**A Self-Navigating Car working on Neural Networks**

**Introduction**

The project Self-Navigating Car using neural networks aims to demonstrate how a self-navigating car works. It works on the basis of neural networks and uses the back-propagation algorithm.

**Artificial neural networks (ANNs)**

Artificial neural networks or [connectionist](https://en.wikipedia.org/wiki/Connectionism" \o "Connectionism) systems are computing systems inspired by the [biological neural networks](https://en.wikipedia.org/wiki/Biological_neural_network" \o "Biological neural network)that constitute animal brains. Such systems learn (progressively improve performance) to do tasks by considering examples, generally without task-specific programming. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually [labeled](https://en.wikipedia.org/wiki/Labeled_data" \o "Labeled data) as "cat" or "no cat" and using the analytic results to identify cats in other images. They have found most use in applications difficult to express in a traditional computer algorithm using [rule-based programming](https://en.wikipedia.org/wiki/Rule-based_programming" \o "Rule-based programming).

**Back-propagation**

Back-propagation is a method used in [artificial neural networks](https://en.wikipedia.org/wiki/Artificial_neural_network" \o "Artificial neural network) to calculate the error contribution of each neuron after a batch of data (in image recognition, multiple images) is processed. This is used by an enveloping optimization algorithm to adjust the weight of each neuron, completing the learning process for that case.Technically it calculates the [gradient](https://en.wikipedia.org/wiki/Gradient" \o "Gradient) of the [loss function](https://en.wikipedia.org/wiki/Loss_function" \o "Loss function). It is commonly used in the [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent" \o "Gradient descent) optimization algorithm. It is also called backward propagation of errors, because the error is calculated at the output and distributed back through the network layers.

**Advantages**

This model can be used by visually challenged people who cannot drive. Since there is no need for a person to navigate the vehicle, human errors can be eliminated while driving. This car can also be useful in the case of long trips where generally a normal human being gets tired and his efficiency decreased. But since this is an A.I. it has been programmed to work at similar efficiency all the time.

**Limitations**

Thought the project has so many advantages it still poses few minute flaws. The first one being the general error in prediction. If there is any error in the prediction of the vehicle, it can lead to a hazard. Second is the time which it requires to predict the route may be longer than an actual human beings’. Another flaw which can be seen is that the hardware required for it to work may not be ideal.

**Existing Approaches**

The idea of having an automatic vehicle is not new and many ideas have come to build on it. .

One such example is the use of Lidar, the three-dimensional laser scanning technology used to create real-time maps of a car’s environment that are compared with pre drawn maps, does the job quite effectively. Though their high cost is a huge let-down.

A new software system developed at the University of Michigan uses video game technology on self-driving and automated cars. The technology enables them to navigate using a single video camera, delivering the same level of accuracy as laser scanners at a fraction of the cost. The major hurdle which this faced was to get a hardware that could do all the heavy lifting of the operations load.

Another one is the use of micro controllers and sensors for achieving the tasks. Radio Frequency Identification technology helps for making self-navigation, with the help of Radio Frequency Identification cards. On-board Radio Frequency Identification module is to locate Radio Frequency Identification tags embedded in the path in the form of a grid. IR sensors are placed into the car in different directions for sensing the obstacles. Alcohol detection module detects that the driver has consumed alcohol or not, if it detects that, then the car will not get turn on for driving. The issue here was the maintenance and accuracy of the sensors were not optimal.

WAYMO by Google says “Our fully self-driving technology will handle all the driving so you can go from door to door without taking the wheel. This will deliver the biggest impact on improving road safety and mobility for everyone.” Basically they are creating AI drivers for their self-navigating cars.

The reason why we think our process is much better than the above mentioned strategies is due the reason that we are only using the prediction data or the trained data of routes and then loading them into the software which will use it to predict the routes. Here the load of processing is done at a remote site and only the predictions are sent. Hence reducing the time and increasing the efficiency of our car. Plus since we are only using cameras for training the neural network their cost and maintenance is cheaper than other technologies.

**Methodology**

**Abstract:**

The ALVINN system captures video frames every couple of seconds and passes them to a (series of) neural networks which have been trained by watching a human drive in similar environments. The trained neural network can then be passed live video frames and will predict how to steer to stay on the road ahead! The needed was a system which could operate in two modes:

Record — the system captures video frames and the control input from a human driver and records them for later use to train the neural network.

Drive — Captures live video frames, passes them to a trained neural network which makes predictions about how to drive/steer which are sent to the car by radio control - hey presto, a self-driving car!

**System Design:**



The system consists of Android phone — mounted on the car, captures video frames of the road ahead using its built-in camera at ~15 fps. An app running on the phone connects to a server running on a laptop computer via Wi-Fi and streams 176x144 grayscale video frames across the connection.

Computer — runs a little Java app called "Driver" which acts as both a TCP server, receiving streamed image frames from the phone and a user interface allowing a human driver to control the car with the cursor keys or mouse. In record mode, the video frames are saved to disk, labelled with the current control input coming from the human driver. The neural network is trained using these labelled frames in a separate environment on the computer. Trained parameters are saved out to files which are in turn read by the Driver app... which in auto mode can feed incoming video frames directly to the neural network and steer according to its predictions, by sending instructions over a serial interface connected to an...

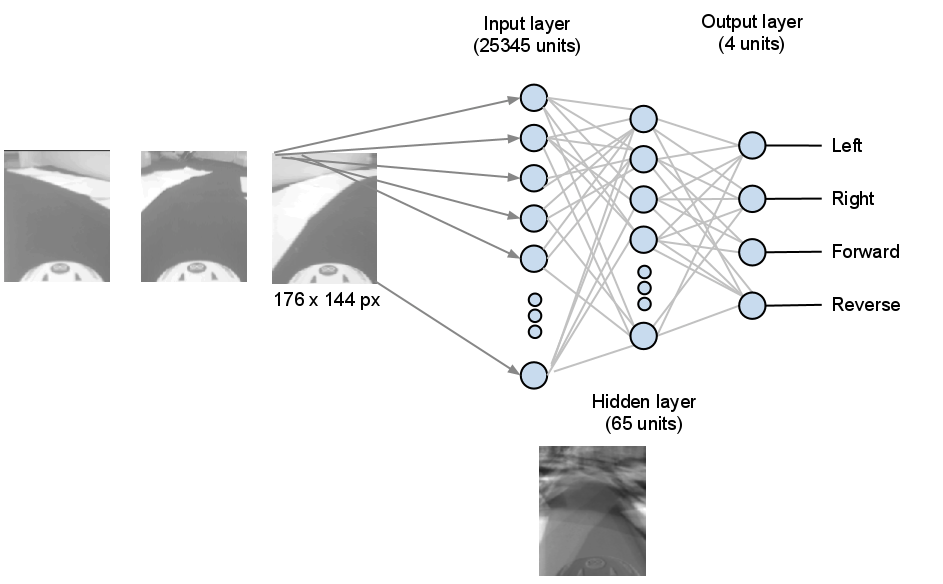
Arduino Uno — connected to the computer via USB and hacked to connect to and simulate keypresses on the car's radio controller PCB.

**The Neural Network**

As on the ML course, the Neural Network is trained using an octave program, and I didn't stray too far from the set up used there. The diagram below shows the architecture of the network I used. Here we have 25345 units in the input layer - 25344 units which are fed the brightness value of an individual pixel in the 176 x 144 video frame (176x144 was the lowest resolution the camera on my phone supported in preview mode) and a bias unit. I chose to use 64 non-bias units in my hidden layer - this choice is fairly arbitrary, but I found that my initial choice of 256 took a pretty long time to train, and 8 units were not expressive enough to drive the car successfully. Crucially, there are four units in the output layer - one corresponding to each of the instructions we can send the car - go forwards, backwards, left or right.

The network is trained using back-propagation which produces weights corresponding to the contribution each input layer unit makes to the activation of each of the hidden layer units, and the contribution each hidden layer unit makes to the activation of each of the output layer units. There's no explicit image processing going on here - the network literally figures out what kind of patterns in the input video frames are useful in making decisions about how to drive the car, based on minimizing the numerical error between the current prediction and all of the recorded examples. The little frame at the bottom of the diagram below is a visualization of the weights assigned to each of the pixels in the input layer as they contribute to just one of the hidden layer units – we may be able to see here that this unit corresponds to some kind of edge detection in the middle distance broadly sweeping to the left or right.

To make predictions in auto mode, we also implemented the same network topology in Java (making use of the Apache Commons Math Library for linear algebra). Neural Network.java contains the interesting code and is a generic neural net implementation you could use for any three layer network (and also contains code for parsing a Real Matrix from an octave .dat file). To test the correctness of this implementation, NeuralNetworkTest.java checks that the predictions from this code are virtually identical to those made with the same input data and network parameters under the octave implementation. The Driver app uses this Java implementation, set up with network parameters loaded from files written by the octave script at the end of the training process.

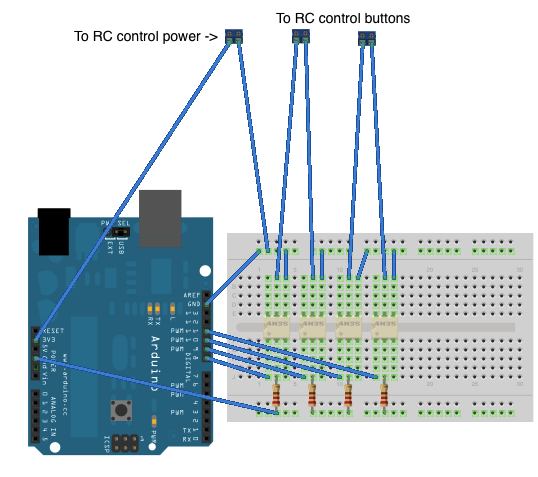


**Radio Control with Arduino**

We wanted the commands sent over the USB / Serial interface could be sent to the car via its original controller. So we bought the cheapest RC car we could find with the controller as a push button on/off type. Then our job was to hack into the controller.

A car like this has on/off switches for each of forward / backward / left and right. We followed the tracks from each side of each switch to the nearest solder joint on the original board. Found the pads for each switch and confirm with a multimeter that the solder joints are the correct ones - when the switch is pressed, the resistance between the two relevant joints will be zero. Once we had identified the joints that matter, we attached patch wires to each point with a soldering iron.

Finally we built out a breadboard that looked like this:



**Arduino sketch**

Finally, we needed a firmware sketch to run on the Arduino board.

if (Serial.available() > 0) {

incomingByte = Serial.read();

left = right = forward = back = LOW;

if (incomingByte & 0x01) {

left = HIGH;

}

if (incomingByte & 0x02) {

right = HIGH;

}

if (incomingByte & 0x04) {

forward = HIGH;

}

if (incomingByte & 0x08) {

back = HIGH;

}

...

digitalWrite(leftPin, left);

digitalWrite(rightPin, right);

...

}

Which reads a byte from the serial interface and decodes it to determine which buttons to push on the remote control which are written out as HIGH signals on the arduino output pins connected to the opt-isolators above.

Putting all these together we were able to make a working prototype.

**Results :**

\*Demo Graph not original values.

**Conclusion**

The self-driving car prototype’s prediction graph shows how the change in the neural networks capacity changes the predictions of the vehicle. Also the project achieves its goal of implementing a self-driving car with a remote training file using back propagation.