CSE 841 Project - Drive Bot

Matt Pasco

pascomat@msu.edu

October 2018

1 Introduction

The goal of this project was to create a bot that plays a driving game. The inspiration for this game came from a game implemented in python from There are 2 main goals to learn from this this report: implement a Genetic Algorithm (GA), implement a Neural Network (NN).

2 Motivation

The motivation behind this research is to better understand genetic algorithms. I am particularly interested in this research as it corresponds to my research with Dr. Cheng and Dr. Mckinley; I do not directly work with navigation as my research but other members do. The ability to simulate autonomous vehicles is extremely important. Many full autonomous vehicles cost upwards to $500,000. If one were to crash that is an incredible loss of money. Running genetic algorithms inside of a simulation allows for many generations to be trained before testing on a real vehicle.

3 Related work

A paper by Shultz et al. [3] explores evolutionary algorithms for driving. I plan to use some of the techniques in the paper, however they do use more complex image recognition for lane following. I plan to see if my algorithm evolves into the popular serpentine movement, where the car moves like a snake.

Research by Georgia Tech uses their autonomous vehicle platform, *Autorally*. Autorally uses way-point based navigation, to navigate a course, however requires many way-points. The genetic algorithm running on their test platform discovered it was more efficient to drift around turns, than making hard turns. If a traditional regression based approach was used, it may not have discovered this effect and rather lose speed to make a turn without "fish-tailing."

4 Evolutionary Computation

Evolutionary computation relies on the notion of 4 main properties: Initial Population, Selection, Crossover, and Mutation. Evolutionary computation follows similar structure to real-world biology; the best performers are selected to "reproduce" in the next generation. Performance is scored based on a predefined fitness function. Developing a fitness function in itself is somewhere where artificial intelligence could be used. A weak fitness function could score a poor performer higher than other based on simple flaws. Each member of the generation, has encoded weight values correlating inputs to outputs.

Typical machine learning algorithms follow regression based approaches, such as gradient descent. The inherit problem with this is in the case of local minima and maxima. An example of this is if one wanted to climb Mount Everest, they may have to climb several smaller mountains before reaching the base of Mt. Everest. The regression based solution will find a local maxima on top of one of the smaller hills, as it does not want to descend a hill to find the optimal solution, which is the altitude on top of Mt. Everest.

Genetic algorithms start with randomized weight sets, such that over several generations the fitness member will most likely converge at the peak of Mt. Everest.

4.1 Initial Population

An initial population is first created at the beginning of a genetic algorithm. The notion of an initial population is similar to that of a brute force approach. However, instead of trying all possible combinations of weights, which is impossible when weights are typically continuous values, a limited number of members with randomized weights are generated for the initial population. The total number of members in the initial population should be taken into special consideration.

Too little and the algorithm may not converge to the optimal solution, but to a local maxima.

4.2 Gene Encoding

In genetic algorithms a genome consists of a set of encoded genes. These genes correspond like feature weights in typical machine learning algorithms such as the Perceptron algorithm. Gene encoding is particularly helpful in the next following steps as well as storage for future usage.

4.3 Selection

After all members of a generation have been tested, members are ranked based on their score from the fitness function. The best n performers are then selected to "reproduce" through the crossover process. Picking a good number of best performers is critical, too many and the solution is not much different than brute forcing; too few and correct convergence may not be reached.

5 Programming

Inspired by the game mentioned, a new game was implemented in python. Unfortunately, due to the need of labeled data, the original plan to use *Sci-Kit learn* had to be removed and machine learning algorithms were implemented from scratch using *Numpy* for data manipulation,

5.1 Level Design

Games were designed to be modular. This allows for easy changes and swaps to courses to see how different genomes act. The files were configured using JSON (Javascript Serialized Object Notation), this allows for configuration changes such as start position, drive-able areas, and waypoints which are used for fitness scoring. An example of this shown in Figure 10.

5.2 Gameplay

Figure 1: ScreenShot of the car.

The game represents 5 core elements show in Figure 1: 1. A Red Circle, this represents the "car." 2. 3 Small Blue Dots, these are the car's "sensors." 3. Gray Square - This represents the road. 4. Green Square, this represents off-road. 5. White Square with X, represents a waypoint (used for fitness).

Figure 2: ScreenShot of the game with debug terminal.

6 AI Driver

There were four different types of drivers trained: Perceptron, Linear Regression, Multi-layer Perceptron, and Multi-layer Regression.

6.1 Linear Classification (Perceptron)

This driver implemented a multi-output linear classification model. The model receives 4 inputs: 3 being the sensor distance scaled to a [-1, +1] range, where -1 corresponds to about to hit an obstruction and +1 meaning no obstruction seen; the fourth sensor being the cars speed. Additionally, the Perceptron model has a bias weight corresponding to each of the outputs. The four outputs were 0/1 for: *turn left, turn right, speed up, brake.*

Turning left or turning right, changed the angle by 22.5^o in their respective directions. *Speed up* increased the speed of the car by 10 pixels per frame, *brake* decreased the pixels per frame by 15, or 1.5x the increase. The car's speed automatically slows by 5 pixels per frame, to act as friction.

The Perceptron algorithm shown in Figure 4, works as follows. For a given set of n inputs denoted as X_n; n weights denoted by the vector W_n, to produce output y. Each input x_i is multiplied by each weight w_i and those values are summed up, that value is added to the *bias weight* b. The summation is then passed into a *Sigmoid* function to provide an 'S' shaped curve, discretizing lower values to 0 and higher values to 1.

Typically to train their is an error training step, however we leave that for the Fitness and Selection steps.

$$S(x) = \frac{e^x}{1+e^x}$$

Figure 3: Sigmoid Function

$$y = S(\sum_{i=0}^{n} w_i x_i + b) \quad or \quad y = S(W^T X + b)$$

Figure 4: Linear Classification with n weights

6.2 Linear Regression

Linear regression model implementation follows a similar driver to the one used in the perceptron. The model receives the same 4 inputs, however only produces 2 outputs; [-1, +1] range for both speed or angle rate. Instead of a quick 22.5^o turn, the car could limit the turning rate between [-22.5, +22.5] degrees. The speed output also had a [-10, +10] value range for decelerating / accelerating respectively.

The linear regression algorithm shown in Figure 5 follows the same logic as the Perceptron in 4 however, does not go through a sigmoid function. Meaning instead of producing a classification decision with discrete values, we are left with a range of outputs between [-1, +1].

Again we skip the error training step, leaving that for the Fitness and Selection steps.

$$y = \sum_{i=0}^{n} w_i x_i + b \quad or \quad y = W^T X + b$$

Figure 5: Linear regression with n weights

6.3 Multi-layer

A neural network approach was also tested. Neural networks or Multi-layer models were of particular interest as they can find relationships between inputs, when determining which way to turn the car should consider all sensors.

Figure 6, represents a neural network with 3 inputs, 2 hidden layers, and 1 output [1]. The hidden nodes, take in the outputs from the last layer and compute and output using either a perceptron or linear regression algorithm. Since each node also has a set of weights and must compute the selected algorithms, the memory and compute time is increased.

Both node algorithms were tested, predicting their same respective output values. The hope of utilizing this model would to better handle quick turns or obstacles. 2 hidden layers were selected, with a hidden layer size of 4.

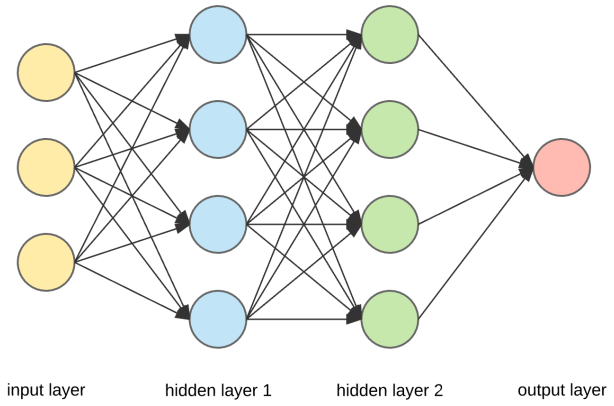


Figure 6: Neural Network [1]

6.4 Fitness and Evolution

$$f(w, t, s) = 400 * w^2 + t + s$$

Figure 7: Fitness function

We asses the fitness of a driver with 3 main metrics: number of waypoints hit w, total distance traveled t, distance from the start waypoint s. This function is represented in Figure 7. To enforce correct behavior fitness priority was added to w; this ensures cars do not drive the wrong way or get stuck in a local minima.

With an initial population size of 100 randomized driver, whose gene's are the weights used in the machine learning model. Each generation the top 10 performers are selected for crossover and mutation. The cross-over is a 50-50 split of weights from each driver. Mutation of weights involve a random value in the range of [-0.05, +0.05] change to each weight value.

The generation is stored to a text file, which the program can later use as an initial population to resume simulation.

6.5 Convergence

To stop the evolution process, the program observes the best fitness score across all generations. If the fitness score has not changed in the last 10 generations the program assumes that the algorithm has converged, and the program stops.

7 Evaluation and Results

To evaluate the models 3 tests were run on each to see how many generations it took before the car could reach the last way point. However, it should be noted that it is probable that a model could converge faster than others just due to the initial population randomization.

7.1 Perceptron

The simple perceptron actually performed the best, converging quite quickly however as more obstacles were added the perceptron struggled when there was an obstacle in the middle and the car could turn either right or left to avoid it. Meaning that the perceptron found a relationship

looking to turn opposite to where there was an obstacle found, not where there was an opening. It took about 15-20 generations before converging.

In a few tests, the algorithm converged to a serpentine approach, where the car moves like a snake, constantly turning left, right, left, right. This behavior was exhibited in tests by Schultz et al. [3].

Training this model took about 45 minutes. For its performance and time requirements, perceptron behaves well in simulation however the erratic behavior would not perform well in reality.

7.2 Linear Regression

This model also performed well, and only took about 10 more generations to converge over the perceptron model. These drivers exhibited smoother behaviors but often struggled with sharp and narrow turns.

The time to train 30 generations took about 2 hours.

7.3 Multi-layer Perceptron

The multi-layer perceptron model outperformed the perceptron as it could handle in-the-road obstacles however the time to train was much longer than both of the single-layer implementations. This model also exhibited the serpentine behavior, however due to the erratic behavior, this is not feasible for real use.

These models took 43-66 generations to converge, with a convergence time of 3 to 5 hours.

7.4 Multi-layer regression

This multi-layer could also handle in-the-road obstacles, however the training time was much longer.

This model took 51 to 74 generations to converge taking 4 to 7 hours. The behavior was much better and actually exhibited smoother behavior.

8 Conclusion

In conclusion, the perceptron model performed better than expected, however its behavior would need to be adapted before it could be used in a real scenario. Instead of instantaneous angle change,

the angle change could be a desired state, where the car may take a few seconds to reach that new desired heading.

The Multi-layer regression performed the best but also took the longest to train. Although the final model could navigate all of the courses that I designed, however I am concerned that my courses could be forcing a local minima problem. However, I would recommend this model.

9 Future work

This work could be easily adapted to try more sensors, different sensor viewings, and different outputs. I think it would be interesting to try a state-based approach with heuristic functions; ie. inputs would be: *can move left, can move right, can move forward, can move backward.* A few heuristic functions could be implemented and tested to determine a final decision. I think it could also be interesting to utilize next waypoint distance as an input, but in a truly autonomous environment, exploration without waypoints could be a possibility.

In work by Ozguner et al . for a DARPA challenge, they proposed a route selection algorithm, it would be interesting to try applying this same work in this environment [2].

References

[1] Arden Derat. Applied deep learning, 2017.

[2] U. Ozguner, K. A. Redmill, and A. Broggi. Team terramax and the darpa grand challenge: a general overview. In *IEEE Intelligent Vehicles Symposium, 2004*, pages 232–237, June 2004.

[3] Alan Schultz and JOHN GREFENSTETTE. Using a genetic algorithm to learn behaviors for autonomous vehicles. In *Guidance, Navigation and Control Conference*, page 4463, 1992.

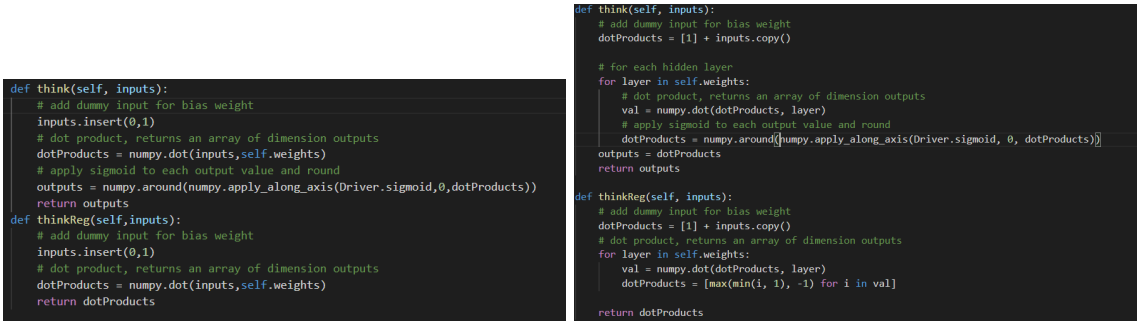
A Appendix

A.1 Work

All code was written by me and from scratch, the initial game design was used as reference however, game and level design was written by me. Originally I wanted to utilize sci-kit learn for their

machine learning models however, their models required a labelled data set. As I was looking for a evolutionary model, weights could not be used.

A.2 Python Snippets



```python
def think(self, inputs):
    # add dummy input for bias weight
    inputs.insert(0,1)
    # dot product, returns an array of dimension outputs
    dotProducts = numpy.dot(inputs,self.weights)
    # apply sigmoid to each output value and round
    outputs = numpy.around(numpy.apply_along_axis(Driver.sigmoid,0,dotProducts))
    return outputs
def thinkReg(self,inputs):
    # add dummy input for bias weight
    inputs.insert(0,1)
    # dot product, returns an array of dimension outputs
    dotProducts = numpy.dot(inputs,self.weights)
    return dotProducts
```

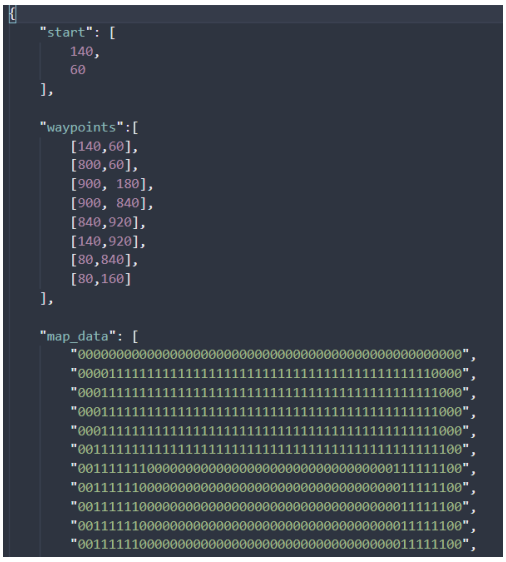
```python
def think(self, inputs):
    # add dummy input for bias weight
    dotProducts = [1] + inputs.copy()

    # for each hidden layer
    for layer in self.weights:
        # dot product, returns an array of dimension outputs
        val = numpy.dot(dotProducts, layer)
        # apply sigmoid to each output value and round
        dotProducts = numpy.around(numpy.apply_along_axis(Driver.sigmoid, 0, dotProducts))
    outputs = dotProducts
    return outputs

def thinkReg(self, inputs):
    # add dummy input for bias weight
    dotProducts = [1] + inputs.copy()
    # dot product, returns an array of dimension outputs
    for layer in self.weights:
        val = numpy.dot(dotProducts, layer)
        dotProducts = [max(min(i, 1), -1) for i in val]

    return dotProducts
```

Figure 8: Python Implementation of Perceptron and Linear Regression, both single and multilayer

```python
def crossover(pair):
    weights = []

    split = int(len(pair[0].weights)/2)

    weights = pair[0].weights[:split] + list(pair[1].weights[split:])

    return Driver(weights)


def mutate(genome):
    genome.weights = [genome.weights[i] + (numpy.random.rand(genome.weights[i].shape[0],
                                                             genome.weights[i].shape[1]) - 0.5) * .05 for i in range(len(genome.weights))]

    return genome
```

Figure 9: Python Implementation of crossover and mutation

A.3 Level Design



```
{
    "start": [
        140,
        60
    ],

    "waypoints":[
        [140,60],
        [800,60],
        [900, 180],
        [900, 840],
        [840,920],
        [140,920],
        [80,840],
        [80,160]
    ],

    "map_data": [
        "00000000000000000000000000000000000000000000000000",
        "00001111111111111111111111111111111111111111110000",
        "00011111111111111111111111111111111111111111111000",
        "00011111111111111111111111111111111111111111111000",
        "00011111111111111111111111111111111111111111111000",
        "00111111111111111111111111111111111111111111111100",
        "00111111100000000000000000000000000000111111100",
        "00111111000000000000000000000000000000011111100",
        "00111111000000000000000000000000000000011111100",
        "00111111000000000000000000000000000000011111100",
        "00111111000000000000000000000000000000011111100",
```

Figure 10: ScreenShot of a course file in JSON.