Autonomous Car
Mini-Assignment: Take a Right Turn
Understanding Model Architectures and how to model Data

Introduction

This week we start with a simplified problem statement that we will use to understand how data should be modeled given the problem statement, what potential problems can arise when trying to use a Neural Network on raw data and how to model our resources to build into an efficient pipeline to solve our task. Towards this goal and the goal of building towards an autonomous car model, we begin with a basic task of how to navigate a car by controlling its acceleration and direction to make a simple right turn given that some other cars are coming from the opposite direction in the next lane. We asked you to generate data for this particular application so that you can use it to train your architectures. We also asked you to meanwhile think on how you can model your architectures to achieve the goal of making the car successfully completing a right turn autonomously. Will a simple MLP with multi-output labels such as [Right, Left, Accelerate, Decelerate, No Action] be enough? Even if so, what challenges will the data introduce?

The Data

Let us understand the data that we are capturing. First, we are generating one file each for a successful turn and for a turn in which we collided with another car. However, we are not saving those while which neither collided with another car nor took a successful right turn. The file represents a 2 dimensional data. The number of columns is the number of frames that are recorded (Note that we are sampling these frames at ~17 fps). Each row corresponding to these columns represents the data captured in that frame. In each row we store a total of 23 values. Below is a mapping of the feature number [0-22] with the corresponding meaning of that value:

We will follow the following naming convention for the rest of this document.:
Car A: car controlled by us
Car X, Y, Z: the other three cars appearing in sequence when the game starts.
Point T: top point of the junction
Point O: the middle point where the two roads intersect
Please refer to Figure 1 to see the visual mapping of this nomenclature alongwith coordinate reference for the given game map.
The position and velocity signs are with respect to the Point O which acts as origin

0: X Position of car A
1: Y Position of car A
2: X Velocity of car A
3: Y Velocity of car A
4: Y Distance of car A from the Point T
5: X Distance of car A from car X (thus, gives us the location of car X too)

6: Y Distance of car A from car X
7: Y Velocity of car X
8: X Velocity of car X
9: X Distance of car A from car Y
10: Y Distance of car A from car Y
11: Y Velocity of car Y
12: X Velocity of car Y
13: X Distance of car A from car Z
14: Y Distance of car A from car Z
15: Y Velocity of car Z
16: X Velocity of car Z
17: Orientation of car A with respect to Y axis.
18: X Acceleration of car A
19: Y Acceleration of car A
20: Score = (Variance of the velocities with which the car moves)
21: 1 if collision detected else 0
22: (string) Action taken: one of {UP, DOWN, RIGHT, LEFT, No_Control}
Note that except for the 23rd data (action taken), all other data values are float

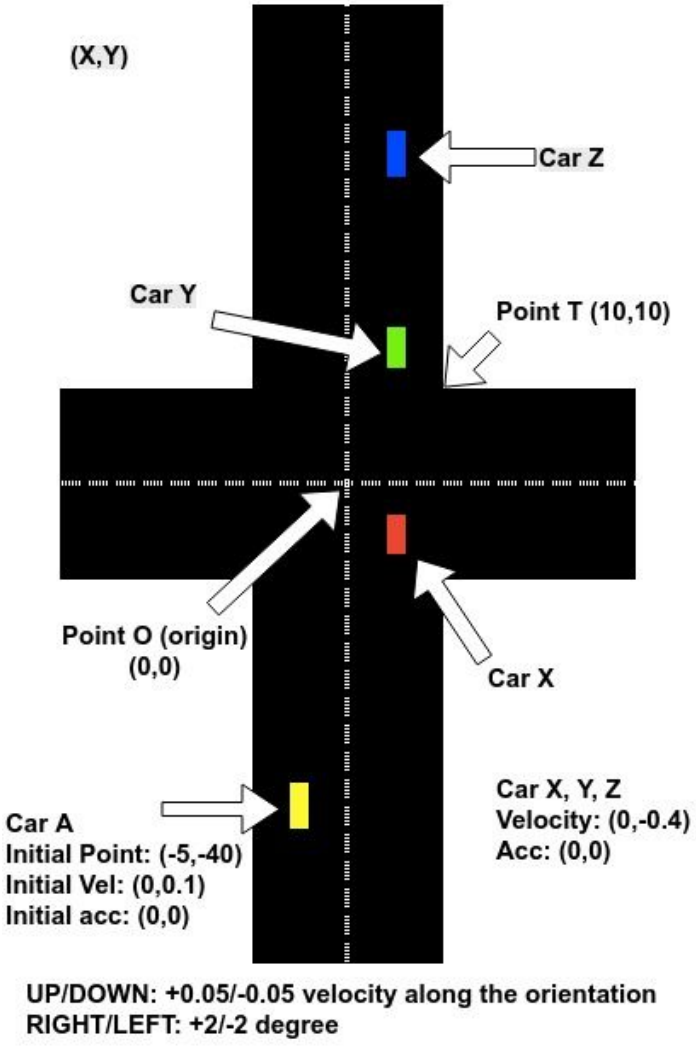

Figure 1

Data Outliers

We collected a lot of data (thanks to all the participants of the course). Collecting huge amount of data often helps us to add just enough noise to train a model that can perform robustly given any noisy data. However, we should know the difference between noisy data and outliers. The high-level target of the assignment is to train a car to take a successful right turn. However, the more important task is to do this step smoothly. Think about it, we don't want to sit in an autonomous car that has sudden/jerky motions or which breaks the basic traffic rules (going into the other lane much before the junction, taking a U-turn where it isn't allowed , speeding too much or even reversing the car on the one way road). Therefore, we have pruned the data for all such outliers (they have been placed in a separate folder in case anyone is interested). It is important to understand that we haven't removed noise from the system but only the outright outliers. The noise is still present in the form of variations in the trajectory, speed, accelerations.

Feature Modelling

The data that is provided should be used to extract meaningful features to solve the problem. This is where your intuition comes in too. Depending on the architecture you choose and the target you decide on, the features should be modelled accordingly. For example, let's say you only want to teach a car to take a right turn ignoring the fact that it might collide with the other cars coming from the opposite direction, you would not want to use the data of other cars and introduce some new features like the angle of the car from the junction, its X and Y distance from the bottom junction, etc. In such a case, you would want to use 2 models, one for just controlling the car's direction as just mentioned and another one to control the car's speed depending on the traffic.

Don't be constrained in the way you think

An important thing to note is that an autonomous car system need not be a single end-to-end model. It may as well consist of more than 2 models and most usually does, controlling different parts of the car. So, do not be constrained in your thought process and experiment with your own different models. In the next section, we will give you an outline (a hint) of one such approach, the code for which we will provide soon (between 1-2 days), we want you to experiment first, so that you make the errors that we make and learn in the process rather than understanding the code directly.

This week's homework

We have uploaded the code on the same link that was provided before under the name game_data.zip under a folder "Final Dataset". This week's lab target is that given the architecture intuition, code and data, you will build a system that achieves the target of autonomously controlling all 4 controls of the car to successfully make a right turn considering the traffic constraints by the end of this week. Right now the velocities of the cars X,Y,Z are fixed which makes the target very simple to achieve, we will introduce the option of randomizing it upto a certain extent and give you hints on how to achieve the same when we release the code for solving the current problem (i.e. within the next 2 days). The next week's session will be on Random Forest and Decision Trees, however, you can discuss your models with the mentors once you have completed the target of that particular

session. We will later make the problem wider by introducing images instead of the fixed definite values in the current data like the parameters for other cars, the turn, etc.

Possible Architectures

This is the main experimental and brainstorming exercise. Lets first go through a very naive architecture.

1. MLP to predict [UP, DOWN, LEFT, RIGHT, No_Control]

Let's say we decide that based on the data we have collected, we learn a model which at each frame predicts whether the car should take a left, take a right, accelerate, decelerate or do nothing.

Problems with such an architecture

The first problem is that each trip file lets say has about 120 frames on average. Of these 120 frames. Of these 120 frames, at least about 60% of the frames have the label "No_Control", ~25% are "RIGHT", ~10% are "UP" and the remaining 5% include both "DOWN" and "LEFT". This is the same for every trip, and hence scales similarly for the whole data that is collected. Therefore, with ~60% of the frames belonging to only one class and only 5% of the frames belong to 2 classes, the data itself is very biased and will easily attain ~70% accuracy if you are using a simple loss function like MSE. One way to work around this is to use a window around each frame and relabel that frame as some class other than "No_Control" if some other action is taken in that window. This will imply that if you took a right in the vicinity of some location, the model should learn to take a right at that location too. But, will this design work? Maybe experiment with the data and check the network behavior.

Another problem is that the other cars will be at different positions in different time frames, so unless we use data from previous frames to basically encode the behavior of each car, will the network be able to predict the behavior of car A based on only the current frame data.

One more problem is that the because there are different trajectories to reach a path, one path (in red in Figure 2) might take a right turn very late, while some other path (in Yellow in Figure 2) might start taking a right turn very early. Let's say they have constant velocity. This basically tells the network that for the exact same other parameters, my car can take a right slowly throughout the trip, or it can also take a steeper right very late (At Point A) into the trip. Therefore, for the same parameters the car can take a right or do nothing at many locations, so, will training on such a Neural Network work? Again, if you are unsure of how the model will behave, we encourage you to try experimenting and analyze the network behavior.

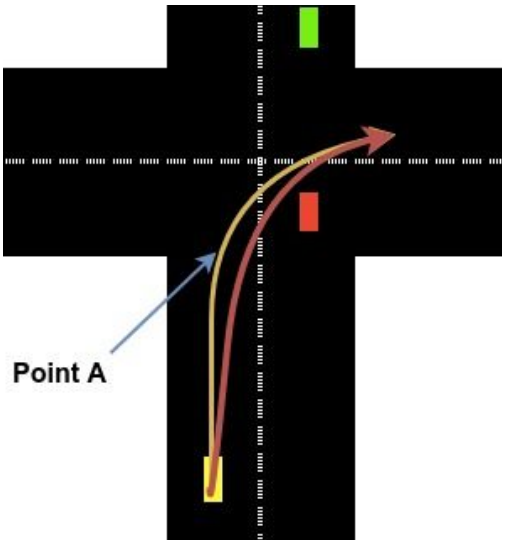

Figure 2

2. Breaking the network into 2 separate models and using data constraints to our advantage

Here, instead of trying to learn a single model which predicts everything, we will use 2 different models, one for steering the car left/right and the other for deciding the acceleration/deceleration value of the car. Let's look at the second model first.

a. Determining the value of acceleration

Let us first assume that the car steers (left/right) well such that it almost stays on a smooth curve such as the one which was fixed in game version 2 (takeright_fixed_trajectories). Now, since in the current game setting the cars (X,Y,Z) coming from the opposite side have a fixed velocity, we exactly know at what time will each car intersect the smooth curve that we are assuming our car A will follow. Let these times be t_x, t_y, t_z. Now since our assumption is that our car A will follow almost the same smooth curve with very less variations, we can almost exactly find the distance of our car A from the point at which it completes the right turn (middle of the start of right road). So basically we have:
the current speed of our car,
the distance it has to travel, and,
the time at which other car reaches an intersection point in its path.
Therefore using all this information it is a simple physics of finding an acceleration for our car A such that it doesn't reach the intersection point at the time other cars are reaching it. Therefore, we can easily determine our acceleration in this setting without the need for a ML architecture. This will be your task now. Hence the relevance of the second game version.

Now, as we said these are small building blocks which will be used to approach our task under a more unconstrained setting. So, now let us remove the constraint that the velocities of cars X, Y, Z are fixed and they have variable accelerations. So, how do we approach this problem now still

maintaining the assumption that the car is following close to a smooth curve?
HINT: Can we somehow train a model to predict the times (t_x, t_y, t_z) at which each car (X,Y,Z) will reach the intersection point based on their behavior from last few frames.

Another constraint that can be relaxed is that we don't even know the location of the car X,Y,Z as we have a camera on our car A which gives us an image of the front view. How do we estimate the times (t_x, t_y, t_z) now?
HINT: depth estimation + object recognition + multiple frames

Therefore, summarising we have a model (simple physics right now) which can determine the acceleration of our car assuming that it follows a smooth curve.

b. <u>Determining how to steer the car (left/right) such that it follows a smooth curve</u>
If we can ensure that the car somehow follows a smooth curve, we can use the model explained above in point 'a' to ensure that the car does not collides with other cars. So, we are left with the task of only taking control of the steering wheel to control the path. Now, we know that trying to tell whether our car A should take a right or a left based on how it behaved over collected data might not be a good measure. Can we introduce some other measure which doesn't have that much variation across the samples?
Let's think about the orientation of the car. We know that for the car to follow a smooth curve the change in orientation of our cars should be smooth. Moreover, we know that for a successful run, the car will enter the right turn at approximately 90 degrees with respect to Y axis. Therefore, what this implies is that for the car A to follow a smooth curve (which is what is desired from an autonomous car), its orientation with respect to its distance from the right turn will be a smooth curve. Therefore, this problem reduces to learning this curve via MLP regression (or maybe some other approach?) on our successful runs. Once we learn such a curve via regression, we can tell at each point what the orientation of the car should be and hence decide whether to steer towards right or towards left.

One major benefit of this approach is that we are ensuring that the car enters at ~90 degrees with respect to the Y axis (maybe remove the data that enters at <75 degrees to make a car behave in the optimal way. Note: the data collected is really noisy and that is why there is so much pruning over it, because we don't want our car to learn from a bad model and then drive like that). To check if this model is valid, we can plot a graph with Y axis as the orientation and X axis as the distance of the car A from the right turn (0,10) after removing the points which are not in the range [90+15,90-15] degrees at distance ~ 0.

Summarizing, we use the two models described above as follows:
'a' for controlling the acceleration to avoid collision, and,

'b' for controlling the right/left steering of the vehicle by making it follow a smooth trajectory.

Both the models are used at each step to control the steering and acceleration of the vehicle at each frame. Moreover, we can make changes in model 'a' to handle variations in accelerations of cars X, Y, Z as well as handle image data. Similarly, model 'b' can be easily modified to estimate different smooth curves for different routes that we might take on a road trip.

3. **Besides doing this, we encourage you to try out your own new designs and do tell us how it works :D**

How to integrate with a trained machine learning model

Open the file src/commander.py,
Load your trained model under __init__(self) function
Do feature modelling and get the output label under getCommand(self, featureVecs) function
We will upload a toy example for the same soon (within 1 day). Please read this doc carefully and formulate your ideas and write your architectures by that time.

How to run the game

1. Download the AIML_Hackathoon2_gamedata.zip from goo.gl/j6m96N
2. Unzip it and go to AIML_Hackathoon2_gamedata folder
3. Start your virtual environment that you created before
4. To play takeright:
   Go to takeright folder [make sure this has 2 folders: src, trips]
   Run: python src/main.py to start the game.
   This will save your game data in the trips folder.
5. Similarly for the other version of the game - takeright_fixed_trajectories:
   Go to takeright_fixed_trajectories folder [make sure this has 2 folders: src, trips_constrained]
   Run: python src/main.py to start the game.
   This will save your game data in the trips_constrained folder.

Installation Instructions

<u>Library dependencies [Your game will run if all the dependencies are met]</u>

1. Python 3.6
2. PyOpengl
3. OpenCV [not required right now]
4. Numpy
5. PIL

<u>Installation Instructions for Mac</u>

1. Install python3.6
   brew install python3 (if it fails update brew and try)

2. Install Virtualenv
   pip3 install virtualenv
3. Create a virtualenv and activate it.
   virtualenv take_right_game
   . take_right_game/bin/activate
4. Install pyopengl for the graphics of the game
   pip install pyopengl
   pip install pyopengl-accelerate
5. Install numpy, h5py
   pip install numpy h5py
6. Install opencv [not required right now]
   pip install python-opencv
7. Install pillow
   pip install pillow

Installation Instructions for Linux(16.04)

1. Update to get the latest versions of each module in your system:
   "sudo apt-get update"
2. Install Python3 virtualenv
   "sudo apt-get install -y build-essential libssl-dev libffi-dev python3-dev python3-venv"
3. Install pyopengl
   "sudo apt-get install -y python-opengl"
4. Make a path to save your virtual environment and activate it
   mkdir envs
   cd envs
   pyvenv game
   source game/bin/activate
5. Install other libraries from inside the virtual environment
   pip install pyopengl pyopengl-accelerate numpy pillow h5py

For Windows

1. Install Python3.6
   a. Download and run the .exe file using this link: https://www.python.org/ftp/python/3.6.5/python-3.6.5.exe
   b. When you run this executable, there will an option at the bottom to iclude python3.6 in the PATH, check that box and continue with the installation
2. Install Microsoft Visual Studio Build Tools (dependency for pyopengl-accelerator
   a. Link: https://www.visualstudio.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=15
3. Download DLL files
   a. Download the zip file using ftp://ftp.sgi.com/opengl/glut/glutdlls.zip
   b. Unzip it and copy glut.dll and glut32.dll
   c. Paste both the files in the folders takeright and takeright_fixed_trajectories
4. Install the virtualenv

    a. Go to windows command prompt
    b. Run: py -m pip install virtualenv
5. Make a virtualenv
    a. Run: py -m virtualenv game
6. Activate the virtual env
    a. .\game\Scripts\activate
    b. Make sure you have entered your virtual env (game)
7. Install remaining dependencies
    a. pip install pyopengl pyopengl-accelerate numpy pillow h5py

Running the game

1. Running the normal takeright game (make sure you are in the virtual env (game))
    a. cd {path to "AIML_Hackathon2_gamedata" folder}/takeright
    b. python src/main.py
       ## do not run "main.py" from inside src, else your trips wont be saved
       ## all your trips will be saved to trips folder
       ## if you are using windows make sure you have glut.dll and glut32.dll in the folder from which you run the python command
2. Running the constrained takeright game (make sure you are in the virtual env (game))
    a. cd {path to "AIML_Hackathon2_gamedata" folder}/takeright_fixed_trajectories
    b. python src/main.py
       ## do not run "main.py" from inside src, else your trips wont be saved
       ## all your trips will be saved to trips_constrained folder
       ## if you are using windows make sure you have glut.dll and glut32.dll in the folder from which you run the python command

Converting the trips file in the folder into a single dataset

Run python src/createDataset.py
It will save a single .npy file for all the trips that you ran
Note: you would probably want to make seperate files for collision and no-collision trips

---

HAPPY CODING!
and
DRIVE WELL!