We are IntechOpen, the world's leading publisher of Open Access books
Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter

Computational Efficiency: Can Something as Small as a Raspberry Pi Complete the Computations Required to Follow the Path?

Toby White

Abstract

This chapter explains the development processes of a prototype autonomous toy car. It focuses on the design and implementation of transforming a normal remote control toy car into a self-contained vehicle with the capability to drive autonomously. This would be proven by making it follow a track of any layout. It uses a neural network (NN) in the form of a multilayer perceptron (MLP) to process images in real time to generate a movement instruction. Upon completion, the vehicle demonstrated the ability to be able to follow a track of any layout, while staying between both sides of the track. The collision avoidance system proved to be effective up to a distance of 50 cm in front of the vehicle in order to let it stop prior to hitting an object. The neural network processing of the image in order to classify it in a real time proved to be above the expectation of around 5 FPS and has an accuracy score of over 90%.

Keywords: raspberry pi, image recognition, classification, machine learning, neural networks

1. Introduction

Given the recent development of self-driving cars by companies such as Tesla, Google and others, it was of interest to attempt to replicate this on a smaller scale, by implementing a similar method on a small electronic toy car. An issue many developers in this industry are having is the issue of leaving the device connected to a wide area network (WAN) all the time, leaving the vehicles vulnerable to not only hacking but also vulnerable to being unable to make decisions in out-of-reach places such as the countryside where a WAN may not be available or only available intermittently. The decision was made to attempt the replication on a small scale, using a closed network with the computations occurring within the vehicle, as opposed to externally. The study was viewed as a proof of concept to test the possibility and feasibility, which could help lead research on full-scale self-driving cars and potentially also enhance the toy industry.

This study mainly focuses on a proof-of-concept implementation of artificial neural networks (ANNs) to classify road sign images in a real-time scenario

integrated with relevant image processing techniques. The study includes how the processed data outputted by ANN models is translated into defined movements of the car and whether it is suitably efficient to follow a track.

Neural networks are often used for image processing. Once the network is trained and the weighting values/structure is stored, all that would be required on future runs of the program would be to load these stored values/designs for the trained network for it to have the ability to run correctly. This means that once the network is trained, real-time image classification and thus prediction of the movement become very efficient. This is a beneficial method for a project of this sort, as the car is driving in real time and thus computations must be made in real time.

The overall objective was to allow the vehicle to drive autonomously around an unknown track with little to no error, utilising collision avoidance.

2. Background and literature survey

This project is a proof-of-concept study to demonstrate how existing ANN state-of-the-art knowledge and related theory can be used for developing an autonomous toy vehicle and letting it to operate without the need to use external computations. It has been adopted as a research problem, which means it has resulted from a lack of current evidence into the subject matter, and thus this project is an attempt at proving that such a concept is not only theoretically feasible but actually possible.

2.1 Hardware

The system is comprised of three subsystems: an input unit (sensors, camera), the processing unit (Raspberry Pi) and an output unit (L293D motor controller connected to Raspberry Pi).

The input required two front-placed ultrasonic sensors both at a 15° left and right angle from the vehicle's direct line of sight. This is mainly used to ensure no objects are within a certain distance from the front of the vehicle (this prevents collisions) and a Pi Camera which is used to stream video data. This data is used in the processing subsystem by the neural network to calculate the direction and recognise the track and stop signs. Both peripherals are constantly sending data to the Raspberry Pi.

HC-SR04 sensors, raspiCam and L293D chip were the best hardware options of obvious choice—because they are all the most commonly used [1, 2] supported and documented hardware in the Raspberry Pi.

The processing unit handles multiple tasks such as dealing with input data, scaling down and applying necessary imaging filters to the data received from the camera, calculating distances based on data received from the ultrasonic sensors to be used for object detection, controlling the motors via L293D chip, utilising a neural network to process the data from the camera to recognise the track ahead and further recognise on-track stop signs and assigning a corresponding movement instruction based on this data.

The output unit consists mainly of the hardware which is wired to the Pi such as the motor controller and the motors themselves, both of which are controlled by the Pi.

This report follows a structure where I outline the research into my theory, the design process of the vehicle itself including the neural network, implementation and finally testing processes involved in proving this as a concept.

2.2 Hardware libraries

A variety of libraries/application programming interfaces (APIs) were required to implement the hardware in C++:

WiringPi: "WiringPi is a PIN based GPIO access library written in C for the BCM2835 used in the Raspberry Pi. It is released under the GNU LGPLv3 licence and is usable from C, C++ and RTB (BASIC) as well as many other languages with suitable wrappers" [3]. WiringPi allows the user to apply and change power levels to the peripheral devices which are installed [4]. This results in a lot smoother code interface which avoids system calls to use the hardware, in particular, the use of the "softPwm" function which allows easy and convenient power on/off capabilities which will be necessary for sending "pulses" of power to the ultrasonic sensors.

libv4l (Video 4Linux): "libv4l is a collection of libraries which adds a thin abstraction layer on top of video 4linux2 devices. The purpose of this (thin) layer is to make it easy for application writers to support a wide variety of devices without having to write separate code for different devices in the same class." [5]. The libv4l library will allow the Pi Camera to be recognised as an external camera through the Pi, which allows extra functionalities on image processing and avoids the need of a system call through the C++ code to use it.

PIGPIO: "pigpio is a library for the Raspberry which allows control of the General-Purpose Input Outputs (GPIO). pigpio works on all versions of the Pi." [3].

These three libraries would allow me to use the camera through the Pi, control the motors' directions and send the relevant pulses to the ultrasonic sensors to calculate a distance regarding object detection.

2.3 Image processing

The image stream from the camera is a 60 FPS video stream in 1920 × 1080 resolution. This obviously is too big to be fed into any kind of machine learning algorithm and would require huge amounts of processing, which, given the relatively low computational power of a Raspberry Pi, would not be efficient enough to give the car the ability to work in real time.

To combat this, a variety of image processing techniques can be applied. For example, the image would need to be scaled down to 10x10 which still is of high enough resolution to recognise lines but is more computable.

An interesting article regarding image processing by Rosebrock showed a very useful image processing process which would be very beneficial for my type of project called canny edge processing:

- "Step 1: Smooth the image using a Gaussian filter to remove high frequency noise.
- Step 2: Compute the gradient intensity representations of the image.
- Step 3: Apply non-maximum suppression to remove "false" responses to edge detection.
- Step 4: Apply thresholding using a lower and upper boundary on the gradient values.
- Step 5: Track edges using hysteresis by suppressing weak edges that are not connected to strong edges" [6].

The following results are presented in **Figure 1**.

It appeared to work quite well, with adaptable options regarding the strength of the effect. It was decided an automatic level should be applied for ease of use. It is clear that this is easily adaptable for the issue of recognising two lines either side of the vehicle and would (assuming a plain background) solve the problem very efficiently.

This reduction in noise on the image would also help with real-time processing and stop the issue of anomalous results which would cause the vehicle to potentially go off-track.

2.4 General architecture of an autonomous vehicle

This is a generalised diagram of how these types of devices usually run.

Machine learning algorithms are often used in self-driving cars due to their advanced capabilities to be adapted for use in image processing. The most popular form of machine learning algorithm is known to be artificial neural networks, which are very good, in that a NN can be trained methodically for nearly every given circumstance imaginable. It can be highly adaptable in terms of size/computing power required and can be designed in any given way relevant to the user's needs. To do this, calculation and specification of the number of nodes in each layer need to accurately produce the required outputs (**Figure 2**).

In a NN, the connections between each node have a "weighting" value which is applied to the data which passed through it; these values are determined through training via the use of "training data" and then stored in a final version which is called the "model". The model can then be used with unseen data in real time to produce an accurate output; this is called "testing" via the use of "test data". Adapting these values to be correct for all input circumstances is known as the aforementioned "training"; this can be done through a variety of different methods, one of which is backpropagation. When training is complete, the weighting values can be stored, and thus further usage of the NN from then on will only require loading these stored values to make it run correctly (provided no changes have been made to the design of the NN) [7]. This means that once the network is trained, prediction of the movement becomes very fast. This is beneficial especially for real-time computation projects, as computation is done at a very efficient rate.

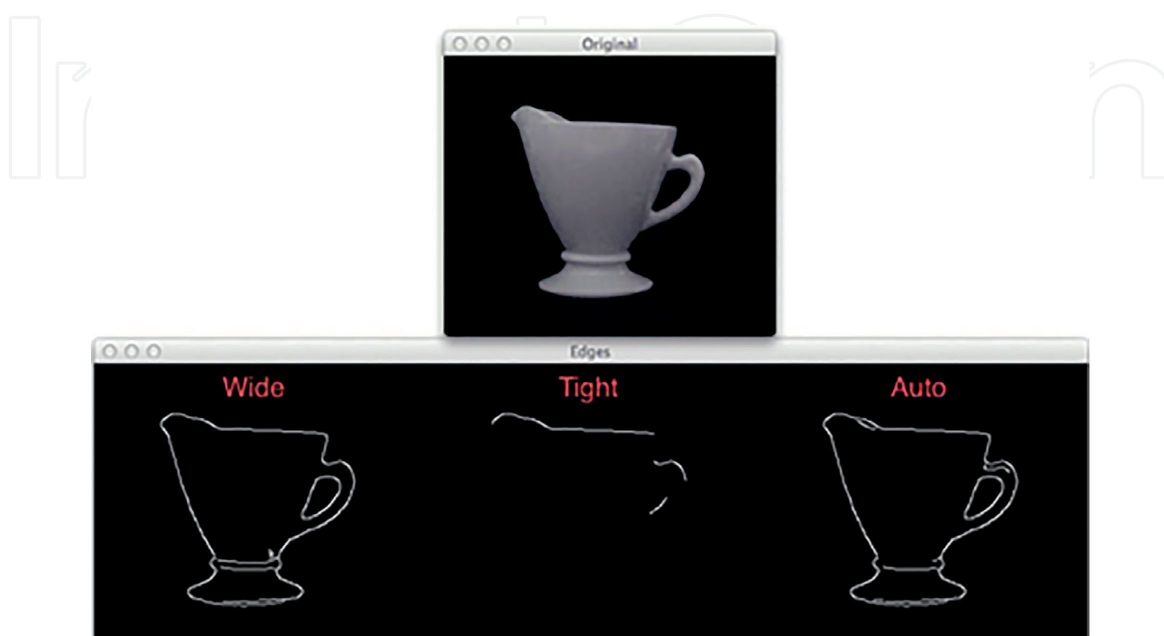

Figure 1.
An overview of canny edge detections [6].

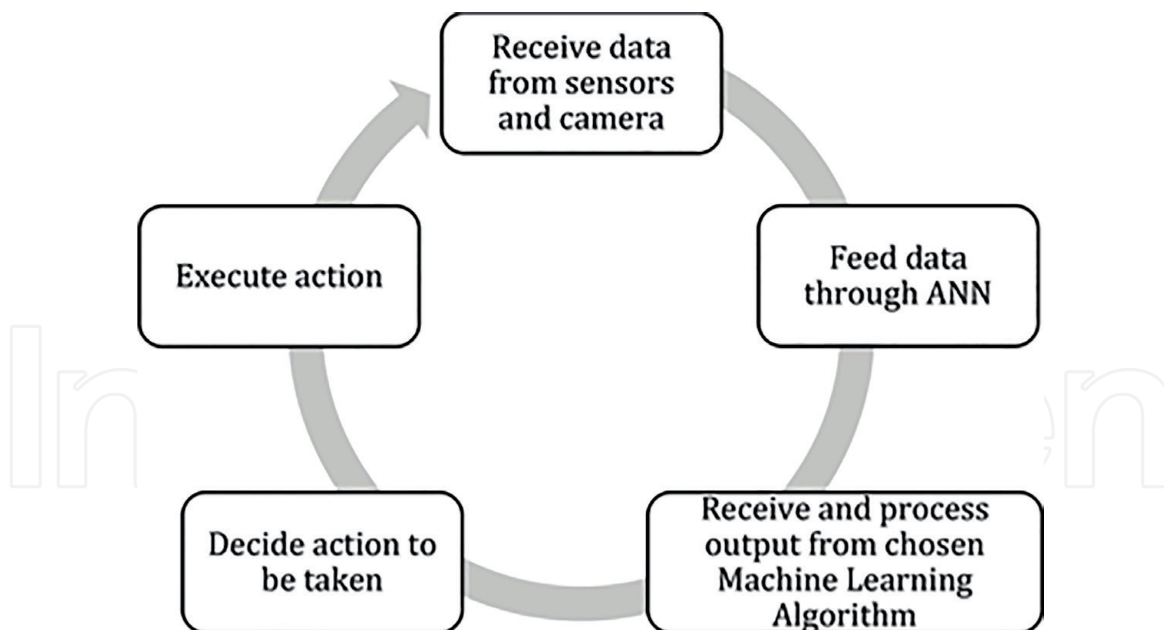

Figure 2.
Autonomous car system overview.

The middle (hidden) layers of the neural network are used to recognise patterns, such as in this case, the edges of the track, which are conceived from the input data. The output layer interprets these patterns to generate a probability for each output being true. These probabilities are then interpreted to determine whether to turn left/right or stay straight and go forwards or backwards.

By installing a Raspberry Pi, camera and ultrasonic sensors, the aim was to give the Raspberry Pi capabilities of driving the car through a written piece of software which is be run using C/C++. The intention was to create, within this, a form of artificial intelligence called a neural network. The neural network acts as the “brain” of the device and after extensive training can be used for image processing/classification of real-time camera data in order to aid the computation of real-time decisions on which movement instruction the car should follow during the movement step. This allows the device to drive autonomously and thus be adaptable to different situations. It does this by recognising two white lines either side of the vehicle, known as the “track” which the car should stay between.

2.5 Related works

Considered to be the first “self-driving remote-controlled (RC) car” design was done in 2015 by “Team Pegasus”, a team of graduates from the Gothenburg University. They used Arduino and Raspberry Pi to build a robot car, integrated with a remote-controlled software application running on an Android device [8]. Team Pegasus is the first creator of an Android-powered RC car using similar computation methods; however, they used an external server which was connected to the mobile device to perform the computing process.

The only other similar project found was done by Zheng Wang using Python and utilising a NN to process the images. Again, this used an external server connected to the Pi network to do the computing. This was a very small-scale device but worked very well. Wang also utilised OpenCV—an open source computer vision library which has a large variety of functionalities. Wang’s device was able to follow lines, recognise objects and stop signs [9]. The only real weakness of this was its inability to do on-board computation.

These projects both used very large-scale NN to do the computation efficiently; thus, both were required to use an external server to do the computation. This was a problem as the device can only be used indoors with a suitable connection to the server. Despite the large-scale network, it could only be implemented on a small-scale track to ensure there was always an adequate connection to the server.

3. Computational foundations

3.1 Machine learning

Machine learning is one of the main forms of artificial intelligence; it is an area concerned with how machines can learn to recognise patterns in data and thus enable them to predict future outcomes based on previous patterns. It is a branch of artificial intelligence which often interlaces with a wide variety of mathematic functions and pattern recognition techniques [10]. Some methods of machine learning include decision trees, K-nearest neighbours (KNN) and the most famous ANNs.

3.2 Artificial neural networks

An artificial neural network is a system that is designed to replicate the connections from the synapses within the human brain and their associated learning processes. The human brain consists of many neurons (cells) which all can process information independently. Within these cells are three main parts, the cell body, which contains the nucleus, the "dendrites" which receive information and a long axon connected to other cell's dendrites for outputting information. Information passed between neurons in the brain through the dendrites is in a form similar to that of an electrical signal. When the signal level is of a required value (which will be set within the individual nucleus), the neuron will activate, and information will be sent along its axon to other neurons [11]. This can be applied to a piece of written software to replicate this process for any modelling and problem-solving purposes.

Another similarity that can be achieved through an ANN is processing information via interconnected nodes using simple signals, each link between nodes has its own numerical "weight" when coded, and this is the primary means of learning, which is achieved by altering these "weight values" towards an optimum value to gain a connection with a high success rate.

ANNs are code-based representation of a biological neural network; they are incredibly adaptable and have a variety of different structures/training techniques which can be used to adapt the network based on the most efficient design to solve the problem in hand. NN applications are mainly used for data mining purposes; however, they are also adaptable for use for many other computational problems. One example of this is using a particular model of NN, renown as multilayer perceptron (MLP), for image recognition to note and recognise the difference between things such as handwriting styles, animal types, facial recognition and many others.

3.3 Learning process in ANN

There are two types of learning strategies available to the training of neural networks:

- Supervised learning: within this the concept of a "teacher" is present, and its role is to compare the network output with the correct output and make "adjustments" if necessary. One of the most well-known examples is

"backpropagation" algorithm, which is a gradient-based approach minimising the error level between the output and the desired results [11]. This is highly effective when your training data is of a high quality and well labelled. However, it runs risks in terms of performance should you have any anomalies within your training set that you are unaware of.

- Unsupervised learning: within this there is no concept of a "teacher", and thus the network learns by examining the input data over multiple iterations until any inherent properties are discovered [12]. This is much more effective over a longer period of time and can be very efficient when done correctly.

The learning process of ANNs is also known as the process of "training", where a set of test values or a "training set" is passed through the ANN and its output values are compared to those which are expected and then the values are altered accordingly for the next iterative trial. To compare this to the human NN, the ANN's nodes are trained to fire in response to a particular input of bits/pattern. Each input will have its own value or according weight, and this will be calculated by multiplying the value by its assigned weight. If the sum of all these inputs equals a greater value than the activation threshold, the neuron will activate; this is the same as in ANN's. Once these values are accurate for many cases, should the same pattern of input be placed into the ANN, its associated output will be given. This is a typical process of machine learning. If in non-training mode, the ANN will process the bits as normal and usually and if training was sufficient, the NN should still produce the correct output in the end.

For this project, supervised learning was used as we had an abundance of very high-level training data to work with, all of which was already labelled.

3.4 Multilayer perceptrons

Multilayer perceptrons (also known as backpropagation neural network) are a type of neural network. As seen in **Figure 3**, they consist of a feedforward network structure including one or more hidden layers and a non-linear activation function (e.g. sigmoid) and utilise a supervised learning approach to training using a method called "backpropagation". MLPs are the most commonly used form of ANNs since they have a very flexible form—meaning they are adaptable to many different circumstances and problems [13].

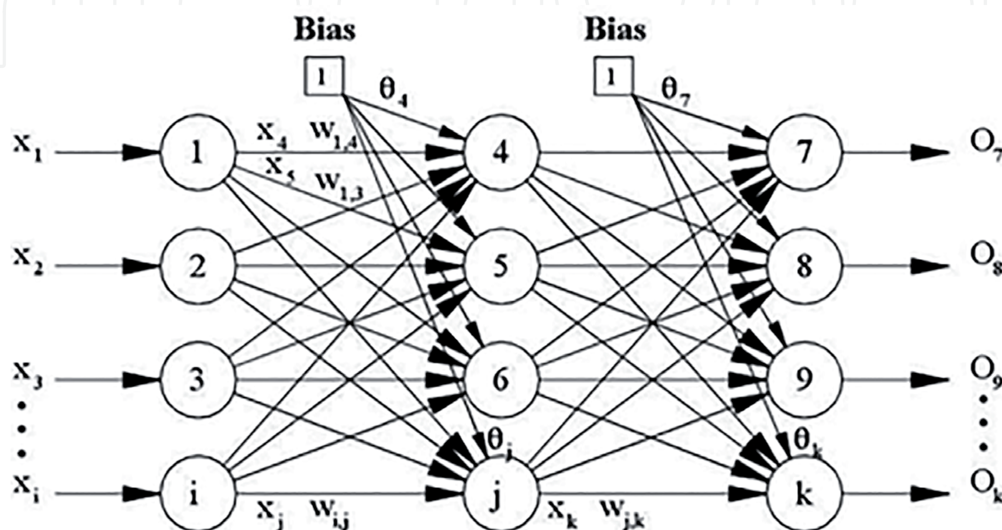

Figure 3.
Backpropagation neural network with one hidden layer [14].

3.5 Feedforward structure

The feedforward structure was the first released structure for a neural networks and is also the simplest form. It originates with an X amount of input nodes, in which each receives one value. The number of input nodes is always equal to the size of one individual element in the data set. For example, when using an MLP for image processing, if the image is 20 × 20 pixels, there would need to be 400 input neurons to account for each pixel. This helps to determine the size of the network, and because of this all data being fed into the network must be of the same size.

As these values go through the network in the same method as in a NN, an input pattern from the data set is passed through the input layer, and each value proceeds to propagate through the layers of the network with the weights being applied to the value until the output values are produced. Typically, the output values are a set of probabilities for each output option to be true.

When this process is occurring during the iterations through the training set, the set of output values is compared to the expected output set, and a set of error values is calculated for how much of an error each node contributes to the overall error. This error calculation occurs across all layers in the network due to the simple fact that inevitably all layers and their respective nodes contribute a small amount to the overall error. To combat this the error calculation occurs through each layer, and each input value to determine the amount connection has contributed to the overall error. This is then used in the backpropagation function to alter the weights accordingly.

3.6 Training the model

Training the model is the most crucial part of the process. Firstly, the weighting values are all initialized with random numbers between −1 and +1. Then the training set or "training data" is imported.

Next begins the recursive process which, each time, presents one piece of training data to the network, which propagates through the network, and an output is produced. This output is then compared to the labelled output of the training data. Following this comparison, the training algorithm propagates backwards through the network, known as "backpropagation", and alters the weighting values accordingly. This is repeated a finite amount of times specified by the person running the training model, until it is completed and the values are stable.

Backpropagation algorithm is a supervised learning technique which is applied to train neural networks. It works by propagating the error backwards through the network and internally altering the weight values between each node to try to improve the quality of output by minimising the error in the next runs. This is done in the "training" phase where the network is shown a "training set" which consists of a series of data and the correct output per data item is attached. This is then fed through the network, and backpropagation alters the weights to make it work for all data types and their corresponding outputs.

Backpropagation works by receiving the error value for each node, as calculated in the feedforward propagation, and utilises these error values while propagating through the network to adjust the node weightings positively or negatively in accordance with the error values per node.

This is explained in more detail [11] with "Once the error signal for each node has been determined, the errors are then used by the nodes to update the values for each connection weight until the network converges to a state that allows all the training patterns to be encoded. The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the

delta rule or gradient descent. The weights that minimise the error function are then considered to be a solution to the learning problem" [11].

3.6.1 Training algorithm

The standard pseudocode for training a neural network using backpropagation method which can be adapted to any language is as follows (**Figure 4**):

3.6.2 Over-training and under-training

While training the network to understand data, it is very important not to over-train or under-train the network, as both have problems associated.

Under-training can occur when the network does not have enough hidden layers/nodes within these layers to accurately represent the complexities of the problem accurately. The result of this is that the network is not of a sufficient size to recognise patterns effectively and will consequently under-fit the data pattern.

Over-training can be caused by a network that is overly complex—meaning that it follows the major pattern exactly and when confronted with other data, may not produce results that are within the average range of the training data. Networks with too many hidden nodes will tend to over-fit the data pattern [16].

This can be combatted by designing an adequate network structure prior to training and ensuring the rate/iterations of backpropagation are not too low or too high, thus creating a network that creates good solutions to new data problems.

```
Assign all network inputs and output
Initialize all weights with small random numbers, typically between -1 and 1
repeat
  for every pattern in the training set
    Present the pattern to the network
//      Propagated the input forward through the network:
   for each layer in the network
     for every node in the layer
       1. Calculate the weight sum of the inputs to the node
       2. Add the threshold to the sum
       3. Calculate the activation for the node
     end
   end
//      Propagate the errors backward through the network
   for every node in the output layer
     calculate the error signal
   end
      for all hidden layers
     for every node in the layer
       1. Calculate the node's signal error
       2. Update each node's weight in the network
     end
   end
//      Calculate Global Error
   Calculate the Error Function
  end
while ((maximum  number of iterations < than specified) AND
  (Error Function is > than specified))
```

Figure 4.
Training a neural network using a backpropagation algorithm [15].

3.6.3 Other machine learning algorithms

Machine learning is a vast field of artificial intelligence, which includes many learning and modelling approaches. Another algorithm which could be an alternative solution to this problem is to implement the K-nearest-neighbour search algorithm, which is one of the most known unsupervised learning algorithms used for data clustering and classification. This is explained briefly in [17] as "Nearest neighbour search is an optimization technique for finding closest points in metric spaces. Specifically, given a set of n reference points R and query point q, both in the same metric space V" [17].

This would allow a training set similar to that of a neural network; however, here, they are all stored alongside their relevant move instruction. In larger data sets, this could be very beneficial, as there would be a large training set to compare to; however, with such a simple task as this, autonomous car application, it could be considered "overkill" as the potential for undertraining is too high. Furthermore, in safety-critical situations such as a car driving, a "best guess" kind of algorithm may not return the best results; should an anomaly arise, it could have serious repercussions.

3.6.4 Libraries for machine learning

There are two commonly used C/C++ libraries which support machine learning, FANN and OpenCV. The decision was made to use OpenCV due to the fact that it is much more well-documented and has a tutorial part of their documentation which is considered to be very helpful. Furthermore, neither FANN's documentation nor the library itself has been modified since 2007, which deems it potentially outdated for newer versions of C++ [18].

3.6.5 Compelling performance constraints

The developed system carries a variety of performance constraints. The Raspberry Pi does not have high processing capabilities, and thus the network needed to be efficient enough for the device to get at least five MLP runs per second in order to run smoothly. The amount of time between calculations will directly affect the quality of the cars' driving ability. This study is however merely a proof of concept, and thus a high processing power is not necessary to prove the concept works.

A further performance constraint would be in relation to the quality of the driving of the vehicle, for example, it could be very jittery when driving or be quite smooth—this is all proportionate to the amount of computations the neural network can do per second. If the computation power of the Pi means that this is quite slow, naturally, the vehicle will not be 100% accurate in staying between the lines, whereas if it can be developed to work efficiently, this may be less of a problem and the vehicle will drive more "smoothly".

A solution's quality could be measured in terms of the number of errors/stops the vehicle makes in a given time frame, and the lower this value is, the more efficient the device is.

3.7 Hardware design and implementation

Below is the design of the hardware side of the system, excluding the external camera; we used two motors to control drive and steering and two HC-SR04 ultrasonic distance sensors to ensure for obstacle detection (**Figure 5**).

DOI: http://dx.doi.org/10.5772/intechopen.88342

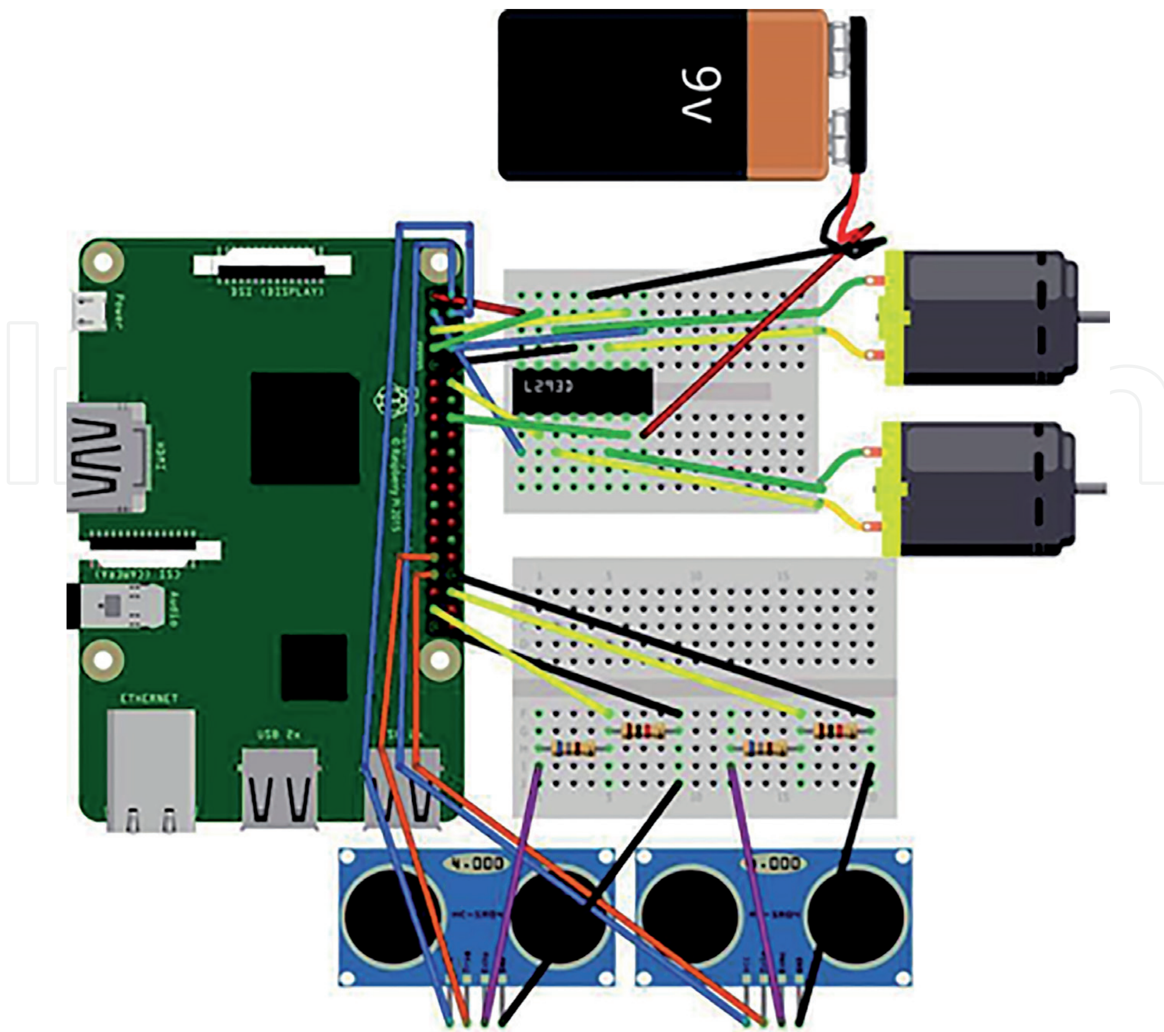

Figure 5.
System design architecture of hardware.

3.7.1 Hardware circuit design

Implementation of the hardware can be a challenge without the correct equipment. Following the initial designs, in the prototyping stage, a breadboard was used (see above) to manually switch wires and avoid soldering until the circuit was fully working. However, breadboards are not a long-term solution, and the wires will begin to fall out of the sockets, so at the point that a working circuit is created, it should be noted down in a form such as this (**Figure 6**).

This then allows the creation of a long-term solution: a custom printed circuit board (PCB) soldering is required to do this, but it is much more secure and thus more reliable, especially in projects which experience a high level of movement. PCB designs usually represent the following (**Figure 7**).

3.8 Track design

A more modular layout is required for a project of this sort, because it is designed to be totally mobile and adaptable to different circumstances. Consequently, a Scalextric-type (modular) approach with around 6 corners and 10 straight pieces would be a good idea.

The turning circle of the car required a large space, and the size of an A2 paper was chosen to make a modular-type design of which various sheets could contain different directions which led to the following two possibilities:

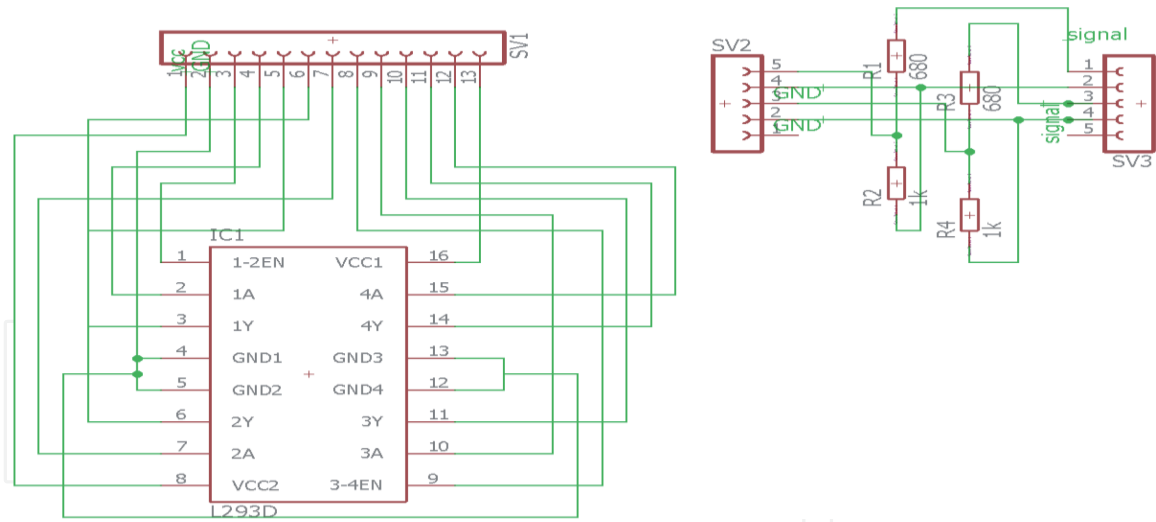

Figure 6.
Hardware circuit, sensors and motor controller.

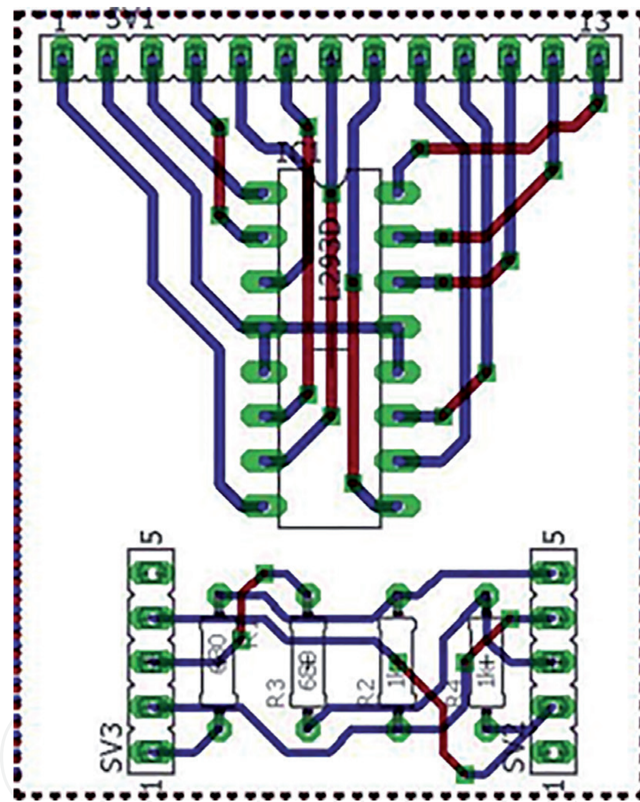

Figure 7.
Final PCB design of the hardware.

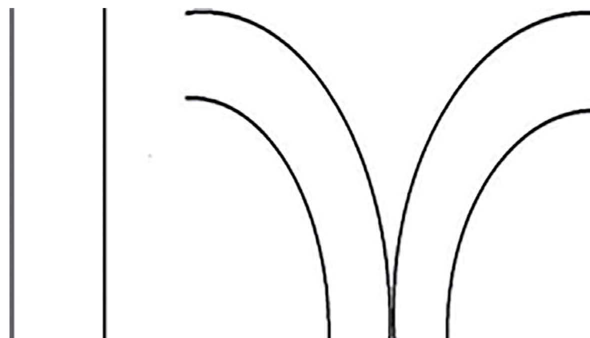

Figure 8.
White background with black lines.

Figure 9.
Black background with white lines.

Due to the nature of the testing environment/demonstration environment—a floor—a decision was made to determine the track would have to blend into the background on which the track was being set, in case of the unlikely event when the track went off course. It was viewed, while considering the "canny edge detection" in **Figure 8**, that the white background may interfere with the surrounding environment and the image preprocessing would essentially and completely alter the image, so it is viewed that the black background would be more suitable for the environment the vehicle will be in (**Figure 9**).

4. Implementation

4.1 Machine learning

4.1.1 Neural network topology

The following decisions were made:

1. The images are scaled down from 1024x720 to 10x10 to add efficiency to the network as the larger the image, the larger is the network; 100 bits was an efficient base level for the number of nodes in the input layer.

2. The neural network should have five layers, to account for a steady rate of drop off in the number of nodes per layer, inevitably aiming to end up with three outputs.

3. Given the objective was to achieve 5 FPS while the vehicle was driving and the length of the test track was around 25 m, we determined at a steady speed of 2.5 km/h; the vehicle would undergo around 300 frames per iteration. This was then used to calculate that at a high level, while taking into account the possibility of overtraining and undertraining, the training set should be around 50x the size of the data it would be handling, taking into account duplicate images around corners and other variables, so a training set of ~15,000 images were collected manually and labelled accordingly.

The MLP topology for this project is as follows (**Figure 10**):

The first layer is the input layer—this has 100 nodes to represent the 100 values in the one-dimensional array the live image is converted to. Each one of these nodes represents 1 pixel in the 10x10 image.

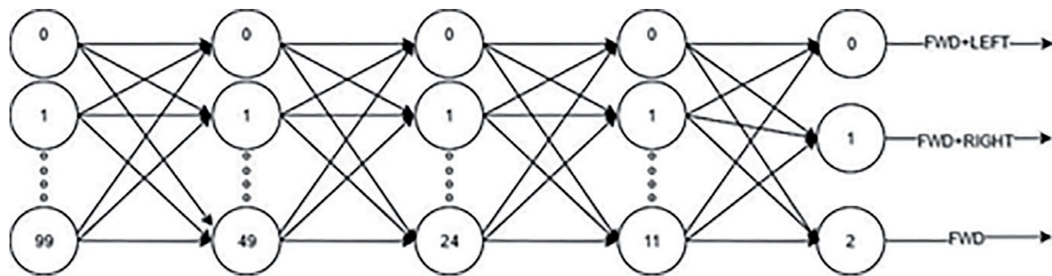

Figure 10.
System topology.

The second, third and fourth layers have 50, 25 and 12 nodes, respectively; these are known as hidden layers, and these sizes are calculated automatically.

The fifth layer which happens to be the output layer has three nodes, which all correspond to the steering control instructions:

- Output 0: FORWARDS | LEFT
- Output 1: FORWARDS | RIGHT
- Output 2: FORWARD ONLY

4.1.2 Data collection

The training images are taken by a small script on the Pi which allows a photo to be taken every X seconds. This was set to 0.8. The image resizing and image filters were added and then implemented; while this was running, the car was moved round the track allowing for multiple angles and directions. All of these photos were then stored on the Pi.

From that set of photos, manual collation was required to store each picture in its corresponding class folder, in this case, one of 0, 1, or 2.

Once these images had been correctly stored, another part of the program could read them from their containing class folder and compute the following to store them in an XML file named Train_data.xml.

This data is all then stored in Training_data.xml. The same process occurs again for generating/gathering of training data.

This process occurs to convert the real-time image into a format that is suitable for a neural network's input Layer (**Figure 11**).

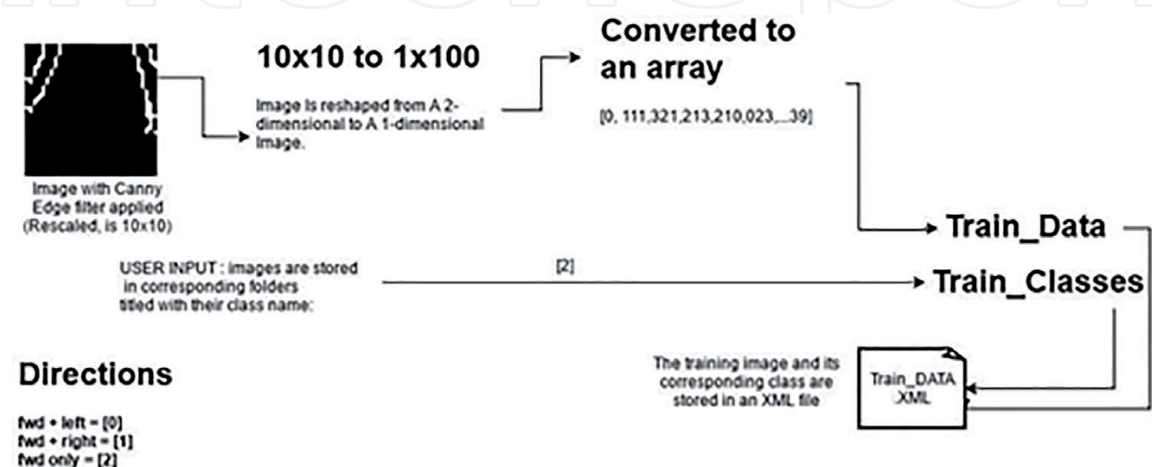

Figure 11.
The process of generating training and test data sets.

The image is filtered using the image processing techniques (black and white, Gaussian blur, canny edge detection) in order to make the image as simple as possible and have only the lines as a defining feature; this means there is less chance of errors in recognising and classifying the image via the NN.

The images are scaled down to 10x10 in order to make an efficient and enough neural network; the larger the image, the larger the size of the neural network will need to be, which is directly proportionate to the efficiency/iterations per second of the image classifying loop that the device can perform. Thus, a smaller image resolution was chosen in order to make the device as efficient as possible.

Firstly, the filtered image is retrieved from the live stream from the camera and reshaped row by row to create a 1x100 image and converted to a one-dimensional array of 100 values ranging from 0 to 255 to represent each pixel of the image by each particular element of the array. The one-dimensional array is then directly fed into the neural network as the correct input format. Provided this process occurs on each data item, it can then be fed into the network for training, testing and prediction purposes.

4.2 Training the network

Training the network is done via the Train() function within OpenCV. Prior to calling this, the associated parameters have to be set. The process of doing this is relatively simple due to the helpful documentation provided in the OpenCV source codes and thus drastically reduces the amount of code required to show this.

Firstly, all values from training data set need to be loaded and added to the stack in their respective math formats (**Figure 12**).

Following this, the network needs to be defined and its topology declared (**Figure 13**).

Next, the relevant network parameters need to be set.

Firstly, the activation function must be set, due to the fact that this network is a Multilayer-perceptron, the chosen activation function is sigmoid (**Figure 14**).

```
fsa.open("TRAIN_VALUES.xml", cv::FileStorage::READ);
    cv::Mat train_data, train_labels;
    fsa["TrainingData"] >> train_data;
    fsa["classes"] >> train_labels;
```

Figure 12.
Loading the data from XML file.

```
int nfeatures = train_data.cols;
Ptr<ml::ANN_MLP> ann = ml::ANN_MLP::create();//define MLP
   Mat_<int> layers(5,1);
   layers(0) = nfeatures;    // input 100
   layers(1) = 50;       //nclasses * 16 hidden
   layers(2) = 25;       //nclasses * 8  hidden
   layers(3) = 12;       //nclasses * 4  hidden
   layers(4) = nclasses;    // output, 1 pin per class = 3.
   ann->setLayerSizes(layers);
```

Figure 13.
Declaring the network and assigning its topology.

```
ann->setActivationFunction(ml::ANN_MLP::SIGMOID_SYM,0,0);
```

Figure 14.
Set the activation function to sigmoid.

Next, setting the training method and training criteria, a backpropagation rate of 0.0001 for 50,000 iterations was chosen, the higher level of iterations will make the weighting more secure and could be potentially viewed as "over-training"; however, in the given circumstance of a real-time application, the network has to be as accurate as possible. The rate of backpropagation ratio being set to 0.0001 is the standard rate which is used by the majority of users (**Figure 15**).

Following this, all that is required is to call the TRAIN function from the CV library, to run the training algorithm with the above specified topology, parameters and data (**Figure 16**).

4.3 Prediction

Prediction is used in the testing process and the self-driving process. Prediction is the processing of unknown data in order to categorise the class of the image it is being given.

In the case of testing, this is used to compare the network's output to the correct output in order to calculate the accuracy of the network.

In the self-driving process, the unknown data (camera image and sensor date) is fed into the neural network via prediction to determine the classification of the image. The value returned is used to determine the current state of the track in front of the vehicle, and this is the direction the car should move in.

In the real-time processing part of the project, the real-time image has the aforementioned image preprocessing functions applied to it, to transform it from a fully coloured image into a one-dimensional array of binary values, to match the input layer of the MLP.

Prediction works by receiving this correctly formatted array and passing it to the input layer. This is then fed through the network (which has been fully loaded from the NNVALUES.XML file).

Following this, the network will produce an array of float values, representing the probability of the image belonging to that class. So, in this case it will produce an array of three float values, representing the probability the image which belongs to each of those classified outputs (0, 1, 2).

The highest value in this array is recorded, and its location in the array (0, 1, 2) defines which class the image is most likely to belong to.

This highest value location is then fed into a switch statement which implements a move instruction based on the value.

4.4 Autonomous driving

Prior to the calling of the AutoDrive function, the following criteria must be met in order for it to work fully:

All hardware devices must be fully installed and working.

Training and testing data must be created and stored.

```
ann->setTrainMethod(ml::ANN_MLP::BACKPROP, 0.0001);
ann->setTermCriteria(TermCriteria(
TermCriteria::MAX_ITER+TermCriteria::EPS, 50000, 0.0001));
```

Figure 15.
Set training method and training criteria.

```
ann->train(train_data, ml::ROW_SAMPLE, train_classes);
```

Figure 16.
Train model.

Training must be called to train the network and save it once trained.

Testing of the network must prove a high accuracy level of the network.

The track must be ready to use.

All of this is however done by the administrator to initialise the system prior to demonstration.

Following this, the device can be placed onto the track and will drive around it, utilising object detection and NN prediction in order to make informed decisions on which directions to drive.

It can begin the self-driving Loop in which it will continue driving until the process is halted. This loop will continue to iterate, driving the car in the desired direction based on the current image of the track in front of it, which will change each iteration.

5. Testing

5.1 Unit testing

Unit testing is the testing for the initial components and all basic instructions in the system. These are developed from the basic implementation of the system (**Table 1**).

5.2 Integration testing

Integration testing is testing to see if the more advanced functionalities of the system are comprised from the design process (**Table 2**).

Test	Test criteria	Where	Qualifier	Pass
1	Does front motor turn right?	TestMotors()	Visual	Yes
2	Does front motor turn left?	TestMotors()	Visual	Yes
3	Does rear motor go forward?	TestMotors()	Visual	Yes
4	Does rear motor go forwards and front motor go left?	TestMotors()	Visual	Yes
5	Does rear motor go forwards and front motor go right?	TestMotors()	Visual	Yes
6	Does sensor A function halt when object is within 15 cm?	TestSensors()	Visual	Yes
7	Does sensor B function halt when object is within 15 cm?	TestSensors()	Visual	Yes
8	Sensor A distance of 15 cm shown when object is 15 cm away	DistanceSenseA()	Visual	Yes
9	Sensor A distance of 50 cm shown when object is 50 cm away	DistanceSenseA()	Visual	Yes
10	Sensor B distance of 15 cm shown when object is 15 cm away	DistanceSenseB()	Visual	Yes
11	Sensor B distance of 50 cm shown when object is 50 cm away	DistanceSenseB()	Visual	Yes

Table 1.
Unit testing and results.

Test	Test criteria	Where	Qualifier	Pass
12	Does canny edge filter apply to image stream from camera?	TestCamera()	Visual	Yes
13	Does camera image become scaled down to 10x10 resolution?	TestCamera()	Visual	Yes
14	Can image stream from camera be converted to black and white?	TestCamera()	Visual	Yes
15	Can image stream from camera be saved/loaded?	TestCamera()	Visual	Yes
16	Can image stream from camera be reshaped from 10x10 to 1x100?	TestCamera()	Visual	Yes
17	Can image be converted to a 1D array of values [0–255]?	TestCamera()	Visual	Yes
18	Can a saved image be loaded?	AutoDrive()	Visual	Yes
19	Can a set of images and their classes (read from folder name) be stored as a training data set?	ReadScanStore()	Visual	Yes
20	Can training data be stored in an XML file?	ReadScanStore()	Visual	Yes
21	Can a MLP be defined and created based on its topology?	TrainNetwork()	Visual	Yes
22	Can a trained MLP be stored once trained to an XML file?	TrainNetwork()	Visual	Yes
23	Can a trained MLP be loaded from an XML file?	TestNetwork()	Visual	Yes
24	Can a training data set be used to train an MLP?	TrainNetwork()	Visual	Yes
25	Is backpropagation the algorithm used to train the MLP?	TrainNetwork()	Visual	Yes
26	Does test data allow user to define accuracy of a trained MLP?	TestNetwork()	Visual	Yes
27	Does OpenCV's backpropagation algorithm work?	TestNetwork()	Visual	Yes
28	Are weight values stored of trained MLP not all the same?	TestNetwork()	Visual	Yes
29	Does NN/MLP allow prediction with current image?	AutoDrive()	Visual	Yes
30	Calculate most likely output via neural network (prediction)	AutoDrive()	Visual	Yes

Table 2.
Integration testing and results.

5.3 System testing

The system testing is the tests of the overall system. These are defined by the research section to determine what is needed from the system for the user (**Table 3**).

5.4 Acceptance tests: qualitative results of self-driving

These tests are used to qualitatively test the self-driving capabilities of the car and measure how smoothly it runs.

These are measured visually during demonstration and thus are only considered to be an opinion as opposed to a solid pass/fail scheme (**Table 4**).

Test	Test criteria	Where	Qualifier	Pass
31	User menu option: testing for camera	SysMenu()	Visual	Yes
32	User menu option: testing for MLP accuracy	SysMenu()	Visual	Yes
33	Is user menu easy to use?	SysMenu()	Visual	Yes
34	User menu option: testing for motors	SysMenu()	Visual	Yes
35	User menu option: testing for ultrasonic sensors	SysMenu()	Visual	Yes
36	Does the vehicle stay still if there is an object within 15 cm of the front of it?	AutoDrive()	Visual	Yes
37	Does the vehicle stay between the two lines needed to drive?	AutoDrive()	Visual	Yes
38	Does the vehicle successfully drive one loop around the track?	AutoDrive()	Visual	Yes

Table 3.
System testing and results.

Test	Test criteria	Where	Qualifier	Score
41	Score out of 10 for smoothness of drive when following a straight line	AutoDrive()	Visual opinion	9
42	Score out of 10 for smoothness of drive when going around a corner	AutoDrive()	Visual opinion	6
43	Score out of 10 for smoothness of driving when following an entire track	AutoDrive()	Visual opinion	7

Table 4.
Acceptance test and results.

6. Evaluation

6.1 Performance summary and achievements

Overall the project has achieved what it was required to do; it is fully able to drive around the track to a suitable level of accuracy. The chosen libraries were suitable for the project and provided well-documented functions regarding all aspects required to enable the vehicle to do the tasks that it had to perform.

In my opinion, the best feature of the vehicle is the custom-made circuit board, which required a great deal of time and effort to design and build. Once this occurred, everything from the operational hardware point of view went very smoothly.

The system passed all of the required tests. However, there was not enough time to implement some of the “would like to” aspects of the MoSCoW requirement analysis, but as discussed below, there is potential for further development.

6.2 Reflection

The main aims were to verify:

Can the vehicle recognise a track via NN and camera data?

Can the vehicle follow the track?

Can the vehicle control the car motors?

Can the vehicle utilise collision avoidance via ultrasonic sensors?

Can the vehicle recognise and stop at stop signs for a certain amount of time?

Following the test phase, the vehicle has shown the capabilities required to pass the first four out of five aims.

The vehicle fully utilises the neural network to accurately classify the real-time image to decide upon and then implement a movement instruction. This allows the vehicle to follow the track given any set layout and follow it until the procedure is manually stopped. A variety of obstacles have been placed in front of the vehicle during operation, and when this occurs, the ultrasonic sensors recognise it, and the vehicle stops within the specified distance in order to avoid collision. The vehicle remains stationary until the obstacle moves away or is removed from its path. If the obstacle is moved around incrementally, it will "follow" the object staying at a minimum of the specified distance away. This is a very useful functionality for potential additions to the project which is to have the vehicle following other moving vehicles or staying a specified distance away from the obstacle should it be moving.

However, the stop sign detection capability was not implemented due to time constraints. To do this a Haar cascade qualifier must be used; this is an entirely different machine learning algorithm and thus would have to be implemented alongside the neural network. This naturally would have added many further functionalities. This was placed into the "would like to have this requirement later to develop the system design further" section of the MoSCoW analysis of the requirements because not only would this further implementation be time-consuming but finding small-scale stop signs that would be adequate for this task proved to be challenging and thus was not achieved in time.

6.3 Relation to MoSCoW and further improvements

In relation to the MoSCoW analysis, "all of the must have", "should have" and "could have" related requirements were fully met to a good standard.

Further developments may be made as specified to cover the "would like to" aspect of the requirements.

Implementing a backpropagation algorithm manually as opposed to relying on the one provided from OpenCV, this would be desirable as it has the potential to be developed in a manner that may be more efficient for the problem at hand than just a generic algorithm.

A possible improvement would also be integrating the capability of designing, recognising and acting upon stop signs via a Haar cascade qualifier. This would be a further development to the device which could allow it to have more functionality and thus be more applicable to real-world projects.

Another adaptation which would have been interesting to implement would be using a K-nearest neighbour algorithm for the machine learning part of the project. This could be used to compare the results from this to that of the neural network, to see which is more accurate. However, as aforementioned this was not implemented currently because of the much larger data set that would be required to train it.

The only final improvement which could be made to the system would be to improve the mathematical functions applied to the ultrasonic distance sensors, which will be discussed below. These are currently only accurate at reading distances up to 50 cm using a simple mathematic function which for the present problem is acceptable, but future adaptations to the device may require more accuracy and thus more advanced mathematics to allow it to be accurate up to 5 m.

6.4 Obstacles faced

6.4.1 Hardware

Hardware design encountered a variety of problems with the circuitry of the vehicle which connected the external components to the Pi. Using a breadboard to connect all the components was not viable in the long term, and testing proved that as the car rattled, wires would come free as they were not fixed in place.

This was fixed by having a permanent version of the circuit made and soldered in the workshops at the university, producing a finished product seen in **Figure** 7. This took a while to be perfect, and because of this, other tasks had to be put on hold pending the new board design. The new board completely solved the problem and provided both stability and much greater longevity to the product.

6.4.2 Software

The software design process also had problems along the way which required solving.

The ultrasonic sensors in front of the vehicle required advanced mathematical formulae to be accurate at all distances, which would take time to complete, whereas using much a simpler formula would only mean they were accurate up to 50 cm as opposed to 5 m. This in the end was chosen as the vehicle only needs to be able to stop within 50 cm of an object, and anything beyond that distance cannot be considered an object in the vehicle's path. A useful add-on to develop the device further would be to utilise this functionality correctly for all distances. However, for the purpose of the current project, it was not viewed as critical.

6.4.3 Track design

Overall, the track design was a good choice. The black background stops the edges of the paper from being seen as different to the colour of the dark floor, meaning only the edges of the track (white lines) are seen as fully qualified edges by the edge detector. The only issue with the track is that the corners can be viewed by some as "too tight" and thus the vehicle sometimes struggles to stay completely between the lines while turning through them. However, this is not a major issue as overall the vehicle remains between the lines for the majority of the driving cases, and if it does go out of the lines, it will self-correct.

6.4.4 Image recognition and obstacle detection

Obstacle detection works perfectly, allowing the vehicle to stop moving if anything is within 15 cm in front of it. This is exactly what was stated in the system requirements.

Image recognition correctly classifies the images based on the direction they are qualified under, with a score of 114/114 on the test data.

7. Conclusion

Overall, I feel that this was a successful project, in that it demonstrates a clear proof of concept that the computations required for autonomous cars do not have to be performed externally but may be done within the vehicle itself. The effect of

this will be to make it much more versatile and adaptable for different environments and requirements. Another benefit of increased capacity and functionality within the vehicle would be to make it less vulnerable to external access, such as hacking, which could have implications for the vehicle, its user and others.

The limitations imposed by the scale of the vehicle used in this work will affect the physical space available to house the computing hardware to that which will fit within the vehicle. This will have consequences for the functionalities the car is able to effectively demonstrate when using such small-scale computational devices.

Acknowledgements

I would like to thank my supervisor Dr. Mehmet Aydin, professor of enterprise system module, for being available throughout the entire process of this project and having an in-depth knowledge of all the software techniques and development techniques required to complete this task.

I would also like to thank Dr. Larry Bull, professor of artificial intelligence, for his help in the general structure of the neural network and advice regarding the training data.

Notes

This project was chosen because it is considered to be very complex and the use of neural networks to process images is a study that is always improving. Furthermore, this project was chosen as a proof of concept for autonomous vehicles, which are constantly in development for future real-world applications. This means it is an area where this theory may be beneficial for further research by others.

My intention was to improve this by scaling down the size of the neural network, making the device more portable and thus more realistic. This can be done by allowing the computations to be done on the Pi itself, making the device more mobile. It was concluded that this could potentially come at the cost of performance as trying to do computations of a large-scale neural network on a Raspberry Pi will be near impossible. As a result of this, the network needed to be scaled down, meaning much smaller images being passed, and the FPS rate will also need to be reduced to around 5–10 FPS. This is sufficient, considering the project is only a proof of concept.

Author details

Toby White
University of the West of England, ARTIMUS Solutions LTD, Bristol, United Kingdom

*Address all correspondence to: toby@artimus-uk.com

© 2019 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

References

[1] Zimine T. Drive a Lamborghini with your Keyboard. 2013. Available from: http://thelivingpearl.com/2013/01/04/drive-a-lamborghini-with-your-keyboard/ [Accessed: 22 February 2019]

[2] Bell G. Software Adafruit's Raspberry Pi Lesson 9. Controlling a DC Motor|Adafruit Learning System. 2015. Available from: https://learn.adafruit.com/adafruit-raspberry-pi-lesson-9-controlling-a-dc-motor/software [Accessed: 21 February 2019]

[3] Wiring Pi. 2017. Available from: http://wiringpi.com/ [Accessed: 22 February 2019]

[4] Henderson G. WiringPi. 2012. Available from: https://projects.drogon.net/raspberry-pi/wiringpi/ [Accessed: 22 February 2019]

[5] Phillips. Phillips/libv4l. 2017. Available from: https://github.com/philips/libv4l [Accessed: 22 February 2019]

[6] Rosebrock A. Zero-parameter, Automatic Canny Edge Detection with Python and OpenCV—PyImageSearch. 2015. Available from: http://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/ [Accessed: 22 February 2019]

[7] Galkin I. Crash Introduction to Artificial Neural Networks. 2017. Available from: http://ulcar.uml.edu/~iag/CS/Intro-to-ANN.html [Accessed: 22 February 2019]

[8] Platis D. The World's first Android Autonomous Vehicle. 2016. Available from: https://platis.solutions/blog/2015/06/29/worlds-first-android-autonomous-vehicle/ [Accessed: 22 February 2019]

[9] Wang Z. Self Driving RC Car. 2015. Available from: https://zhengludwig.wordpress.com/projects/self-driving-rc-car/ [Accessed: 22 February 2019]

[10] ByteFish. Machine Learning with OpenCV2. 2017. Available from: https://www.bytefish.de/pdf/machinelearning.pdf [Accessed: 22 February 2019]

[11] Kim D, Papagelis A. Back Propagation Tutorial. 2006. Available from: http://www.cse.unsw.edu.au/~cs9417ml/MLP2/ [Accessed: 22 February 2019]

[12] Berwick B. Neural Networks II. 2014. Available from: http://web.mit.edu/6.034/wwwbob/recitation8-fall11.pdf [Accessed: 22 February 2019]

[13] Kendall G. G5AIAI: Neural Networks. 2017. Available from: http://www.cs.nott.ac.uk/~pszgxk/courses/g5aiai/006neuralnetworks/neural-networks.html [Accessed: 22 February 2019]

[14] Wei L. Neural Network Model for Distortion Buckling Behaviour of Cold-Formed Steel Compression Members. 1st ed. Helsinki: Helsinki University of Technology Laboratory of Steel Structures Publications

[15] Tveter D. Evaluating a Network. 1st ed. Canada: McGill University; 1995

[16] Barry R. Artificial Neural Network Prediction of Wavelet Sub-bands for Audio Compression. 2000. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.362&rep=rep1&type=pdf [Accessed: 22 February 2019]

[17] Mateo F, Sovilj D, Gadea R. Approximate k-NN delta test minimization method using genetic algorithms: Application to time series. 2010;**73**:10-12

[18] Rpi Forum. Raspberry Pi View Topic—C vs Python for GPIO. 2017. Available from: https://www.raspberrypi.org/forums/viewtopic.php?f=33&t=22640 [Accessed: 22 February 2019]

IntechOpen

IntechOpen