolo with 4 executors in Fast mode*
Network: Tiny Yolo  |  Speed: Slow  |  Spark workers: 4

Tiny Yolo - Slow

General Spark Timeline

![Spark Jobs Timeline]

**Executors**

<table>
<thead>
<tr>
<th>Executors</th>
<th>Summary</th>
<th>RDD Blocks</th>
<th>Storage Memory</th>
<th>Disk Used</th>
<th>Core</th>
<th>Active Tasks</th>
<th>Failed Tasks</th>
<th>Complete Tasks</th>
<th>Total Tasks</th>
<th>Task Time (UC Time)</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
<th>Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>972</td>
<td>972</td>
<td>15 h 14 m</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Dead</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>972</td>
<td>972</td>
<td>15 h 14 m</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Executors**

<table>
<thead>
<tr>
<th>Executors</th>
<th>Summary</th>
<th>RDD Blocks</th>
<th>Storage Memory</th>
<th>Disk Used</th>
<th>Core</th>
<th>Active Tasks</th>
<th>Failed Tasks</th>
<th>Complete Tasks</th>
<th>Total Tasks</th>
<th>Task Time (UC Time)</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
<th>Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>972</td>
<td>972</td>
<td>15 h 14 m</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Dead</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>972</td>
<td>972</td>
<td>15 h 14 m</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Fig. 4.2:** These figures show the spark timeline and executor summary for Tiny Yolo with 4 executors in Slow mode

Kerney Mohammad Owais
Network: Tiny Yolo  Speed: Slow  Spark workers: 4

Executor Specific Timeline

![Executor Specific Timeline](image)

**Fig. 4.3:** This figure shows the executor specific spark timeline for Tiny Yolo with 4 executors in Slow mode.
Fig. 4.4: This figure shows the Spark timeline and executor summary for Yolov2 in Fast mode with 4 executors.
Executor Specific Timeline

Fig. 4.5: This figure shows the executor specific spark timeline for Yolov2 with 4 executors in Fast mode
**Network:** Yolov2  |  **Speed:** Slow  |  **Spark workers:** 4

**Yolov2 - Slow**

**General Spark Timeline**

**Executor Summary**

---

**Fig. 4.6:** This figure shows the Spark timeline and executor summary for Yolov2 in Slow mode with 4 executors
Fig. 4.7: This figure shows the executor specific Spark timeline for Yolov2 with 4 executors in Slow mode
Testing with 2 worker nodes

2 worker nodes (executors) were set up on 2 machines as shown below:

1. 10.42.0.88:41895
2. 10.42.0.87:37219

All 6 cores of each worker were utilized and 6 GB of memory on each executor was occupied. The driver was also allotted 6 GB of memory on the master node.
Network: Tiny Yolo  |  Speed: Fast  |  Spark workers: 2

Tiny Yolo - Fast

General Spark Timeline

Fig. 4.8: This figure shows the Spark timeline and executor summary for Tiny Yolo in Fast mode with 2 executors
**Network**: Tiny Yolo  |  **Speed**: Fast  |  **Spark workers**: 2

**Executor Specific Timeline**

*Fig. 4.9: This figure shows the executor specific spark timeline for Tiny Yolo with 2 executors in Fast mode*
Network: Tiny Yolo  |  Speed: Slow  |  Spark workers: 2

Tiny Yolo - Slow

General Spark Timeline

Executors

Executor Summary

*Fig. 5.0: This figure shows the spark timeline and executor summary for Tiny Yolo in Slow mode with 2 workers*
**Network:** Tiny Yolo  | **Speed:** Slow  | **Spark workers:** 2

**Executor Specific Timeline**

*Fig. 5.1: This figure shows the executor specific spark timeline for Tiny Yolo with 2 executors in Slow mode*
Fig. 5.2: This figure shows the spark timeline and executor summary for Yolo v2 in Fast mode with 2 workers
Network: Yolov2 | Speed: Fast | Spark workers: 2

Executor Specific Timeline

Fig. 5.3: This figure shows the executor specific spark timeline for Yolo v2 with 2 executors in Fast mode
Network: Yolov2  |  Speed: Slow  |  Spark workers: 2

Yolov2 - Slow

General Spark Timeline

Fig. 5.4: This figure shows the spark timeline and executor summary for Yolo v2 in Slow mode with 2 workers

Kerney Mohammad Owais
Network: Yolov2  Speed: Slow  Spark workers: 2

Executor Specific Timeline

**Fig. 5.5:** This figure shows the executor specific spark timeline for Yolo v2 with 2 executors in Slow mode.
3.2.2.2 Analysis - Tiny Yolo with 4 workers vs Tiny Yolo with 2 workers

With the Fast speed setting, the highest frame rate was 22 FPS achieved using 4 executors. Hence, a speedup was observed with more workers and more workers definitely led to a better performance. The time delay while using 4 executors was only slightly lesser than the time delay of detection achieved using 2 executors. Not achieving a much larger boost in performance by increasing the number of executors was an anticipated situation which can be attributed to the slow network connection between the master and worker nodes. This is evident by the finding that Scheduler Delay occupies a higher amount of time in each executor while using 4 executors as compared to 2. Even though actual computation performance is theoretically supposed to increase with 4 executors, having 2 extra workers induces added latency for Spark’s job scheduler which decreases the speedup achieved significantly. As far as accuracy is concerned, it was more than slightly better with 4 executors. This can be attributed to the fact that each frame got more processing time as more executors were performing frame detection.

The most time-consuming operations inside each executor while operating Tiny Yolo were Scheduler Delay, Task Deserialization and Executor Computing Time.

When comparing the Slow speed setting to Fast using 4 executors, Scheduler Delay occupied a much larger portion of time as compared to Task Deserialization while the ratio was almost 1:1 for the Fast Setting. Executor Computing Time was the shortest for both settings and comparable in each.

Using 2 executors, the Slow setting had a higher proportion of Scheduler Delay as compared to Task Deserialization Time while it was the opposite for the Fast setting. Executor Computation Time was the least time-consuming process for both settings.

Analysis - Yolov2 with 4 workers vs Yolov2 with 2 workers

When comparing the performance of Yolov2 with 4 executors against 2 executors, a slight speedup was observed with 4 executors. As far as accuracy is concerned, it was more than slightly better with 4 executors. This can be attributed to the fact that each frame got more processing time as more executors were performing frame detection.

The most time-consuming operations inside each executor when operating Yolov2 were Task Deserialization, Scheduler Delay and Executor Computing Time.

When comparing the Slow speed setting to Fast using 4 executors, more time was spent by each executor in Scheduler Delay as compared to Task Deserialization while in the Fast setting, Task
Deserialization had the highest proportion of time spent. Getting results occupied the least time for both speed settings for Yolov2 on 4 executors.
A similar trend was observed in the results for each speed setting of Yolov2 using 2 executors. Scheduler Delay, however, was the most time-consuming process for both settings while the ratio of Scheduler Delay to Task Deserialization was larger for the Slow speed setting as compared to that for the Fast speed setting.

3.2.3 Results: Implementation 2
The results for the first three testing parameters are reported in the table below and the Spark Cluster Analysis follows afterwards.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Time Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Spark</td>
<td>50%</td>
<td>5-8 sec</td>
</tr>
<tr>
<td>1 worker</td>
<td>50%</td>
<td>5-8 sec</td>
</tr>
<tr>
<td>2 workers</td>
<td>50%</td>
<td>6-9 sec</td>
</tr>
</tbody>
</table>

**New test**

*Fig. 5.6: This figure shows the comparison results for Tiny Yolo with local spark, 1 executor and 2 executors*
3.2.3.1 Implementation 2- Spark Cluster Analysis

For implementation 2, the following results include:

1. The spark event timeline to see when each job returns results
2. The executors’ summary to analyse the resources and tasks in each executor
3. The event timeline and summary for each executor for both stages of the same Job as in 3

Note: Each job in this implementation has two stages, as shown and described in the general DAG diagram below:

![DAG Diagram for Implementation 2](image)

Fig. 5.7: This figure shows the DAG diagram for Implementation 2
The first stage, which will be referred to as Stage A, is responsible for fusing multiple operators in the execution into a single Java function to optimize execution performance. The second stage, Stage B, performs the actual object detection tasks using YOLO as well as the map functions for collecting and packing results.

**Testing with 1 worker node**

1 worker node (executor) was set up on:

1. 10.42.0.88:41895

All 6 cores of the worker were utilized and 5 GB of memory on the executor was occupied. The driver occupied 2 GB of memory on the master node. See figure below:
General Spark Timeline

Executors Summary

---

*Fig. 5.8: This figure shows spark timeline and executor summary for Tiny Yolo with 1 executor (1 master + 1 worker) for Implementation 2*
Stage A: Executor Specific Timeline

**Fig. 5.9:** This figure shows spark timeline for executors in Stage 1(A) for Implementation 2 with 1 executor
Stage B: Executor Specific Timeline

**Fig. 6.0:** This figure shows spark timeline for executors in Stage 2(B) for Implementation 2 with 1 executor
Testing with 2 worker nodes

2 worker nodes (executors) were set up on:

1. 10.42.0.88
2. 10.42.0.87

All 6 cores of the worker were utilized and around 6 GB of memory on each executor was occupied. The driver occupied 2 GB of memory on the master node.
Network: Tiny Yolo  |  Speed: Fast  |  Spark workers: 2

General Spark Timeline

Executor Summary

---

**Fig. 6.1:** This figure shows spark timeline and executor summary for Implementation 2 with 2 executors
Stage A: Executor Specific Timeline

Fig. 6.2: This figure shows spark timeline for executors in Stage 1(A) for Implementation 2 with 2 executors

Kerney Mohammad Owais
Stage B: Executor Specific Timeline

Fig. 6.3: This figure shows spark timeline for executors in Stage 2(B) for Implementation 2 with 2executors
Network: Tiny Yolo | Speed: Fast | Spark workers: 1 (Local)

Testing with single executor on local machine (Spark local)

General Spark Timeline

Executor Summary

---

**Fig. 6.4:** This figure shows spark timeline and executors summary for Implementation 2 with local spark

Kerney Mohammad Owais
Stage A: Executor Specific Timeline

![Image of Spark timeline for executors in Stage 1(A) for Implementation 2 with local Spark]

**Summary Metrics for 1 Completed Tasks**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Min</th>
<th>25th percentile</th>
<th>Median</th>
<th>75th percentile</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>15</td>
<td>45</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>GC Time</td>
<td>0.4s</td>
<td>0.4s</td>
<td>0.4s</td>
<td>0.4s</td>
<td>0.4s</td>
</tr>
<tr>
<td>Peak Execution Memory</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle Write Size/Recorded</td>
<td>51.4MB/102</td>
<td>51.4MB/102</td>
<td>51.4MB/102</td>
<td>51.4MB/102</td>
<td>51.4MB/102</td>
</tr>
</tbody>
</table>

*Fig. 6.5: This figure shows spark timeline for executors in Stage 1(A) for Implementation 2 with local Spark*
Network: Tiny Yolo | Speed: Fast | Spark workers: 1 (Local)

Stage B: Executor Specific Timeline

![Spark Timeline for Executors in Stage 2(B) for Implementation 2 with local Spark](image_url)

**Fig. 6.6:** This figure shows spark timeline for executors in Stage 2(B) for Implementation 2 with local Spark

Kerney Mohammad Owais
3.2.3.2 Result Analysis- Implementation 2 (Only tested on YOLO at Fast Speed)

The Executor Specific timelines of the three tests (2 workers, 1 worker and local spark) are shown together below to make it easy to visualize the analysis below. (See figure 4.2 on the next page)

Accuracy was found to be consistent with all three testing environments. However, the speed of execution was observed to improve slightly by using more workers and executors. At first, speed of execution was observed to decrease with an increase in executors. This was recently found to be due to an inapt distribution of Spark tasks due to configuration issues in the executors. Once these were improved, the execution speed increased slightly.

For the testing results using 2 executors on two different machines, the Executor Computing Time occupied the largest chunk of execution time for each executor. Task Deserialization was next and Scheduler Delay was the least expensive operation. This is expected as the object detection computation to be performed by the executors is an expensive operation.

The same trend was observed for both the results with 1 worker node (1 external executor) as well as the results with local Spark executor (1 executor on the same machine as the master node).

An interesting result to note was the fact that Executor Computation time occupied a significantly higher chunk of execution time for the local Spark results, as both the driver and executor had to divide the same RAM to perform tasks and hence less memory was available for the neural network operations.

However, Scheduler Delay occupied a larger chunk of time while using 2 worker nodes as compared to local Spark. This is understandable and an expected result as the Linux cluster has a slow network connection and the sending of tasks as well as receiving of results by the scheduler has higher latency when using more worker nodes. This has also been identified as a potential obstacle in increasing execution speed by a significant amount.

It is anticipated that execution computation time is actually lesser when using a higher number of workers which provides a hint of evidence that Spark Streaming with more workers is making object detection quicker if Scheduler Delay is ignored.

Further tests were recently conducted to improve the distribution of tasks amongst the 2 workers. By using 3gb of memory and 3 cores of each executor, improvement was observed in the object detection delay as it went down to 4-7 seconds. See figure below:

---

Kerney Mohammad Owais
Fig 6.7: This figure compares the timelines for Spark executors with local spark (top), 1 executor (middle) and 2 executors (bottom)
3.2.4 Tests on drive time videos played on a monitor

**Equipment:** Pre-recorded drive time videos were played on a computer monitor and a webcam connected directly to one of the Linux machines in the HKU Cluster Lab (machine: student-79) was setup to stream the drive time videos.

**Testing environment:** 2 pre-recorded videos were obtained from the internet. These videos were made using dashboard cameras of cars going around actual roads.

**Justification:** As any camera stream used in this project needed to be connected directly to the Linux machine with the Java program or at least be transmitted over CS VPN as an HTTP link, it was not feasible to test the object detection system on a real drive time environment. Hence, to get as close to simulating a drive time environment, pre-recorded videos were chosen.

**Results:** As this testing criteria was used as a formality to cover an actual drive time environment and was bound to be inaccuracy prone, the object detection results were only collected as a video of the object detection stream results and will be displayed in the final presentation.

3.2.5 Live webcam stream captured and transmitted as an HTTP link over CS VPN

**Equipment and setup:**
1. A Windows laptop, with a webcam, was connected to the HKU Network as well as the CS VPN
2. VLC Media player was used to create an ‘HTTP’ live stream for the webcam stream at (http://ip_address:8080”)
3. The stream was then retrieved using an OpenCV frame grabber using the above http link

**Testing environment:** The Windows laptop with an attached webcam was placed in the vicinity of a road near the HKU Centennial Campus.

**Justification:** As any camera stream used in this project needed to be connected directly to the Linux machine with the Java program or at least be transmitted over CS VPN as an HTTP link, it was not feasible to test the object detection system on a real drive time environment. Hence, to get as close to simulating a drive time environment, a live stream from a stationary point on a busy road was chosen.

This experiment was also useful in taking into account a lag in the video stream caused by network bandwidths and quotas, which are factors that a real-time autonomous driving object detection system is likely to face.

Kerney Mohammad Owais
**Results:** As this testing criteria was used as a formality to cover an actual traffic environment and was bound to be inaccuracy prone, the object detection results were only collected as a video of the object detection stream results.

There was a 15-20 second lag in sending and receiving the video input from the Windows machine to the object detection system in the lab. The lag was partially due to a slow Wi-Fi stream and mostly due to the delay in stream generation using VLC Media Player (as video embedding is a computationally expensive operation). The video results are embedded in the project power-point presentation.
3.3 Limitations/Difficulties encountered

The limitations/difficulties encountered and anticipated can be broken down into three parts:

- **Limitation concerning the object detection module**

  The object detection module solely relies on YOLO, a deep learning-based object detection system. YOLO is an open source platform, progressively being made better with each subsequent version. For example, Tiny YOLO, however faster and more accurate than other object detection systems, has a few shortcomings in detecting very small objects which are located far from the opening of image frame. Yolov2, while being more accurate than the Tiny version, is quite slow when it comes to processing a continuous stream of frames.

  YOLO version 3, the latest and most advanced version of YOLO, is known to be more accurate and quicker than the previous versions. However, to be used in a Java application, it needs to be implemented and trained on DL4J as a pre-trained version is yet to be available in the DL4J-Zoo repository. This process is a time consuming issue and is the sole reason why Yolo version 3 was not implemented in this project despite its higher accuracy and rich features.

- **Difficulties concerning the Apache Spark implementation**

  Apache Spark is one of the most advanced distributed computing platforms available in the world right now. It is compatible with a plethora of programming languages and offers support for multiple platforms. However, the online support available for Spark is mostly focused on Scala implementations. Requiring a lot of configuration and initial setup formalities, finding support for the Java implementation was a tough task.

  Setting up Spark in standalone cluster mode required installing spark on all the machines to be used as master and worker nodes, configuration of password less SSH access, startup of the nodes etc. Next, allocating enough resources including memory and cores was a tricky task as
well because spark worker nodes can easily crash in the middle of program execution due to running out of memory or space.

Next, as Spark can only deal with serialized objects/classes in any distributed implementation, some of the classes of the object detection system had to be covered with serializable wrapper classes to pass and distribute them using Spark.

All these configuration and setup issues took a big chunk of time in the project timeline before an impeccable Spark setup was established to run the various project implementations.

5. Conclusion

5.1 Summary and Future Direction

If self-driving cars are to win public trust and replace drivers on roads, it is extremely important that the existing research gaps be filled up and the subsystems optimized to be suitable for an intensively safety critical application.

Optimization of the process of object detection and tracking for self-driving cars is one of the most important hurdles faced by autonomous driving systems and requires serious technological improvements in order for them to be production ready. Using deep learning algorithms can greatly increase the speed and accuracy of these systems and this project attempts to use a deep learning based neural network called YOLO deployed on a cloud computing framework, Apache Spark to try and optimize the entire object detection and tracking process.

While the results achieved using Spark batch processing are promising and show a speedup in the system by increasing the number of executors, it would also be interesting to see how performance changes with Spark Streaming using a higher number of executors after more thorough testing. Overall, Spark has the potential to speed up object detection systems to make them faster and more suitable for self-driving cars.
References

[1] World Health Organization, 2018, Road traffic injuries
http://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries


Andrew Del-Colle, 2013, The 12 Most Important Questions About Self-Driving Cars, Popular Mechanics

Bryant Walker Smith, 2013, Human Error as a Cause of Vehicle Crashes, Ctr. For Internet and Society
http://cyberlaw.stanford.edu/blog/2013/12/human-error-cause-vehicle-crashes


Kerney Mohammad Owais
https://www.reuters.com/article/uber-crash/ntsb-uber-self-driving-car-failed-to-recognize-pedestrian-brake-idUSL2N1SV0NR


http://dx.doi.org/10.6028/NIST.IR.7972


https://github.com/bytedeco/javacv

Kerney Mohammad Owais
[12] YOLO-Real Time Object Detection, PJ Reddie, Official Website

https://deeplearning4j.org/

[14] Apache Spark- Official Website
https://spark.apache.org/


[16] ImageNet Database, ImageNet Official Website
http://www.image-net.org/