Simulated Visual Perception for Autonomous Agents

Postgraduate project for the Department of Computer Science at the University of Auckland, semester one, 2005
By Daniel Flower
Supervised by Dr Burkhard Wünsche and Assoc-Prof Hans Werner Guesgen

Abstract

As the number of robots in the world increases, from automatic vacuum cleaners, to toy robot dogs, to autonomous vehicles for the military, the need for effective algorithms to control these agents is becoming increasingly more important. Conventional path finding techniques have relied on having a representation of the world that could be analysed mathematically to find the best path. However, when an agent is placed into the real world in a place it has not seen before, conventional techniques fail and a fundamentally different approach to path finding is required. The agent must rely on its senses, such as the input from a mounted camera, using this information to get around. In this project, a virtual city is implemented, and agents must navigate their way around it by using only what they see from their point of view. By feeding what the agents see into a neural network, they are able to learn how to avoid obstacles, follow the road, and they also show promise in using this technique for path finding.

Table of contents

Introduction

The control of agents in an environment is a difficult problem that has been approached in a variety of ways, from discretising the world into cells to giving obstacles repellent forces to push agents away from them, among many other methods. But in some situations, such as a robot in the real world, the agent has no representation of the world and can only rely on its visual sensors. In these cases, traditional navigation and path finding techniques do not work.

The goal of this project is to investigate the use of neural networks to analyse the visual input of an agent in order to control its navigation. The use of neural networks was inspired by the difficulty in hand coding algorithms to control obstacle avoidance in an arbitrary environment, and the success of the ALVINN system (explained in [1]) which successfully controlled the steering of a vehicle on real roads by analysing the visual input.

This would be achieved by creating a virtual city that agents could navigate around, but importantly the agents would be given no information about the city other than what they saw. The goals of the project are to train an agent to follow the road in much the same way as the ALVINN system, train an agent to avoid obstacles, train an agent to obey traffic lights, and finally use the neural network for path finding.

This report does not explain the theory behind neural networks and it is assumed that the reader has a basic understanding of them.

It is hoped that this project will give an idea of the sort of functions that a neural network is suitable for when controlling an agent based on its visual perception, and which functions are not suitable.

Literature review

The key paper central to this project was "Neural Network Vision for Robot Driving" by Dean A. Pomerleau [1]. In this paper, the ALVINN (Autonomous Land Vehicle In a Neural Network) is described. ALVINN controlled the steering of a modified van while driving on public roads by analysing the image of the road ahead captured by a camera mounted at the front of the vehicle. The image was resampled to the small size of only 30 x 32 pixels, and each of the pixel values were input into a neural network. The neural network had only one hidden layer with four nodes, and the output layer had 30 nodes. Each of the output nodes was a linear representation of the steering amount, from hard left to hard right. When the input was received from the camera, the pixels were put into the neural network and the values propagated down the network into the output layer. The direction to steer was determined by finding the node with the highest output value, and using the values of the surrounding nodes to "fine-tune" the steering value.

Using this technique, the neural network could successfully navigate the vehicle along the road after only around five minutes of training. It is worth keeping in mind that images from the real world were used, which are always noisy and always changing. Trying to "hand code" an algorithm to perform the same task would be supremely difficult, and the fact that the very simple neural network was able to learn so quickly showed that neural networks may have a very important role to play in visual perception systems.

Subsequent papers using the ALVINN system include "Vision Guided Lane Transition" [3] which examines the issues involved in allowing an autonomous vehicle to change lanes while "DAMN: A Distributed Architecture for Mobile Navigation" [4] looked at integrating perception driven navigation with higher-level goals.

The only other related research found was entitled "AI Driven Vehicle" [5], which is from a university project. In this is a vehicle needs to steer its way around a world that has obstacles in various positions. At each time step, the world was partitioned

into three areas: the area to the left and in front of the vehicle, the area directly in front of the vehicle, and the area to the right and in front of the vehicle, with everywhere else ignored. The distances to the closest obstacle in each partition was used as inputs into the neural network, which would use this information to avoid the obstacles. This was therefore only simulating robot vision, and was a huge simplification of vision guided navigation, however it did manage to avoid the obstacles with only a few neurons.

Design

This section gives a high-level overview of the system, starting with an overview of the virtual city, followed by an explanation of the training, which includes some notes on design features included to enhance the performance of the neural network.

Overview of the city

The simulation takes place in a virtual city; figure 1 shows an overview of the city. The city was designed to be simple, yet at the same time provide a variety of situations for the agent.
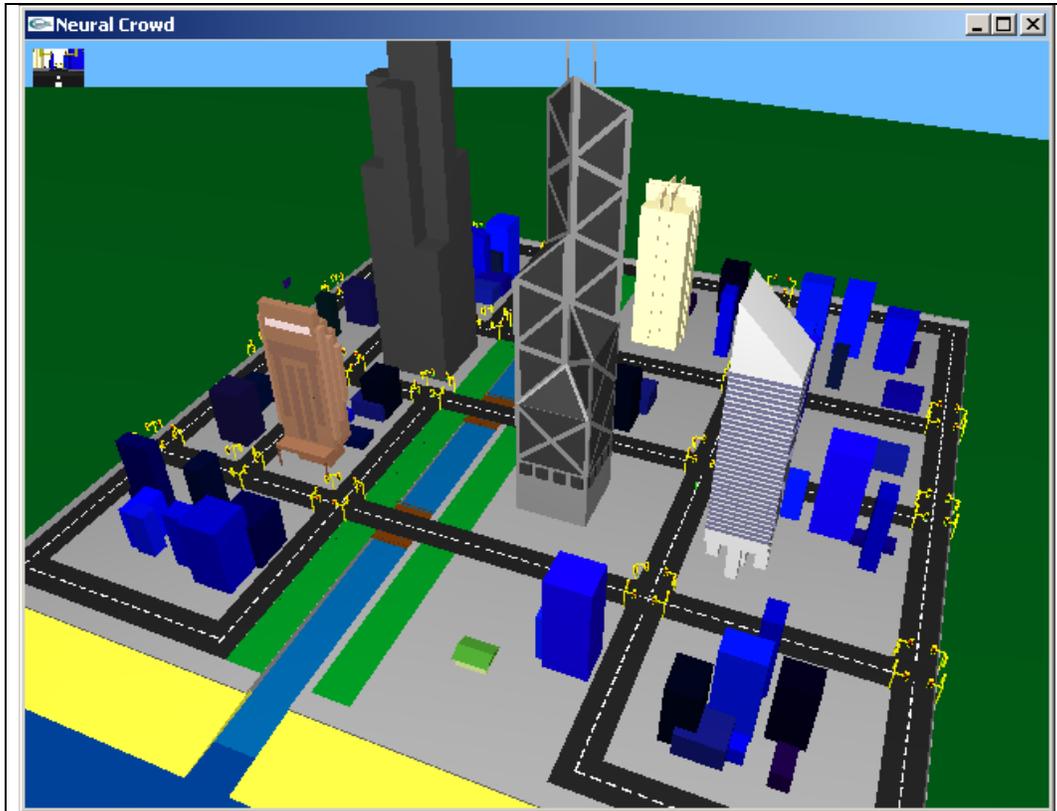
Figure 1: an overview of the city

In terms of simplicity, the places that the agent can walk in all take place in the same plane, so there are no hills, steps or slopes etc. While this makes it easier to program, this simplification was decided upon in order to make it easier for the agent to learn. For example, a staircase or a hill in front of the agent may appear to look the same as a wall or an obstacle, so it would take a lot longer for the agent to learn how to distinguish the difference. Banishing this simplification is future work.

Another immediately apparent simplification is the level of detail of the graphics. For example, many of the buildings are simply blue boxes and there are no complicated objects such as trees etc. Again, this was done to keep it simple, and as is explained below, the view from the agent's point of view is very low resolution so little details would not be seen by the agent anyway. Of course if this technology was to be used in a real-life situation, then the simulation should be as lifelike as possible. For the purposes of this project however, the simple graphics used were satisfactory.

What was important though, was the use of perspective so that the agent can learn to distinguish between objects close to it and far away from it. Basic lighting was also used (ambient and diffuse lighting to be specific) as this also gives clues as to the distances of objects. These considerations were important because as we have all experienced, sometimes a large object far away can cast a similar pattern on our retina as a small object close to us, however we are still able to distinguish between the two due to several reasons such as perspective etc.

It is important to note that the agent only knew what it saw, or in other words it had no information about the city, such as a map nor which objects existed where, nor how far apart the different objects were. This is because the object of this project was to investigate how an agent could find its way around a given area where none of this information was available, such as a robot when placed in a before unseen area in the real world.

Figure 2 shows several features of the city and why those features were chosen.

(a) The streets

(b) The grass and river

(c) Skyscrapers	(d) The beach

Figure 2: a selection of features from the city. In (a), a typical street scene is shown. The roads are dark-coloured and include fixed size, white coloured road markings. These help the agent identify where roads are, and to follow the roads. There are also working traffic lights that agents should obey. Finally, the footpaths surrounding the road are light grey. In (b), the grassy area and river are shown, along with the bridges. These exist solely to give more variety to the city so that, for example, an agent will learn to follow the road regardless of whatever is surrounding the road. In (c), the skyscrapers are shown. These are here not only for show as their existence further challenges the neural network due to their size. This is because a distant skyscraper may be the same size and shape as a small, close object, so the network must learn to distinguish between the two. Finally, (d) shows the waterfront, which again exists to provide more variety in the world.

Training

Training a neural network is composed of two parts: gathering training data, and back propagation, which is using the gathered training data to modify the weights of the neural network.

Gathering Training Data

Gathering training data was simply a matter of controlling the agent with the cursor keys. If the goal was to keep the agent in the centre of the road, then the human trainer would simply steer the agent around the city, ensuring that the agent stayed in the middle of the road. Every 10 frames a screenshot was taken from the agent's point of view, and this was saved along with the steering amount and some other information into a text file, which would create one training instance.

When gathering the training data, there were several important points:
- The screenshot taken was very small compared to what would be shown on a computer screen. If the simulation was run at 640 x 480 pixels, then there would be a total of 480,000 pixels to input into the neural network, which would be slow and memory intensive. This high level of detail is not needed when controlling the agent; rather having only around 750 pixels was sufficient.
- Even though the agent's view of the world was much smaller than the trainer's, it was important that the aspect ratio (i.e. the width to height ratio) was maintained. This is because otherwise the human trainer may see and react to

some obstacle near the side of the screen which the agent would not be able to see, and so from the agent's point of view the human trainer would be reacting to something which was not there, and so the agent would learn to perform evasive actions, even when nothing was there.

- The agent also sees the world in greyscale. The red, green, and blue pixel values were converted into one greyscale value between zero and one. Due to the three-dimensional nature of colour (i.e. hue, luminance and saturation), allowing the agent to see in colour would mean each pixel would require three values, and hence the already large input size of the neural network would be three times larger. For a task such as obstacle avoidance or path finding, the actual colour of objects is not so important as seeing that they are there. After all, we can watch black-and-white footage of street scenes and know where the road is, where buildings are etc, so this simplification was seen to be perfectly acceptable. When talking about the "blue river" or the "green grass", even though the agent will see them both as grey, they will almost always have different intensities and so the agent will be able to distinguish between them.

- A common problem with this type of training is that because the human trainer is able to stay on the road, almost all the training instances will be of the agent in the middle of the road, where it should be. However, when the agent is using the network to decide how to drive, it is likely that sometimes it will end up on one side of the road, or off the road completely, or heading directly towards an obstacle, all of which were not covered in the training examples. The trainer cannot simply just steer the agent off the road in order to show it how to recover because otherwise the agent will learn to sometimes swerve off the road for no apparent reason. The obvious solution was to allow the trainer to turn the recording of the training on and off. When driving along the road, the trainer could stop recording, move the vehicle to the side of the road, start recording again and immediately correct the agent's position. This allows the agent to learn what to do in such situations without teaching the agent to get *into* those situations.

- Above it was stated that only every 10th frame was recorded. This was simply done to reduce the number of very similar training instances in the training set.

Obviously from one frame to the next, neither the scene nor the steering will have changed much so not only is it unnecessary to record every single frame, but it would slow down the back propagation stage of training.

- Controlling the agent in the simulation is much like controlling a character in a first person shooter game. However, in most games of that type, when either the left or right arrow key is held down, the character turns at a constant velocity until the key is released, when the character immediately stops turning. While this is easy to control, if used as training data there would be only three values: a value for left, a value for no turning, and a value for right. The network would not be able to easily figure out when to turn gently (e.g. when there is an obstacle in the distance) and when to make a hard turn (e.g. when the obstacle is right in front of the agent). Therefore the agent had angular velocity as one of its properties, and pressing the left or right buttons would decrease or increase this respectively. The human trainer could then make long, gentle turns, or sudden sharp turns as required. Using an analogue input device such as a steering wheel would remove this requirement.

- Using a neural network of this type has the problem of remembering what the agent has seen, or short-term memory loss. The problem is not simply that the network can't remember where it has been before, but it can't even remember what happened a fraction of a second earlier. If it was heading straight for a wall, it may decide to turn left, but one frame later the inputs may have changed slightly such that it now decides to turn right. This was a common problem in this simulation, where the agent, approaching a wall straight on, would swerve to the left, then to the right, then to the left and so on until it hit the wall. This was combated by inputting the previous steering value into the neural network, which would allow the agent to "remember" what it was just doing, so in the example above if it decided to turn left, in the next frame the fact that was previously turning left would help to persuade the agent to continue turning left. The forward velocity was also input, just in case this had an influence on the steering.

- When training the agent for path finding, a random destination is calculated, and the goal is to reach that destination while avoiding obstacles. When the destination is reached, a new random one is calculated. The destination was

input into the network as the angle between the vector pointing in the direction that the agent is currently moving in and the vector between the destination and the agent's current position. Because the agents and the destinations were at the same height, this angle therefore represented the angle that was needed to turn in order to be moving directly towards the destination[1]. In addition to this, the distance to the destination was also used as input, as the distance may affect the behaviour of the agent when path finding. As in any programming paradigm, "garbage in, garbage out". In other words, it was very important that the human trainer perform the path-finding job well. Rather than giving the numerical values of the angle and the distance to the destination, a large arrow was displayed at the top of the screen pointing to the destination, and the destination itself was highlighted with a bright red cylinder that extended into the sky. With these two additions, it was always very clear where the next destination was, and hence good training examples could be made. It is important to note that these visual features were not made available to the neural network though.

- All inputs were normalised to be a number between zero and one before being used as input to the neural network, which meant minimum and maximum values had to be defined for each input. For the pixel data, this was easy; zero was black, one was white, and everything in between was grey. Minimum and maximum rotational and linear velocities were defined to make, for example, zero hard left and one hard right. The angle to the destination obviously had minimum and maximum values of negative and positive 180 degrees respectively. The maximum value for distance to destination is not so obvious however, so a value of 300 metres was used as a cap, where any value over that was set to be one[2]. By normalising all values, any bias about which inputs are more important than others are eliminated, leaving the neural network to find for itself which inputs are the most important.

---

[1] The usual way of calculating the angle between two vectors is to take the arccosine of the dot product of the two normalised vectors. However, just using this formula will not show whether the current movement vector is to the left or to the right of the destination. To achieve this, the "left" vector was defined as the destination vector rotated ninety degrees counter clockwise around the Y-axis. The angle between the movement vector and the left vector was then calculated. If the angle was less than 90°, then it meant that the current movement vector was to the left of the destination vector.

[2] The city had dimensions of 400 x 400 metres.

- While neural networks can tolerate some noisy data, it is of course beneficial to minimise the amount of noisy training data. For this reason, each time training started a new file was created using the current date and time as the filename. During training, the training examples were written to this file, and over several training runs several files would be created. This meant that if the human trainer made a mistake, the mistake could be removed from the training examples by deleting the most recently created training file, rather than having to delete all the training examples.
- The background colour was set to light blue to look like the sky for the human, and pure white for the agent in order to help it to better distinguish between the objects in the background.

Figure 3 shows an example of what the human trainer sees compared to what the neural network "sees".
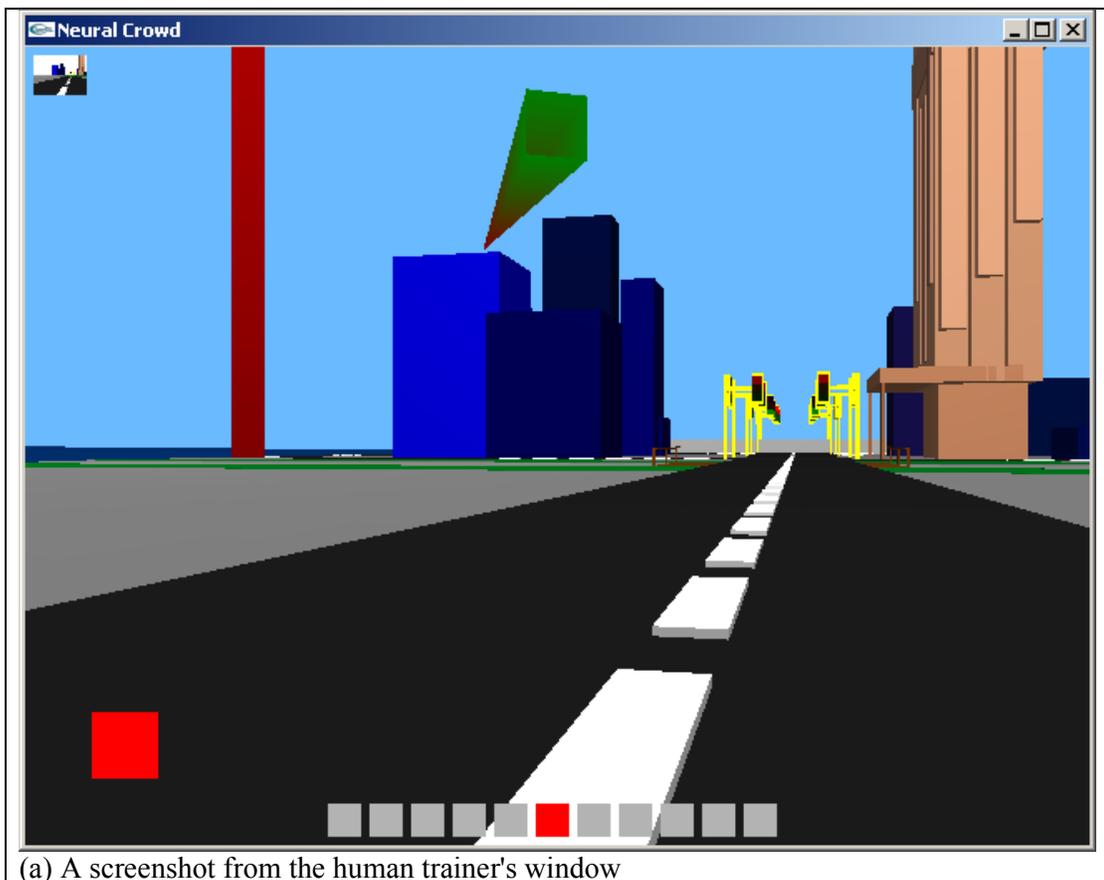
(a) A screenshot from the human trainer's window
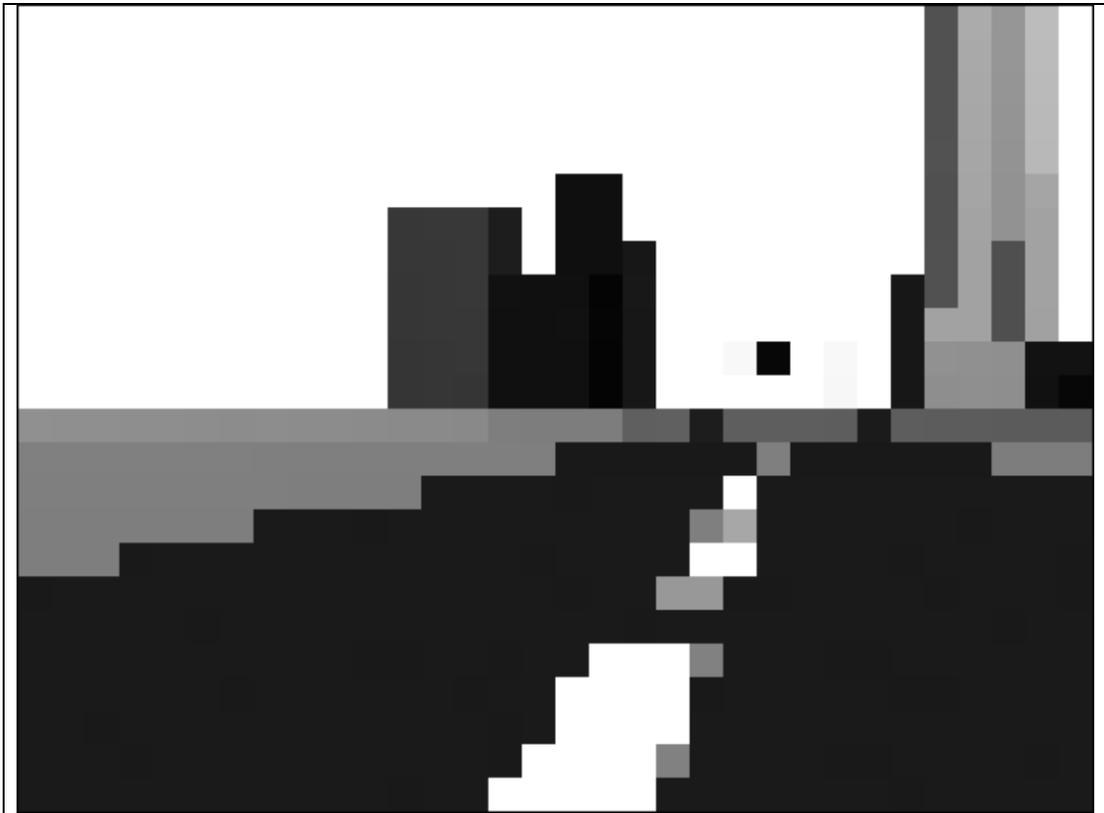
(b) The (enlarged) view the agent gets for the same scene

Figure 3: an example of what the human trainer sees (a) compared to what the neural network "sees" (b). The human trainer's window includes at the top of the screen an arrow pointing to the destination, a large red marker showing the destination in the scene (in this case on the left), a visual representation of the current steering value at the bottom of the screen, a red square at the bottom left of the screen meaning that training is in progress, and finally a small window at top left-hand corner of the screen shows what the agent is seeing. The view the agent gets is greyscale and is of a much lower resolution than the trainer's window. It is also without the auxiliary graphical components described above, however the neural network is given the numerical values of the angle and distance to the destination along with the linear and angular velocities.

Back-propagation

The back-propagation stage is where the neural network gets trained. While the purpose of this report is not to explain the theory behind neural networks in any detail, a brief explanation of what happens during back-propagation will be given now. The inputs of the neural network are each linked to all of the hidden nodes in the network, which in turn are all linked to the output node. Each of these links has a weight, so that when inputs are put into the network, they are multiplied by the weights and these become the inputs for the next layer of the network, until the output layer is reached, which will then have a value which is the network's estimated answer.

The training of the neural network is therefore just the job of finding the weightings of the links which give good results, which it does using the training examples provided. It does this by first giving the weights random values, and then going through a loop, with each iteration using one of the training examples. For each training example, the inputs are fed into the neural network and the neural network output is calculated. This calculated value is compared to the actual value supplied in the training example, and the difference between these two values gives information about how the weights need to be changed. For example, if the calculated value is too large, then the weights need to be lowered. The links which already have the largest values are presumed to be the most incorrect, and so those links need to be changed the most, so each of the nodes in the previous layer are given estimates of what they should have been, and these estimates are compared to the calculated values, so that this process can bubble up to the top of the network. This process can continue until either the error of the network is small enough or a predefined number of iterations have run.

Upon completion of back-propagation, the network is written to a text file, which can be loaded later when running the simulation.

Commandeering

When providing training data for a neural network there are several aspects to consider. One of these is the amount of training data to provide, which in most cases is unknown until training has been completed and the network can be tested. Another is the need to provide a distribution of training examples which is not biased and which approximates what the neural network will see when used with new data. For example, if in the training data there were significantly more left turns than right turns, then on new data the neural network would be more inclined to turn left even when it should turn right. Without a variety of data the neural network will not learn to generalise or to observe the features of its inputs that are important, or more likely will observe features that are not important. For example in this simulation, when an agent was taught to follow the centre of the road, only turning when it was approaching a wall, but was only taught in the areas of the virtual city where the road

was surrounded by the grey foot paths, the agent performed well in areas of the city that it hadn't seen before as long as the foot paths remained. However, when the road went through the section of the city surrounded by green grass and next to the river, the agent got confused, turning wildly to the left and then to the right. While the best strategy for the agent would have been to concentrate on the white lines in the middle of the road, clearly it was also taking into account the areas around the road. In this example, it is quite obvious that while training a greater variety of areas should have been gone over, however it is not always so obvious, for example when avoiding obstacles, there may be some combinations of obstacles together which may confuse the network, without being obvious what those combinations would be to the human trainer.

For these reasons, the idea of "commandeering" was used. After initial training of the neural network, the agent would then be allowed to run in the environment, using its neural network to decide how to react. If the training data was biased to always turn to the left for example, the agent may start to turn left when it should really turn right. At this point, the human trainer can "commandeer" the agent, turning it to the right, and then relinquishing control to the neural network once again. During the time the trainer was controlling the agent, the agent was recording what was happening, adding this new training data to its training set. In this way, the training set becomes less biased, and any unforeseen difficulties or situations that the agent encounters will become covered in the training set.

An old saying says that when a pole is leaning too far to one side, it will lean the same amount in the opposite direction after being "fixed". There is certainly the danger that a biased neural network will become biased in the opposite direction if too much commandeering is performed. The human trainer is relied upon to use their judgment to know when to stop training. If, after some commandeering, the network is retrained with the new data, using the newly created network for subsequent training, then the risk of this happening is diminished greatly. Other methods do exist, such as that used in [1], which used a bounded sized training set. In this case, when a new example was added, the example removed was that which made the average steering direction of all the examples to be straight ahead, which ensured that the neural

network was never biased to turn in one direction. No such methods will be used in this project.

Implementation

This section will first go over the implementation of the virtual city in terms of the graphics and collision detection and then over the implementation pertaining to the neural network.

City implementation

The simulation was implemented in C++ using OpenGL and the GLUT extensions. ColDet, an open source library written by Amir Geva[3], was used for the collision detection.

The objects in the world could be broadly broken into movable objects and static objects, with the former including people and cars and the latter including roads, traffic lights and buildings. Object-oriented programming was used represent the objects, with Car and People objects extending the MovableObject object while the buildings etc extended from the StaticObject object, with both the MovableObject and the StaticObject objects extending the WorldObject object. The WorldObject object contained properties common to all objects such as its position in space and methods such as a draw method, among other fields. The MovableObject object then had properties such as angular and linear velocity and a method to move the object in each time step.

The collision detection software worked using triangles, so all objects in the world were represented using triangles. Rather than hard coding the city into the simulation, a proprietary text file format was created to load in the map. The following shows an example of the map input file:

```
Road   96.0 0.05 0.0    0.0 90.0 0.0    0.1 0.1 0.1    200.0 0.0025 8.0
```

---

[3] Available at http://photoneffect.com/coldet/ as at June 2005

```
TrafficLights -96.0 0.1 40.0    0.0 90.0 0.0    0.1 0.1 0.1    9.0 5.0
9.0    3 600 300 600 0
Model -10.0 0.0 -25.0    0.0 180.0 0.0    0.0 0.0 0.0    18 100 18
boc.model 0
```

The first line says to create a road, with the following three numbers being the X, Y, and Z location of the centre of the road, followed by the rotation of the road (in this case, it is being rotated 90° around the Y axis), the colour of the road (i.e. the red, green, and blue components between zero and one) followed by the size of the road (in metres).

The next line creates some traffic lights. It has much the same format as the road, however it contains four extra numbers. The first extra number specifies the number of roads coming into the intersection, which in this case is three which means it is a T-intersection. The following numbers specify the phasing of the traffic lights, that is to say each number is an amount of time that the light should stay green for. In this case, one of the lights stays green for only half the time as the other two (the extra zero on the end is not used as there are only three roads coming into this intersection).

Finally, a building is created. The keyword "Model" specifies that this will be a StaticObject object and that a file exists including the model information for the object. The final number on the end specifies whether to use the triangles that will be loaded from the file for collision detection or not. In this case, the '0' specifies that the model should not be used for collision detection, so rather just a simple box which takes up the same space as the model will be used. This means that a complicated model with hundreds of triangles can be drawn on the screen, while only a few triangles are needed for the collision detection, which is obviously much more efficient, but can only work if the model is box shaped. The model files are in another proprietary format. The following shows an example of a model file:

```
0.5 0.1587 0.5
# the scale factors to multiply it so all the
# numbers are between -0.5 and 0.5
# recommended ratios are 1.0 6.3 1.0

Colour 0.5 0.5 0.5

Box 0.0 0.0 0.0    2.0 1.0 2.0

Rotate 45.0 0.0 0.0
```

```
Translate 0.0 1.0 0.0
Triangle   -1.0 0.0 -1.0   0.0 5.5 0.0   -1.0 5.0 -1.0
```

When a model is loaded, it is expected that the dimensions of it are one metre by one metre by one metre (so they can be easily stretched by the amount specified in the map file), which would mean all coordinates in the model file should be between -0.5 and 0.5. However, it may be inconvenient to work with such numbers, especially when creating something such as a tall building where the height is so much greater than the width and depth. Therefore, in the model file any numbers may be used, but when being loaded into the simulation they need to be transformed so that every number is between -0.5 and 0.5. The first three numbers of the model file specify the scaling factor needed for each axis when loading the model to achieve this. In this example, the width and depth are specified as numbers between -1 and 1, while the height is between -6.3 and 6.3 (although anything below zero would mean the building went underground). The "recommended ratios" line is to help when specifying the size of the building when adding it to the map, so that the building is not relatively stretched or compressed in any direction. In this example, the width and depth of the building should be the same, and the height should be 6.3 times the width/depth. The subsequent lines show all the functionality of the model files, which were designed to work in a similar fashion to how OpenGL receives information about vertices. The "colour" line specifies the colour of all subsequent parts of the model; the "box" line creates a box (which is really a shortcut for creating 12 triangles in a box shape) centred at the given (model) coordinates with a specified size; the "rotate" line rotates all subsequent parts of the model by the specified amount around each of the three axes; the "translate" line translates all subsequent parts of the model by the specified amount; and finally the "triangle" line creates a triangle with the three specified vertices.

The simulation is run with the filename of a map file passed in as a command line argument. The program goes through the map file, building up a vector of WorldObject objects. At each time step during the simulation, every object has its "process" and "draw" methods called. Most static objects will do nothing in their "process" method, except for the traffic lights, which work by changing the colours of certain triangles in the model when it is time to change the colours of the lights. The

"process" method of movable objects updates the position of the object and checks for collisions. It also runs the neural network to update the rotational velocity, which will be explained more in the following subsection. The "draw" method simply loops through the vector of triangles for the model, simply feeding the vertices to OpenGL, which takes care of the rest (i.e. drawing the 3-D scene onto the 2-D screen buffer, taking care of clipping and perspective mapping etc).

Each agent needs to be able to see the world from its own point of view. Currently, a sub window is created for the agent with a size of 32 x 24 pixels (which has the same aspect ratio of the 640 x 480 pixel window that the simulation defaults to). In each time step then, both the main window display and the agent's display are updated using the same draw method, only from different points of view. The agent can then read the pixel values from its sub window to use as input into the neural network. Using a sub window in this manner is quite restricting because each agent would require its own sub window. If it was found that an agent only needed to run the neural network every x frames, then this could be improved by having each sub window share x agents. This is still not optimal however, so a future improvement would be to use off-screen rendering so that no sub windows would need to be created.

It is important to remember that due to the nature of this project, i.e. controlling an agent based on pixel values, any language and graphics API would be suitable as long as the scene could be rendered from the agent's point of view with perspective, and likewise the representation of objects in the world is not important.

Neural network implementation

A neural network library named FANN (which stands for "Fast Artificial Neural Network") written by Steffen Nissen and Evan Nemerson[4] was used in this project. This library exposes a neural network structure, which each movable object has as one of its properties. The two most important methods are the "train" method and the "run" method.

---

[4] Available from http://leenissen.dk/fann/ as at July 2005.

Training

The back-propagation stage is where the neural network gets trained, and takes place in a separate program, called the "Learner". FANN expects all of the training data, where each example is just a space-separated line of the numerical input values with the expected output on the following line, to be in one large text file. The first step of the Learner is therefore to combine all the training data files created by the simulation into one large text file in a format suitable for the neural network software. The next step is to call the "train" method in FANN.

The "train" method takes in the filename of the newly created file containing all the training data, along with the requested network structure (i.e. the number of layers and the number of nodes in each layer), the learning rate, the maximum number of times to train for, and the desired error level to reach. The method then trains the network, writing regular progress reports to the standard output. Upon completion, it creates a ".net" file on the disk which holds information about the trained neural network so that it can be loaded later.

Implementing this part of the project was therefore a very simple task. A command line program was written in C++ that simply combined all the separate training files into one file, and then called the "train" method. The neural network specifications such as the number of layers and nodes were hard coded into the program, so that to change these specifications the program needed to be recompiled. Figure 4 shows an example of the Learner program training and writing a network to the disk.

```
c:\Documents and Settings\Administrator\Desktop\Neural Crowd\NeuralCrowd\Learner\Release\Lea...  _ □ X
****************************
About to convert the data files in ../simulation/output/ to one data file.
****************************

Processing ../simulation/output/2005-070-6_11.47.19.data
Finished processing 178 lines from ../simulation/output/2005-070-6_11.47.19.data

Processing ../simulation/output/2005-070-6_11.49.46.data
Finished processing 278 lines from ../simulation/output/2005-070-6_11.49.46.data


File written to fannfiles/fanninput.fan

****************************
About to create the neural network in fannfiles/brain.net
****************************
Max epochs      2100. Desired error: 0.0001000000
Epochs             1. Current error: 0.0028157646
Epochs          1000. Current error: 0.0002519939
Epochs          2000. Current error: 0.0001351214
Epochs          2100. Current error: 0.0002017354

****************************
Finished
****************************
```

Figure 4: example output of the Learner neural network-training program. In this example two training files are combined to train the network up to 2100 times.

There is no specific way to calculate the required network topology for a given problem, particularly one like this where pixel values are used as input. Therefore, experimentation is needed to find a good network topology. The input layer had 776 nodes (which is made up of the 32 x 24 pixels, the angle and distance to the destination, and three values each for the linear and angular velocities specifying the velocity in each axis[5]), and there was just the one output layer. Having just one hidden layer with five nodes was sufficient to allow an agent to successfully avoid obstacles or follow the road.

Running

The location of the agents was specified in the map file, and along with the location and size of the agent was the filename of a FANN network file, which is the output of the training programme. In this way, it is possible to specify different neural networks for different agents, so you could, for example, train one agent to move aggressively and another to move calmly.

---

[5] This would allow information such as how fast the object was moving forward, sideways and vertically, along with the rotations in three directions. However, currently only the forward velocity and angular velocity around the Y axis was used, so all remaining velocities were set to 0.0 which meant that the neural network would ignore them.

When an agent was created, the filename of the FANN ".net" file was supplied to a method in the FANN library, which creates and returns a pointer to a neural network structure. At each time step, the pixel values and auxiliary information were copied into an array of floats, a FANN method was called with a pointer to that agent's neural network structure and the array of input floats, and the output of the neural network was returned. This output was denormalised and then set to be the agent's steering value.

Thanks to the excellent design of FANN, the steps involving the neural network were very simple to implement.

Results

As was stated in the introduction, four areas were being looked at when trying to train the neural network. These were obstacle avoidance, following the road, obeying the traffic lights, and path finding, which will now each be looked at in turn.

Obstacle avoidance

The first goal was to make sure the neural network could control an agent by analysing the pixel values. Therefore, a very simple world was created, which was simply four enclosing walls with different sized obstacles distributed within them. There was neither floor nor ceiling, and the walls and obstacles were all the same dark colour. Figure 5 shows an overview of the world created for this test.
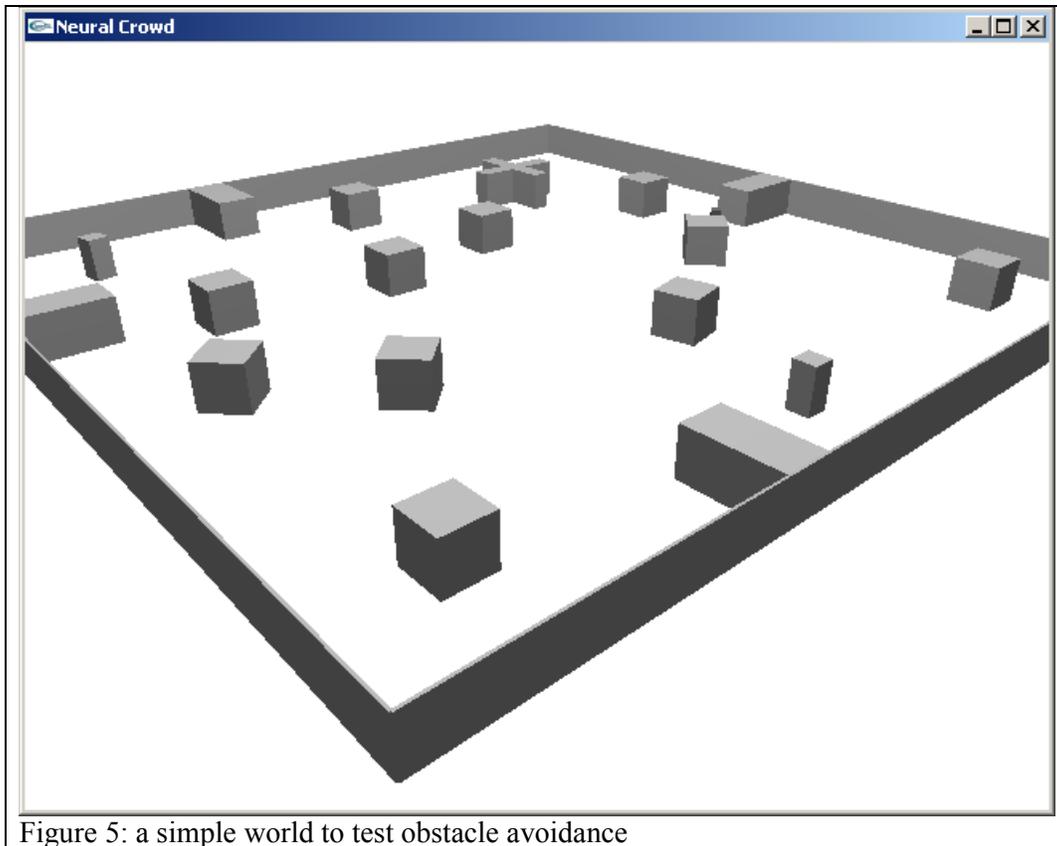
Figure 5: a simple world to test obstacle avoidance

The results from this test were very promising. After only a few minutes of training the neural network was able to steer the agent around the room, successfully avoiding all obstacles. Furthermore, the agent was able to be put in a world with a different layout and successfully navigate around the obstacles without any further training. Figure 6 shows an example of the path taken by the agent in each of the two worlds.
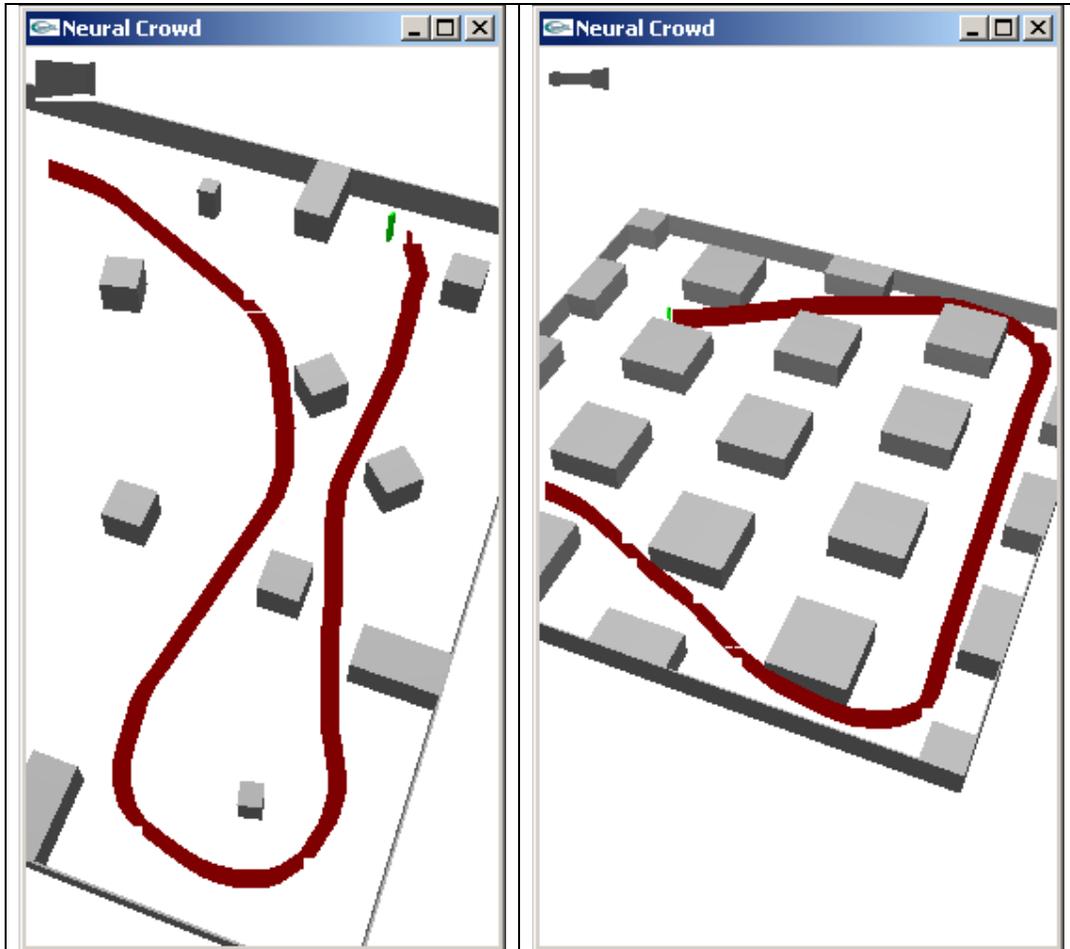
Figure 6: examples of paths taken by the agent. The red lines show the path taken. The image on the left is the world that they agent was trained in, while the image on the right is a world that the agent had not seen before.

The neural network had learnt to not worry about objects that were far away from it, or to the sides of it. It learned that when an obstacle was close, it needed to turn left or right, depending on what was to the left or right of it. Because the neural network had no information about the objects other than what it saw, it obviously learnt to categorise an object as being near or far.

This type of obstacle avoidance is fundamentally different from others, where the agent has knowledge of the position and size of the objects. In those systems, the path taken by the agent is arrived upon by mathematically analysing the situation, which can give optimal solutions when the world follows certain assumptions. However, they are normally rather simplistic, ignoring much of the reasoning that a human would use. For example, if there is a gap between two obstacles, can the agent fit

through it? This not only requires calculating the distance between obstacles, but also it requires reasoning about the shape and size of the agent. And what if one of the obstacles was rotated, on a lean, or of an irregular shape? Hand coding a function to take care of all the possible contingencies would be extremely difficult for a number of reasons, not least of all because we may not be able to identify all the rules we use as humans when performing obstacle avoidance, so training the agent by example seems ideal for obstacle avoidance in an arbitrary environment.

Road following

The next goal was to test the neural network's ability to follow the road, which was similar to the goal in [1]. The agent was trained to drive down the centre of the road, so it was presumed that the neural network would identify the white lines of the road, and steer the agent so that the white lines were centred in the middle of the road. It would need to pass straight through intersections, ignoring the road, and its white lines, perpendicular to it. It would need to turn corners, and the if it approached a T-intersection and had to choose between steering left and right, then it should turn in the direction that was easier (e.g. if it was already steering slightly to the left, then it should turn left).

During training, it was important to train the agent in both directions (so that there were plenty of examples of turning both left and right), and it was also important to train the agent how to recover when it was not centred in the middle of the road by temporarily turning off the training, steering the agent away from the middle, turning training back on and then recovering. Commandeering was also needed to improve the accuracy. Figure 7 shows an example of the agent successfully navigating itself around the roads. Figure 8 shows how the performance was improved using commandeering after it was found that the agent performed poorly around the grassy areas.

Figure 7: an example of an agent following the road. The agent was travelling in the counter clockwise direction, only turning when it needed to.

While the agent performed reasonably well, at times it did deviate from the centre of the road. In a situation such as driving on a road, it is very important that the agent does not deviate off course as this may lead to an accident. In cases where the agent has a representation of the roads in its knowledge base, it may be safer to use this representation to calculate the agent's position rather than using a neural network. Nevertheless, the research in [1] proves that with sufficient and high-quality training, a neural network can learn to steer safely on real-world roads.

(a) Before   (b) After
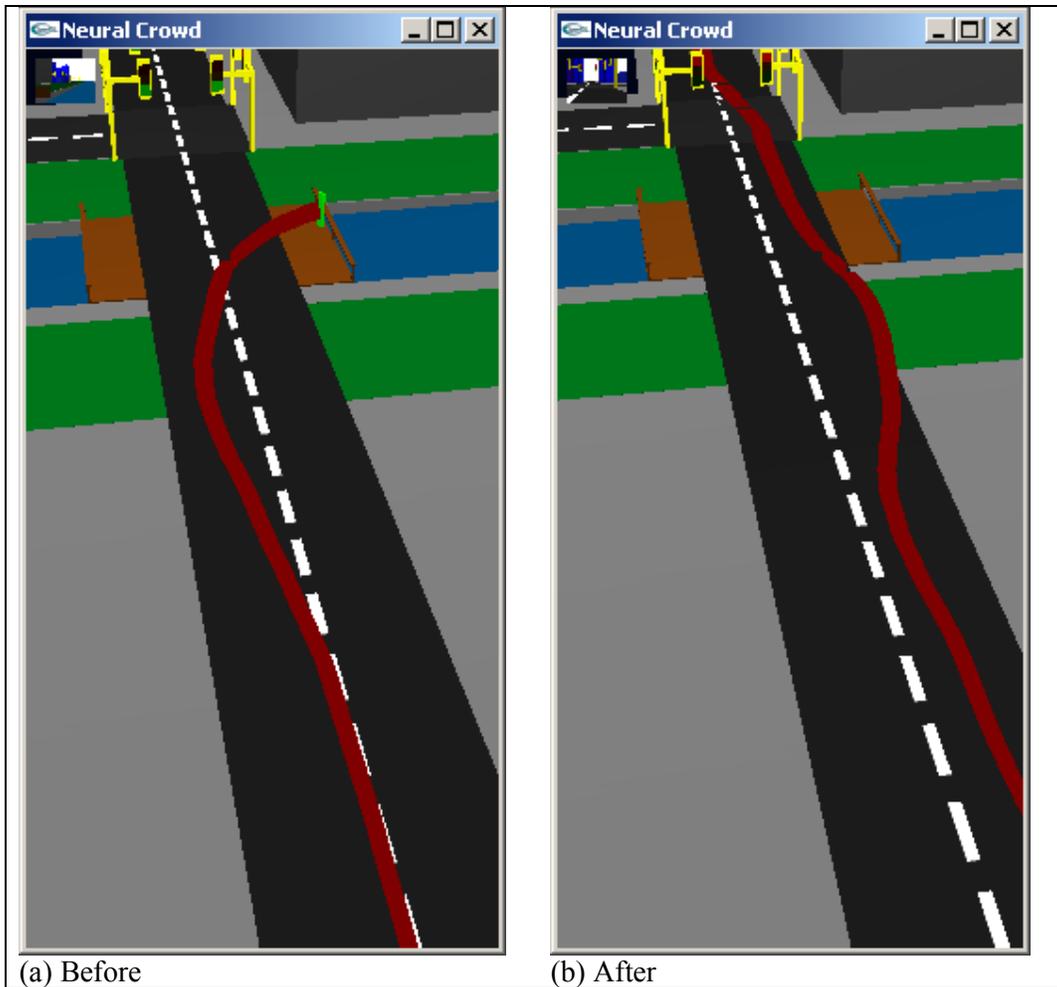
Figure 8: an example of using commandeering to improve performance. On the left is the path taken by an agent when approaching the grassy area. The change in colour of the ground surrounding the road initially confused the agent. The image on the right shows the improvement after retraining (although more training is still needed at this stage to keep the agent in the middle of the road), and illustrates the use of commandeering: the point where the agent gets confused can be seen and corrected, rather than trying to guess where the agent may have problems.

Obeying traffic lights

For this step, the neural network was used to control the linear acceleration of the agent, with steering being human-controlled. It was hoped that the agent would learn to stop for red lights and go for green lights. The behaviour for orange lights should depend on the current velocity. Furthermore, traffic lights in the distance should be ignored, and the agent should stop only when it gets a certain distance away from the intersection.

It was found that the agent was not able to learn this behaviour at all. It may be that when converted to greyscale, the different colours of the lights were too similar to distinguish between each other. Also, the size of the lights were quite small, and depending on the exact location and orientation of the agent, the lights were always in different parts of the screen, which again would increase the difficulty of this task.

Path finding

The final goal of the project was to use the neural network for path finding. This involved telling the agent to move to a random destination, and when this destination was reached a new random destination was given. This is a more difficult task than the others because the agent must head towards a destination while at the same time avoid obstacles, with these two tasks often conflicting with each other. To make it easier, the agents did not need to stay on the road and they were allowed to walk across the river rather than crossing it by a bridge.

The results were promising but far from perfect. While most of the time the agent was able to avoid obstacles and head in the general direction of the destination, its movements were sometimes erratic, it didn't take the shortest path, and it would sometimes head in the complete opposite direction of what it should have. It seemed that the destination acted as an influence on where the agent went, rather than being its *goal* to get there.

Several network topologies were tried; from having a single five node hidden layer up to having three hidden layers each with eight nodes. It was found that keeping it simple was the best strategy, with a single hidden layer with seven nodes giving the best performance, whereas having multiple hidden layers often resulted in a strategy for the agent such as "turn left no matter what", causing it to go around in circles.

Despite the problems, the fact that the agent normally headed in the correct general direction of the destination showed promise. It looks probable that with better training and more experiments with different network topologies, the agent should be

able to successfully path-find its way around the city. Figure 9 shows some example results from the simulation.

(a)
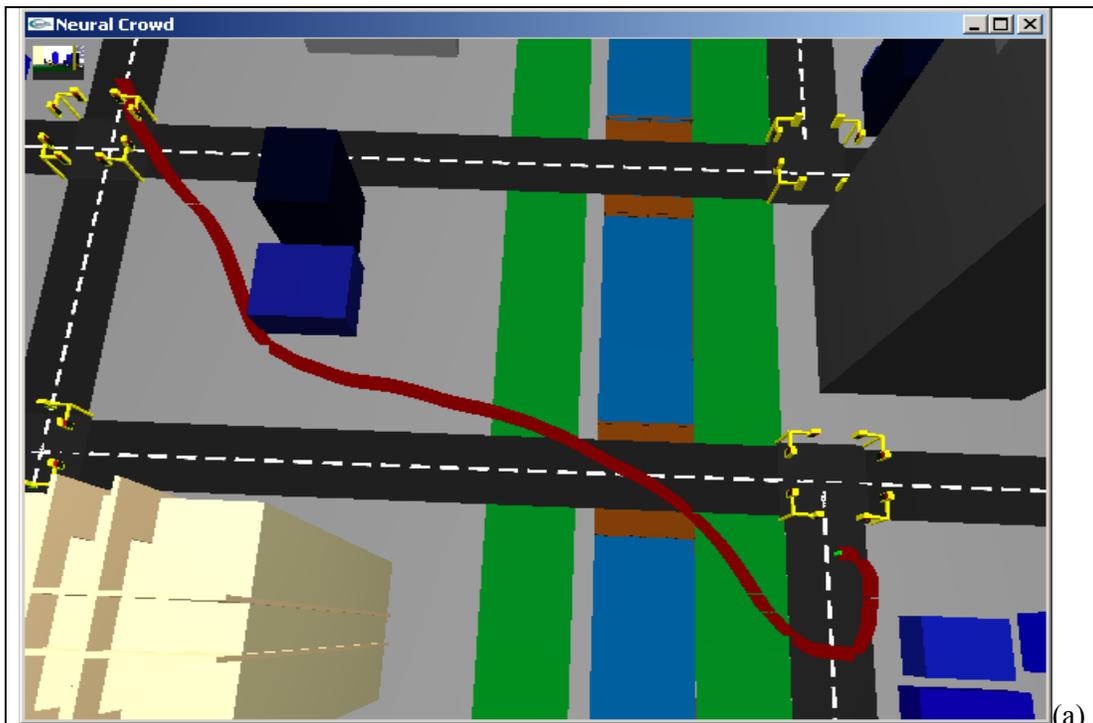
(a) The agent started in the top left of the screen, narrowly avoided some buildings, crossed the bridge, but then had a little difficulty in getting to the correct spot as can be seen by the loop at the end of the path.
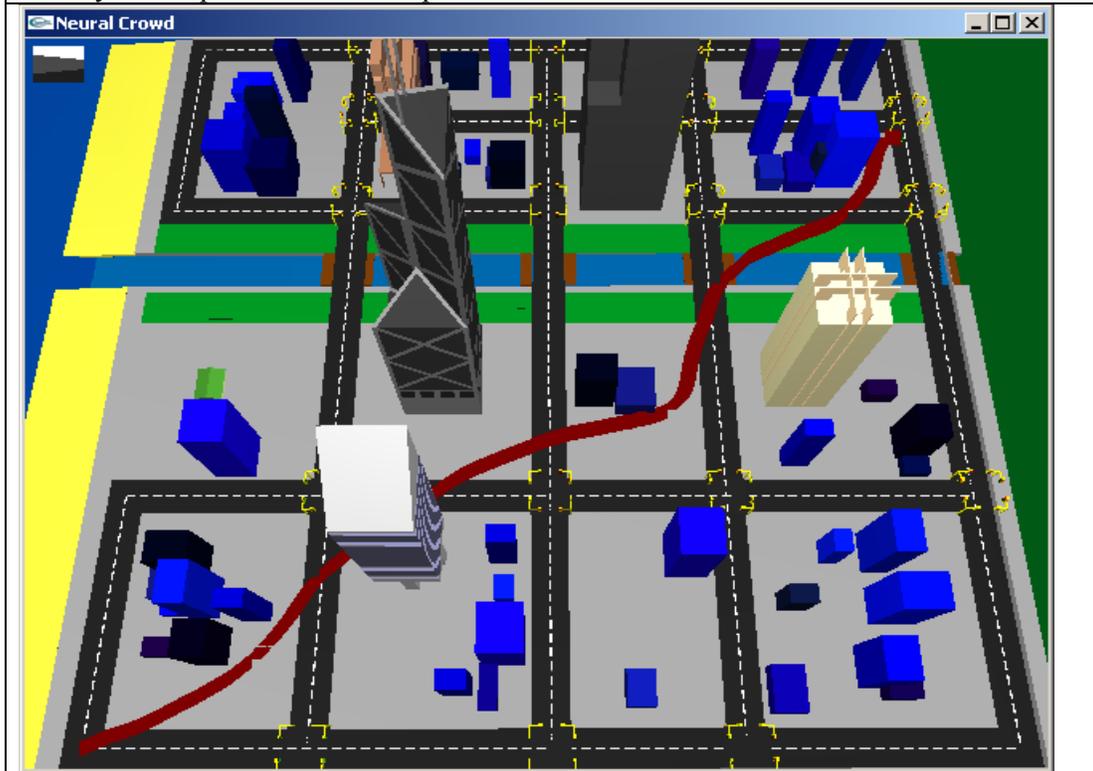
(b) Long-distance path finding, with the agent starting at the bottom left-hand corner of the screen. One of the features to note is the smoothness of the path, which, in the bottom left-hand side of the screen for example, resembles the kind of obstacle avoidance while heading to a destination that a human pedestrian may perform. In other situations though, it only serves to increase the distance travelled, as in the second half of the path where it should have just headed straight across the river.
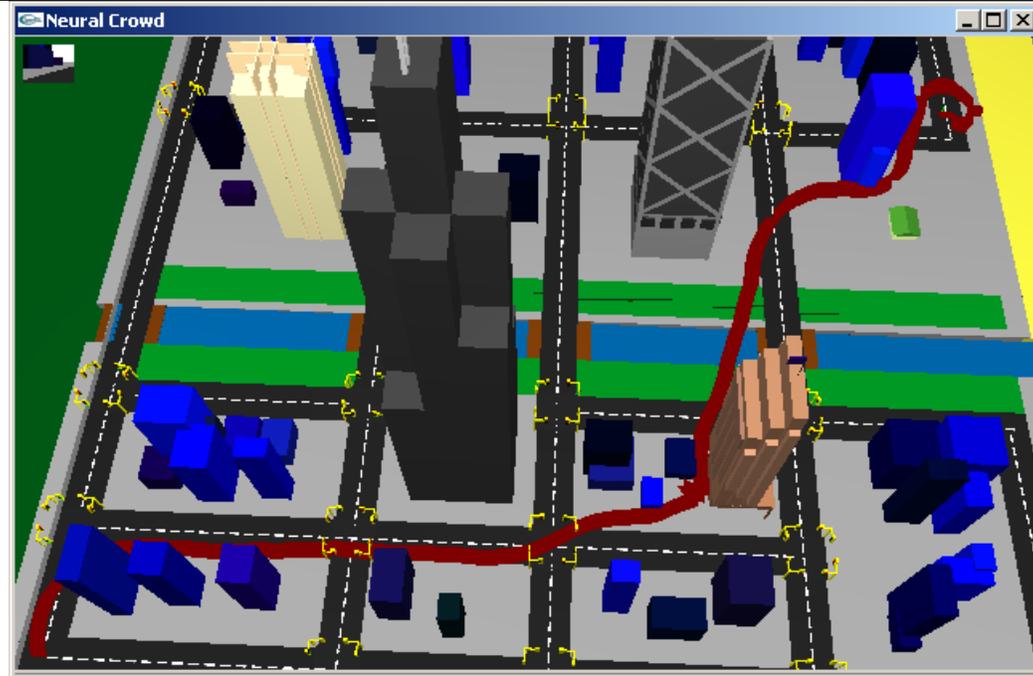
(c) More long-distance path finding, with the agent starting at the bottom left corner. Near the middle of the path, next to the brown building, the agent got stuck and had to be helped out. Once again, at the end of the path the agent had trouble getting to the correct spot, doing a loop before finally arriving at its destination.

Figure 9: examples of path finding. There was no requirement for the agents to follow the road, and they were also free to walk across the river.

It would be useful to study this further because if successful, it may be a good way to move an autonomous agent from A to B. In other words, higher-level planning could determine where the agent should go (or in which direction), and this visual based neural network navigation system could be used to get there, with the advantages given earlier such as the neural network's ability to deal with noisy data and a constantly changing environment which is inherent in the real world. Obstacle avoidance on its own is not so useful, because if a different method was used to drive the agent until an obstacle was detected, then the neural network could avoid the obstacle, however it may end up turning the agent in the completely wrong direction. Furthermore there are difficulties in knowing when to turn the obstacle avoidance on and off, or how to combine it with another method, so if the neural network could deal

with keeping the agent in the correct direction, then it should be able to smoothly guide the agent around obstacles and then smoothly turn back towards the destination.

Conclusions

In this project a virtual city was implemented in order to simulate vision guided navigation of autonomous agents. The agents were given no information about the layout of the world or which objects existed there, and instead rendered the scene from their points of view and decided how to move based on this information, much as a robot would do situated in the real world.

Specifically, the pixel values of the rendered scene were used as inputs into a neural network, with the output of the neural network giving information on how the agent should be moved. The agents were separately trained to avoid obstacles, follow the road, obey traffic lights, and path find. Training consisted of a human trainer controlling the agents as in a video game, with a log of the rendered scene and the action the human controller was taking at each time step. Separately, the neural network was trained using the data from these log files.

The results of this project were mixed, with very good performance in obstacle avoidance and path following, but poor performance in obeying traffic lights. The path finding, which involves both obstacle avoidance and travelling towards a goal destination, showed much promise. It is probable that with further training the neural network could learn to do this better, and it was speculated that in autonomous agents situated in the real world, neural network-based visual guidance might be suitable for moving the agent to a goal destination, with this destination being decided by other higher-level methods. This is because the neural network is able to learn in environments where lots of noisy data is present, and there are many variations in the scene, because with enough training the neural network is able to generalise out the important parts of a scene in a way that would be very difficult with hand coded algorithms.

Future work

As there are already working traffic lights in the simulation, an obvious next step is to have the agents successfully control the acceleration of the car. There would then be two neural networks: one for the steering and one for the acceleration, both working independently. It may be that a third neural network is required that classifies the colour of the lights, which would need to ignore the lights in the distance, and perhaps even the lights in front of it until the agent was close enough. While it may be possible to combine this with some kind of rule-based system (e.g. "if lights = red then stop") it may be better to feed the result of this network into the acceleration network. This is because acceleration depends on more than just the traffic lights; it depends on whether there are corners, obstacles etc, which would make it difficult or impossible to combine with rules. It may be that more networks need to be chained together in this fashion to achieve optimal performance. Having specialised networks chained together has the advantage of allowing each part of the chain to be trained independently, for example once the network had learnt to classify the traffic light colours correctly, there would be no danger of "untraining" it when training it to do something else, and it should also be easier for it to learn a few, separate concepts than one complicated concept.

Another important step would be to change it so the scene from the agent's point of view was rendered into its own off-screen buffer, allowing multiple agents to interact at once. The agents would need to avoid each other, and it could be a novel approach to crowd simulation. Rendering the scene from each of the agents' points of view may prove to be too computationally expensive however, so it would be interesting to see how many agents could run at once, with performance be improved x-fold by having each agent's point of view being updated only once every x frames.

In [2], Reynolds identified three rules that can be followed in order to simulate flocking behaviour seen in animals such as birds. These rules are "separation" (i.e. moving to avoid collisions with others), "alignment" (i.e. moving in approximately the same direction as the others who are close), and "cohesion" (i.e. staying close to the others). It would be interesting to see whether flocking could be implemented in

the system by having the trainer follow those three rules rather than explicitly programming them in. To achieve this, the set-up of the agent's point of view would have to be looked at carefully as animals that flock, such as birds, have their eyes located at the sides of their head to give a larger field of vision, compared to the relative tunnel vision that is currently used in the simulation.

There are many other possibilities that could be explored. For example, can a car learn to indicate before it starts turning? This would require the network to know that it was going to turn before it did. Or, can one car learn to follow another? Would it get confused when the car it was following went close to other cars? This problem may further expose the problem of the network having no memory, because if the car being followed managed to escape the field of view of the follower for even one frame, then the follower would have no idea where to go.

Perhaps the most important future work then is to investigate which problems can be solved by a neural network, and which are better to be solved using other methods.

References

[1] Pomerleau, D. 1996. Neural network vision for robot driving. In Nayar, S., and Poggio, T., eds., Early visual learning. New York, NY: Oxford University Press. 161--181. http://citeseer.ist.psu.edu/pomerleau96neural.html

[2] Reynolds, C. W. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34.

[3] T. Jochem, D. Pomerleau, and C. Thorpe, "Vision Guided Lane Transition," IEEE Symposium on Intelligent Vehicles, September, 1995, pp. 30 - 35.

[4] J. Rosenblatt, DAMN: A Distributed Architecture for Mobile Navigation, doctoral dissertation, tech. report CMU-RI-TR-97-01, Robotics Institute, Carnegie Mellon University, January, 1997.

[5] Kunzle, P., "AI Driven Vehicle", University project, retrieved from http://www.kuenzle-family.ch/philippe/gl/SeniorProject.htm