Hacker Project

Traffic Simulation / Optimization

Joseph Kessler

Principle Software Developer

joe@kessler.net

CONTENTS

PROJECT OVERVIEW

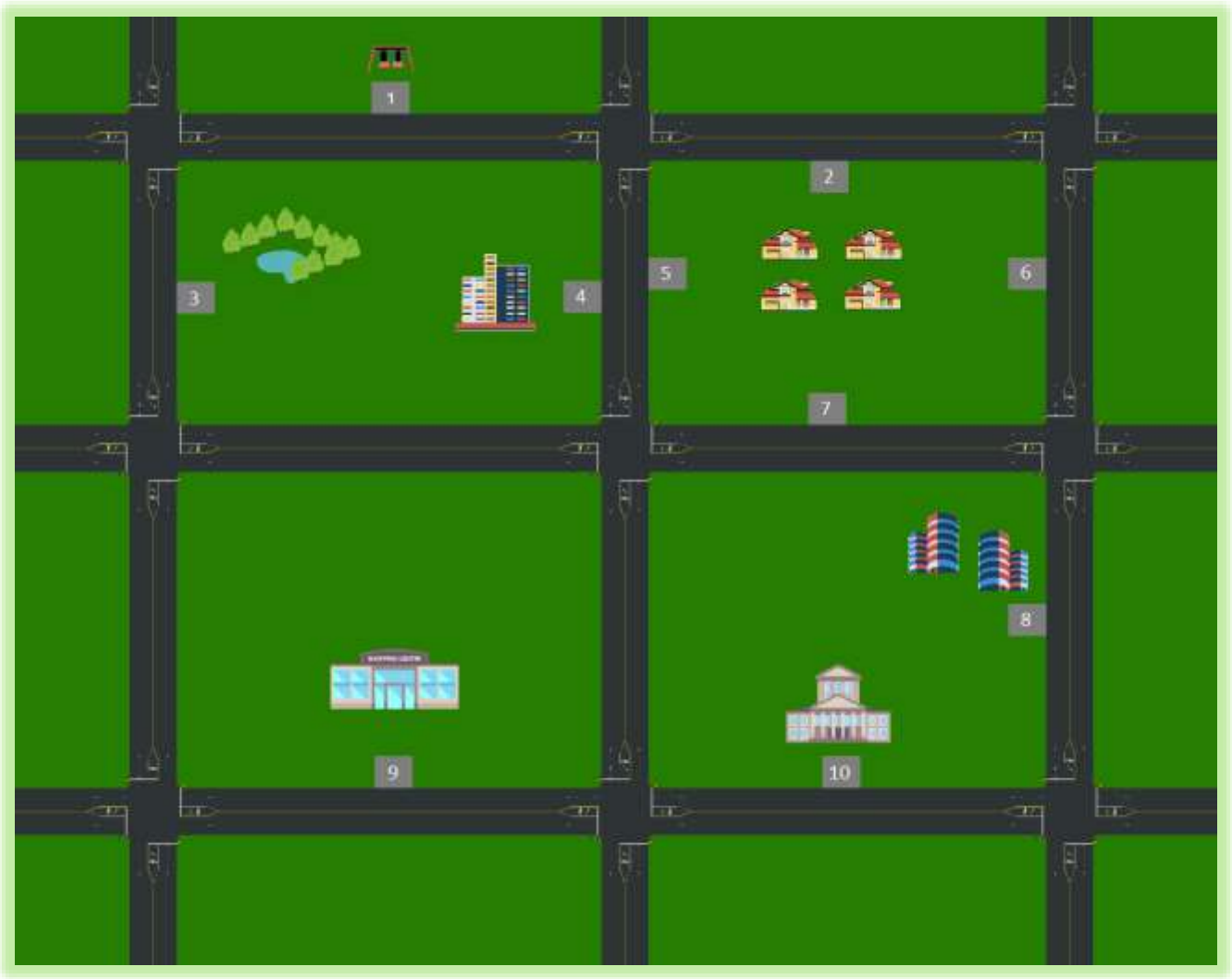
Traffic congestion is a serious problem in the modern urban landscape. Apart from the frustration of being stuck in major traffic and the impact it has on our personal lives, it also directly affects fuel efficiency, contributes to climate change, and even international relations.

One element most directly affecting traffic flow is the logic of the intersection. The simplest ones are purely timer-driven, while others also incorporate sensors to augment their efficiency. The goal of this project is to leverage Machine Learning as part of an intersection's decision process in order to maximize traffic flow and, by extension, relieve the issues mentioned above.

THE MODEL

The Neighborhood

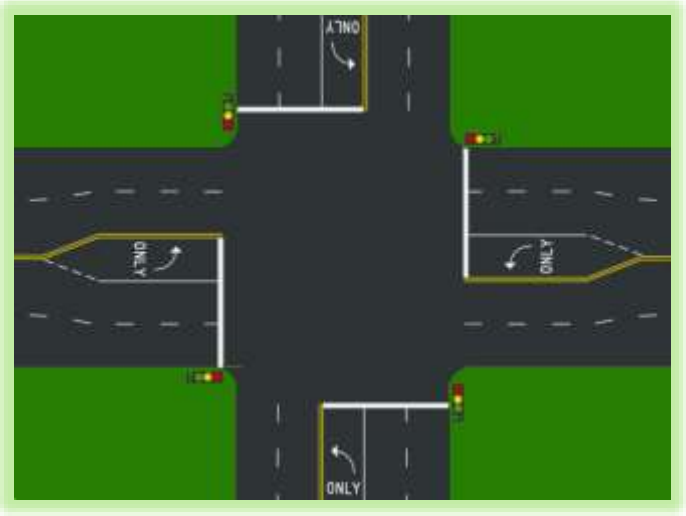
We will model simple traffic flow and intersections within a small fictional neighborhood containing several of those agents working in parallel to assess the broader impact. The neighborhood consists of several intersections with the following grid layout:



Cars will travel into the neighborhood, through the neighborhood, or may start from somewhere in the neighborhood having a specific destination. Individual cars are tracked from the time they are generated until they reach their intended destination. Probability of certain destinations and routes will vary based upon the time of day.

Intersections as Agents

The “agents” in this model are the intersections. As there are a total of 9 intersections in the neighborhood, there are 9 agents operating concurrently that together control the overall traffic flow. Each agent will incorporate intelligence with model weights managed individually from one another. At the core, all 9 intersections are instances of the same class.



Every intersection will have these basic features:

- Two thru-lanes in each direction.
- A left turn lane in each direction.
- No explicit right turn lanes, though cars may turn right according to signal states.
- There are Green and Red lights but no Yellow.

These characteristics were chosen in order to simplify the simulation, which is already fairly ambitious for this capstone.

Traffic Volume

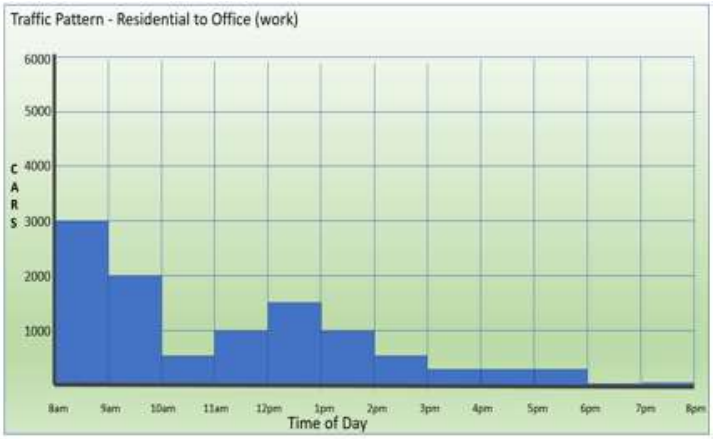
Some roads will have heavier traffic in general, and the overall traffic will reflect patterns like what we typically experience for of time-of-day, day-of-week, etc. For example, in the morning the traffic will be heavier due to people commuting to the office.

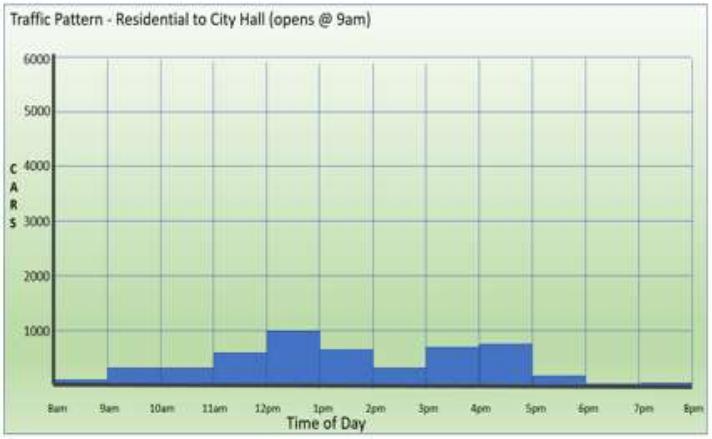
Traffic Patterns Between Destinations

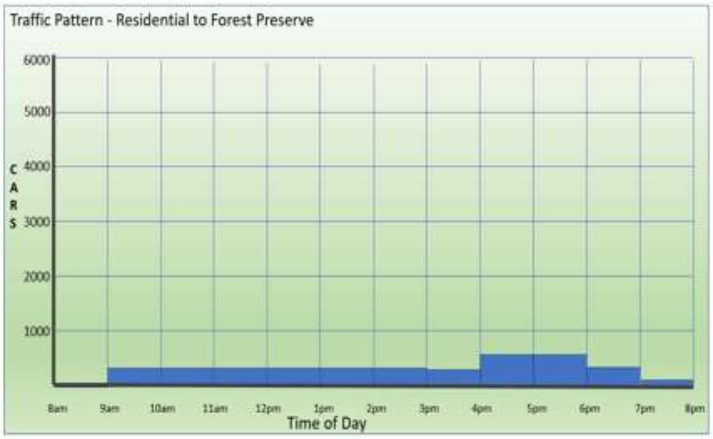
The environment must model:

- Traffic volume and likely destinations at a variety of times,
- Sensor information for sensors at each stop line, and
- Visual information that will model the appearance of oncoming cars (camera sensor)

Fictional training data will be generated for the environment. The ratios shown below are used, though there is a common scaling factor applied to right-size the simulation as needed. The environment will stochastically generate simulated “cars” based upon the traffic patterns described. Each car will be tracked from source to destination.







DATASETS AND INPUTS

Traffic Volume Data

Traffic patterns are mostly fixed and, for the sake of simplicity, are hard-coded into an array that looks something like this:

```
(from neighborhood.py)

# source, dest, source_point_id, dest_point_id, dest_point_id_road_id, source_point_side_of_street, traffic_volumes (for each hour 8am->8pm).
self.traffic_volume_defs = [
     ["residential1", "office",         10,   13,    6,  "south",  300, 200,  50, 100, 150, 100,  50, 250, 250, 250, 0, 100, 50],
     ["residential2", "office",         14,   18,    9,  "east",   300, 200,  50, 100, 150, 100,  50, 250, 250, 250, 0, 100, 50],
     ["residential3", "office",         15,   27,   10,  "west",   300, 200,  50, 100, 150, 100,  50, 250, 250, 250, 0, 100, 50],
```

```
    ["residential1", "playground",     10,   3,   6,  "south",  250, 250, 250, 250, 250, 250, 250,  50,   50,   250, 100, 0,   0],
    ["residential2", "playground",     14,   3,   9,  "east",   250, 250, 250, 250, 250, 250, 250,  50,   50,   250, 100, 0,   0],
    ["residential3", "playground",     15,   3,  10,  "west",   250, 250, 250, 250, 250, 250, 250,  50,   50,   250, 100, 0,   0],
    ["residential4", "playground",     19,   3,  13,  "north",  250, 250, 250, 250, 250, 250, 250,  50,   50,   250, 100, 0,   0],
    ["residential1", "mall",           10,  30,   6,  "south",  100,  20,  50,  75,  50,  20,  20,  75,   75,    50, 100, 0,   0],
    ["residential2", "mall",           14,  30,   9,  "east",   100,  20,  50,  75,  50,  20,  20,  75,   75,    50, 100, 0,   0],
    ["residential3", "mall",           15,  30,  10,  "west",   100,  20,  50,  75,  50,  20,  20,  75,   75,    50, 100, 0,   0],
    ["residential4", "mall",           19,  30,  13,  "north",  100,  20,  50,  75,  50,  20,  20,  75,   75,    50, 100, 0,   0],
    ["residential1", "cityhall",       10,  31,   6,  "south",  20,   20,  50, 100,  50, 250,  75,  50,   20,     0,   0, 0,   0],
    ["residential2", "cityhall",       14,  31,   9,  "east",   100,  20,  20,  50, 100,  50, 250,  75,   50,    20,   0, 0,   0],
    ["residential3", "cityhall",       15,  31,  10,  "west",   100,  20,  20,  50, 100,  50, 250,  75,   50,    20,   0, 0,   0],
    ["residential4", "cityhall",       19,  31,  13,  "north",  100,  20,  20,  50, 100,  50, 250,  75,   50,    20,   0, 0,   0],
    ["residential1", "forestpreserve", 10,   8,   6,  "south",  20,   20,  20,  20,  20,  20,  20,  20,   50,    50,  20, 100, 0],
    ["residential2", "forestpreserve", 14,   8,   9,  "east",   20,   20,  20,  20,  20,  20,  20,  20,   50,    50,  20, 100, 0],
    ["residential3", "forestpreserve", 15,   8,  10,  "west",   20,   20,  20,  20,  20,  20,  20,  20,   50,    50,  20, 100, 0],
    ["residential4", "forestpreserve", 19,   8,  13,  "north",  20,   20,  20,  20,  20,  20,  20,  20,   50,    50,  20, 100, 0],
    ["office1",      "cityhall",       13,  31,   9,  "west",   25,   50, 100, 100, 150,  20, 100,  50,    0,     0,   0,  0,  0],
    ["office2",      "cityhall",       18,  31,  12,  "north",  25,   50, 100, 100, 150,  20, 100,  50,    0,     0,   0,  0,  0],
    ["office3",      "cityhall",       27,  31,  17,  "west",   25,   50, 100, 100, 150,  20, 100,  50,    0,     0,   0,  0,  0],
    ["office4",      "cityhall",       21,  31,  13,  "south",  25,   50, 100, 100, 150,  20, 100,  50,    0,     0,   0,  0,  0],
    ["through1",     "through",         2,  36,  -1,  "east",   100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through2",     "through",        17,  22,  -1,  "north",  100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through3",     "through",        12,   4,  -1,  "south",  100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through4",     "through",        20,  36,  -1,  "south",  100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through5",     "through",         4,  35,  -1,  "east",   100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through6",     "through",         2,  34,  -1,  "east",   100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through7",     "through",        29,  37,  18,  "north",  100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50],
    ["through8",     "office",          2,  18,  12,  "west",   100, 200, 200, 300, 200, 100, 100, 150,  200,   300, 150, 50, 50]]
```

Point to Point Routing Data

Like traffic volumes, directions to cars for how to reach their destinations is encoded in a fixed array.

```
(from neighborhood.py)

# Routes from point to point.  First three numbers are point1, point2, and initial road number.  First direction is the turn-out.
self.route_defs = [
    ["residential1", "office",          10,    13,    6,   "west",    "south",   None,      None,      None,      None,      None ],
    ["residential2", "office",          14,    18,    9,   "south",   "west",    None,      None,      None,      None,      None ],
    ["residential3", "office",          15,    27,    10,  "south",   "south",   None,      None,      None,      None,      None ],
    ["residential1", "playground",      10,    3,     6,   "west",    "west",    None,      None,      None,      None,      None ],
    ["residential2", "playground",      14,    3,     9,   "north",   "west",    None,      None,      None,      None,      None ],
    ["residential3", "playground",      15,    3,     10,  "north",   "west",    "west",    None,      None,      None,      None ],
    ["residential4", "playground",      19,    3,     13,  "west",    "north",   "west",    None,      None,      None,      None ],
    ["residential1", "mall",            10,    30,    6,   "east",    "south",   "south",   "west",    "west",    None,      None ],
    ["residential2", "mall",            14,    30,    9,   "north",   "west",    "south",   "south",   "east",    None,      None ],
    ["residential3", "mall",            15,    30,    10,  "south",   "west",    "south",   "west",    None,      None,      None ],
    ["residential4", "mall",            19,    30,    13,  "east",    "south",   "west",    "west",    None,      None,      None ],
    ["residential1", "cityhall",        10,    31,    6,   "east",    "south",   "south",   "west",    None,      None,      None ],
    ["residential2", "cityhall",        14,    31,    9,   "south",   "south",   "east",    None,      None,      None,      None ],
    ["residential3", "cityhall",        15,    31,    10,  "south",   "south",   "west",    None,      None,      None,      None ],
    ["residential4", "cityhall",        19,    31,    13,  "east",    "south",   "east",    None,      None,      None,      None ],
    ["residential1", "forestpreserve",  10,    8,     6,   "west"     "west",    None,      None,      None,      None,      None ],
    ["residential2", "forestpreserve",  14,    8,     9,   "south",   "west",    "north",   None,      None,      None,      None ],
    ["residential3", "forestpreserve",  15,    8,     10,  "north",   "west",    "west",    "south",   None,      None,      None ],
    ["residential4", "forestpreserve",  19,    8,     13,  "east",    "north",   "west",    "west",    "south",   None,      None ],
    ["office1",      "cityhall",        13,    31,    9,   "south",   "south",   "east",    None,      None,      None,      None ],
    ["office2",      "cityhall",        18,    31,    12,  "east",    "south",   "west",    None,      None,      None,      None ],
    ["office3",      "cityhall",        27,    31,    17,  "south",   "west",    None,      None,      None,      None,      None ],
    ["office4",      "cityhall",        21,    31,    13,  "west",    "south",   "east",    None,      None,      None,      None ],
    ["through1",     "through",         2,     36,    1,   "south",   "east",    "east",    "south",   "south",   "south",   None ],
    ["through2",     "through",         17,    22,    11,  "east",    "east",    "east",    "east",    None,      None,      None ],
    ["through3",     "through",         12,    4,     7,   "west",    "west",    "north",   None,      None,      None,      None ],
    ["through4",     "through",         20,    36,    12,  "east",    "east",    "south",   "south",   None,      None,      None ],
    ["through5",     "through",         4,     35,    2,   "south",   "south",   "south",   "south",   None,      None,      None ],
    ["through6",     "through",         2,     34,    1,   "south",   "south",   "south",   "south",   None,      None,      None ],
    ["through7",     "through",         29,    37,    18,  "east",    "east",    "east",    None,      None,      None,      None ],
    ["through8",     "office",          2,     18,    12,  "south",   "east",    None,      None,      None,      None,      None ]]
```

Video Data for CNN Training

Simplified Snapshots for This Project

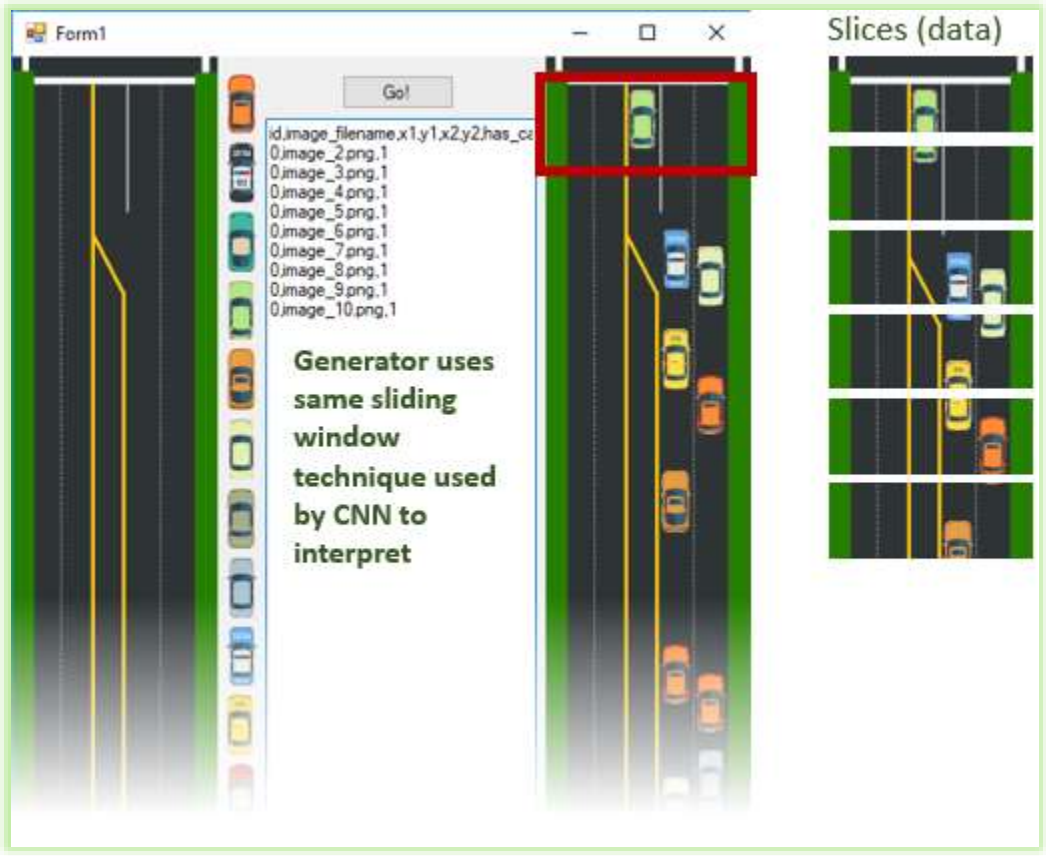
Real video camera data is difficult to interpret and even comprise an entire set of machine learning problems. There are complications such as perspective, occlusion, and other visual artifacts. In addition, any machine analysis must be fast to meet the objectives of its application: must the system process real time video or can it work with periodic snapshots?

For our simulation, video data is generated mechanically by the “environment” based upon positions of the cars. The Neighborhood generates the image data while the Intersection only receives it and must interpret. Video in this image is extremely simple and takes an overhead view which removes complications present in most real-life scenarios.

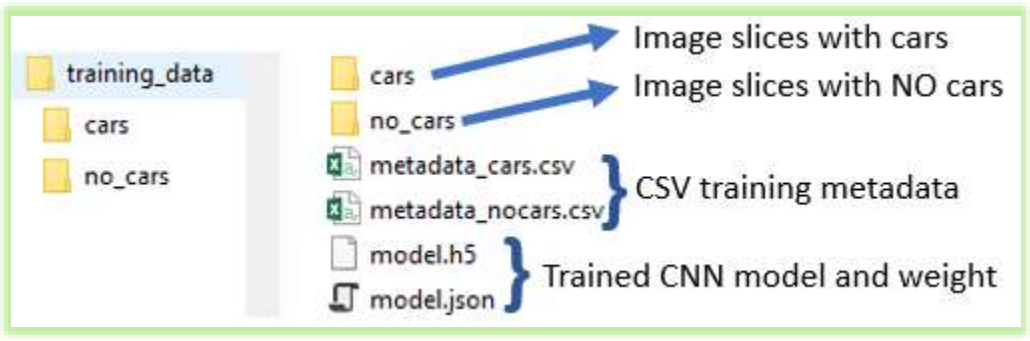
The video is supposedly coming from a camera mounted on a light pole in each oncoming direction. It is not live video but rather snapshots are taken every 30 seconds. The agent will be able to see roughly ¼ mile of oncoming traffic – giving it time to incorporate that information into its decision making.

Actual CNN Training Data Used

Since the CNN is designed as a classifier to interpret the presence of cars within a certain bounding box, the training data was produced mechanically as a random sample of boxes containing cars. A harness was built in C# for this.



There is a corresponding CSV file with each row containing an image name and a binary flag indicating whether it contains any part of a car. Data file layout on disk is the follow:



Code for this harness is included in the project submission but beware, being a simple data generator utility used by this one project and one person, the code is far from professional quality.

ALGORITHMS AND TECHNIQUES

Event Flow

The problem was approached using ideas borrowed from discrete event simulation. The whole thing functions in series, with no concurrency from a programmatic perspective, and hence the performance of the simulation cannot be expected to achieve real-time speed. The actual result was:

- The simulation surprised real-time speed when ML was not used.
- The simulation was slower than real-time when ML was used in series.

In real life these elements would be operating in a more isolated fashion

ML Techniques Leveraged

During the development of the simulation several ML-based approaches were applied in order to derive the best behavior using the simplest code.

- All of the implementations used a CNN to process camera snapshots of oncoming traffic.
- A second neural net, an MLP, was added at one point to further process CNN output.
- The MLP was used in a reinforcement-based approach that compared outcomes (rewards were not immediate).

In the end, the use of a single CNN plus some algorithmic processing proved to be sufficient to achieve the goal, and was also the simplest (and therefore the least prone to defects).

CNN Variations Researched

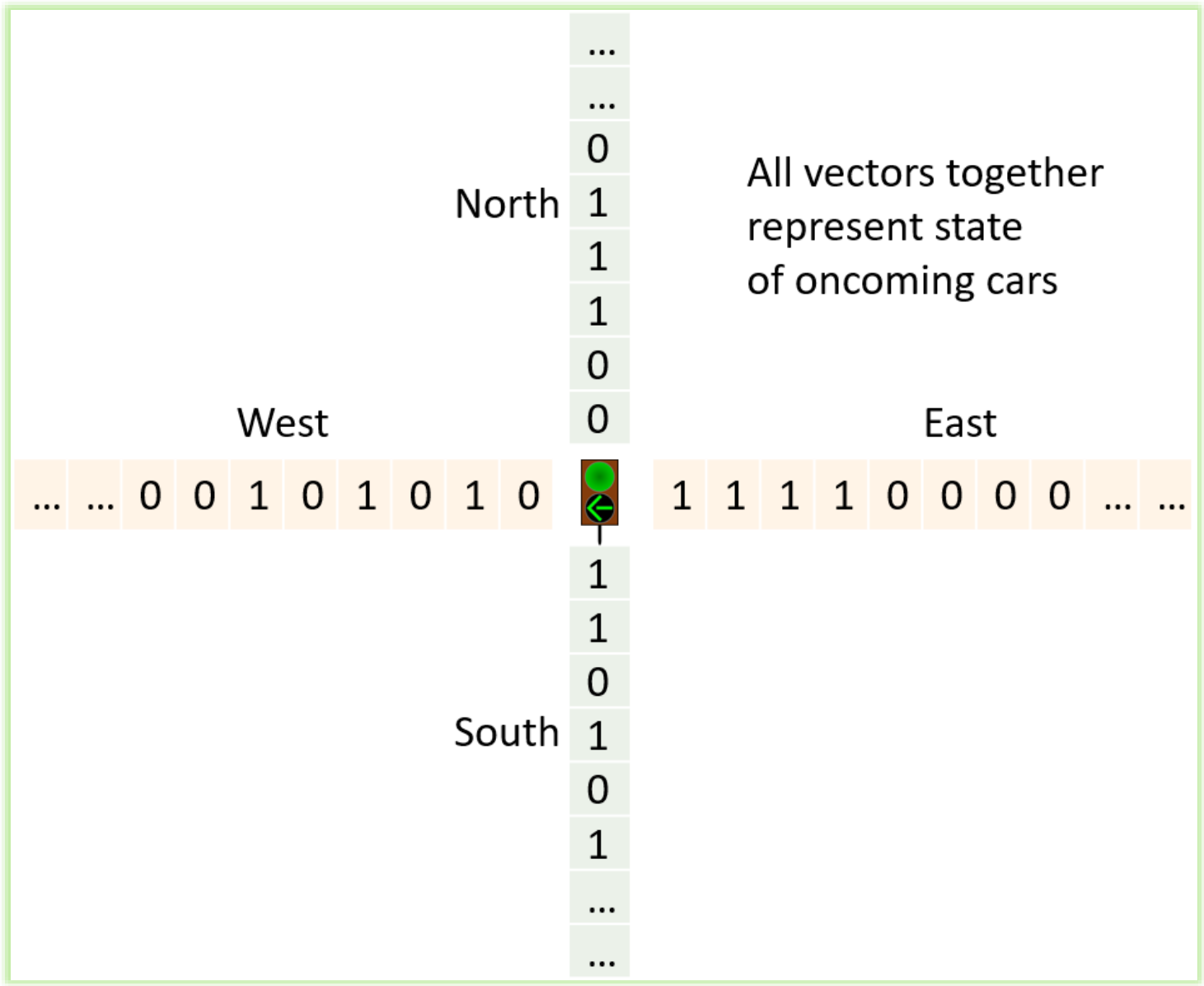
There are a number of ways to count cars in an image and, to achieve this, we looked at several other similar implementations:

- YOLO
- Fast R-CNN
- Several approaches including methods of region proposal

To be brutally honest, I have spent so much time on the project (including research) that I was ultimately forced to use the very simplest of designs.

Reinforcement Learning Approach (Ultimately Removed)

Unlike a simpler reinforcement learning task, this one cannot receive immediate feedback from the prior step. In order to determine whether it made a good decision, it needed to associate a traffic pattern to a *projected* light state when those cars were past the intersection. Using the vector output from the CNN, applied in every oncoming direction at once, that pattern would look like this:



This pattern would be recorded and then the state reassessed after a fixed number of seconds. From this an association was formed regarding whether the "stock" light state given the pattern was okay or if things could be better with a direction change.

This approach was abandoned because:

- The reinforcement delay was arbitrary and required a lot of tuning that might never be settled, and
- It still relied upon an algorithmic "judgement" to compare outcomes.

MLP

At one point during development an MLP was introduced in order to help identify traffic "density maps", as described above, in order to interpret various combinations. This approach works but was ultimately abandoned along with the other technique.

The Chosen CNN Model

The job of the CNN is to determine whether cars are present in the camera snapshot. The code defining the CNN looks like this:

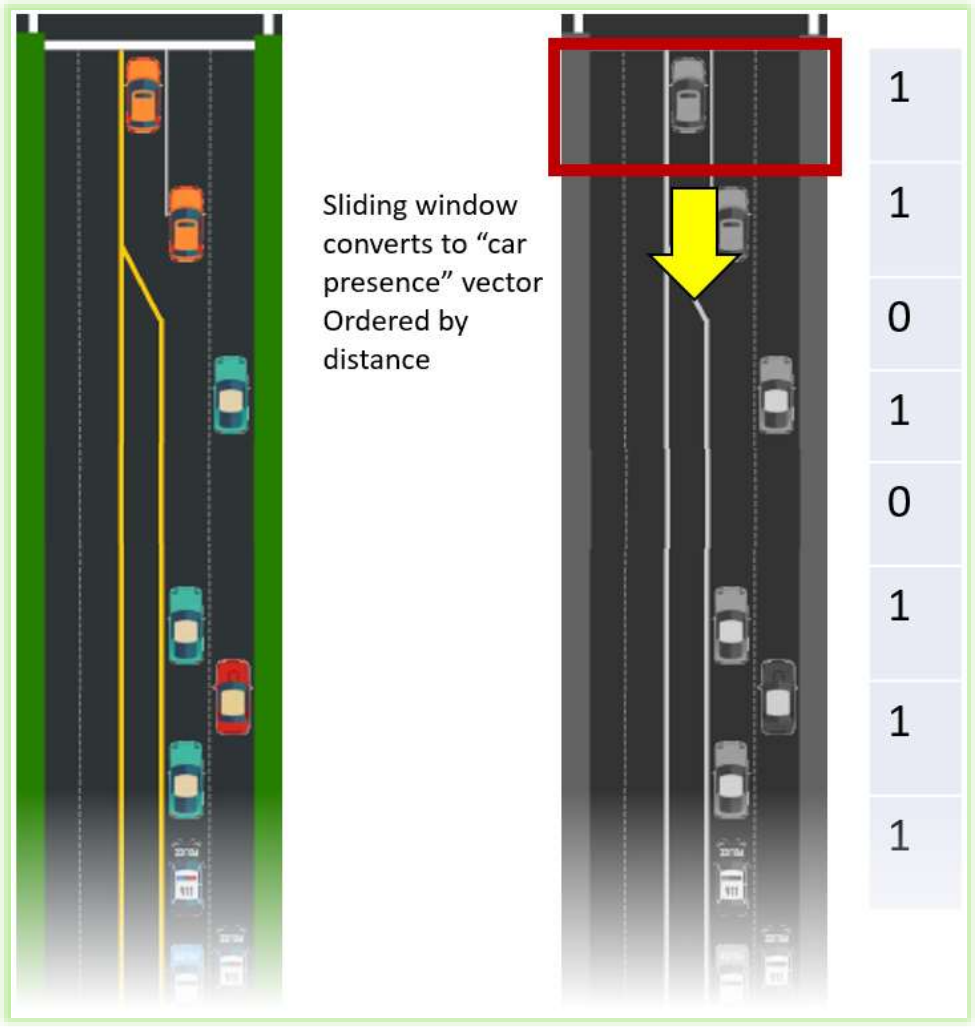
```
# load up the training data
data, labels = myutils.load_training_data(outputPath)

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
imageShape = data[0].shape
num_classes = 1

(trainX, testX, trainY, testY) = train_test_split(data, labels, test_size=0.25, random_state=42)

# grab the image paths and randomly shuffle them
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1), activation='relu', input_shape=imageShape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(32, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dense(32, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Flatten())
model.add(Dense(num_classes, activation='sigmoid'))

print(trainX.shape)
#trainX2 = trainX.reshape(len(trainX), 1, 1)
#print(trainX2.shape)

model.compile(optimizer='rmsprop', loss='mean_squared_error', metrics=['accuracy'])
model.fit(trainX, trainY, batch_size=1, epochs=5, verbose=1)
```

Layers and other hyperparameters were settled on based upon experimentation. It was found that the problem was simple enough that the network could easily handle things even with a correspondingly simple design. In the end the following guidelines were used:

- Input to the CNN had a shape consistent with the “proposed box samples” which were 137 x 50 x 1.
- Activation function ‘ReLU’ was chosen for middle layers as the problem is not linear.
- Activation function ‘sigmoid’ was chosen for the output layer, a single neuron, as single-class binary classification (either cars are present in the sample window or not – number of cars is ignored).

CNN to Vector Translation

In the end, I went with a simple CNN that used a window that slides down the captured image and classifies at key points whether there are cars or not in that bounding box. The CNN was there designed as a simple binary classifier. Either (1) a car is present or (2) is not present. However, the system needs to know more than that in order to be effective.



The resulting vector is very suitable for computing simple “weights” to drive light state changes. As mentioned previous, those decisions were once driven by an reinforcement learn over MLP but it was simplified to be algorithmic because the vector format was so suitable for it.

DEVELOPMENT TECHNOLOGIES

Development Environment

The project was completed using a combination of Python and C# code:

- The simulation and model training itself is written in Python.
- Training data generation for CNN images was done in C# code.
- Graphical debugger was implemented in C#.

The development environment was Visual Studio 2017 using the Python extensions. This environment made it very easy to step through code and evaluate state, similar to how it works with .NET (with which I am most familiar).

The simulation will be use Keras with GPU-based version of TensorFlow as its backend, running on a local Nvidia GTX960 CUDA GPU.

Custom Graphical Debugger

A graphical debugger was created that generates “frames” in real-time representing the state of the simulation during each step. It changes as the simulation runs, even when stepping through running code, or can be used to replay the most recent run.

IMPLEMENTATION

Traffic Generation and Tracking

For each timestep the Environment will generate new traffic based upon the expected frequencies and source/destination pairs. Every car will be tracked individually to keep track of where they are in their travels. Once they reach their destination, the vehicle is released and no longer tacked apart from any statistics that are accumulated from it.

At each time step the cars will “move” and their states are updated accordingly.

“Stock” Intersection Implementation as Reference

A benchmark will be developed by implementing a “stock” behavior that all the intersections (agents) will employee. This default behavior of a light is described here. Assuming that an intersection is at an all-stop condition, and the driver is going North..

```
    def step(self, current_sim_time):

        # Initially start north/south.
        if (self.lanes_active_direction1 == None):

            self.lanes_active_direction1 = self.road_lanes_north_bound
            self.lanes_active_direction2 = self.road_lanes_south_bound

            self.lanes_active_direction1.light_post.light_state_main = "green"
            self.lanes_active_direction2.light_post.light_state_main = "green"

        # How long since we last changed our light state?
        time_since_last_state_change = current_sim_time - self.time_of_last_light_change

        # Turn off any left arrows (set to red) after a short time.
        if (time_since_last_state_change.total_seconds() >= myconstants.timeout_left_turn_signal):
            self.lanes_active_direction1.light_post.light_state_turn_lane = "red"
            self.lanes_active_direction2.light_post.light_state_turn_lane = "red"
            self.lanes_active_direction1.light_post.light_state_main = "green"
            self.lanes_active_direction2.light_post.light_state_main = "green"

        # Maintain current state for a short while (max left-turn time or base for main green).
        if (self.time_to_trigger_light_change(time_since_last_state_change) == False):
            return
        else:
            self.hourly_stats.total_light_state_changes = self.hourly_stats.total_light_state_changes +
1

        # Continue same logic, this time switching to east-west.
        self.lanes_active_direction1.light_post.light_state_main = "red"
        self.lanes_active_direction2.light_post.light_state_main = "red"
        self.lanes_active_direction1.light_post.light_state_turn_lane = "red"
        self.lanes_active_direction2.light_post.light_state_turn_lane = "red"

        if (self.lanes_active_direction1 == self.road_lanes_north_bound):
            self.lanes_active_direction1 = self.road_lanes_east_bound
            self.lanes_active_direction2 = self.road_lanes_west_bound
        else:
            self.lanes_active_direction1 = self.road_lanes_north_bound
            self.lanes_active_direction2 = self.road_lanes_south_bound

        # Northbound turn lane occupied and southbound empty.
        if (self.lanes_active_direction1.are_cars_at_left_turn_trip() == True and
            self.lanes_active_direction2.are_cars_at_left_turn_trip() == False):
```

```
            self.lanes_active_direction1.light_post.light_state_main = "green"
            self.lanes_active_direction1.light_post.light_state_turn_lane = "green"

        # Northbound empty and southbound turn lane occupied.
        elif (self.lanes_active_direction1.are_cars_at_left_turn_trip() == False and
              self.lanes_active_direction2.are_cars_at_left_turn_trip() == True):

            self.lanes_active_direction2.light_post.light_state_main = "green"
            self.lanes_active_direction2.light_post.light_state_turn_lane = "green"

        # Turn lanes empty in both directions.
        elif (self.lanes_active_direction1.are_cars_at_left_turn_trip() == False and
              self.lanes_active_direction2.are_cars_at_left_turn_trip() == False):

            self.lanes_active_direction1.light_post.light_state_main = "green"
            self.lanes_active_direction2.light_post.light_state_main = "green"
            self.lanes_active_direction1.light_post.light_state_turn_lane = "red"
            self.lanes_active_direction2.light_post.light_state_turn_lane = "red"

        # Turn lanes occupied in both directions.
        elif (self.lanes_active_direction1.are_cars_at_left_turn_trip() == True and
              self.lanes_active_direction2.are_cars_at_left_turn_trip() == True):

            self.lanes_active_direction1.light_post.light_state_main = "red"
            self.lanes_active_direction2.light_post.light_state_main = "red"
            self.lanes_active_direction1.light_post.light_state_turn_lane = "green"
            self.lanes_active_direction2.light_post.light_state_turn_lane = "green"

        else:

            self.lanes_active_direction1.light_post.light_state_main = "green"
            self.lanes_active_direction2.light_post.light_state_main = "green"
            self.lanes_active_direction1.light_post.light_state_turn_lane = "red"
            self.lanes_active_direction2.light_post.light_state_turn_lane = "red"

        self.time_of_last_light_change = current_sim_time
        self.daily_stats.light_state_changes = self.daily_stats.light_state_changes + 1

        return
```

In this logic the **`self.time_to_trigger_light_change`** (in **red**) is very central to the logic and its implementation is the driver for the behavior and performance of the intersection. Apart from the nature of that test, all intersections behave the same way in the simulation, simply changing axis's at the right times and then applying standard sensor-based logic.

Refinement

As mentioned previously, different ML techniques were applied before settling on the end state. For each approach the simulation was run and output compared. There were some situations in which comparisons were invalid due to incorrect metrics and those have to be fixed promptly in order to make sound decisions.

In addition, it was necessary to tweak constants to obtain the best results that also consider the limitations of the algorithm in terms of processing power.

```
################################################################################################
# Application constants gathered here for convenience.
################################################################################################

# Location of the training data (for the CNN).
```

```
training_data_path = "C:\\BWHacker2022\\training_data\\"

# Output path for results files and debug frames.
training_working_output_path = "C:\\BWHacker2022\\runtime_output\\"

# Number of "sim seconds" between debug frame generation.  Basically, the debugger resolution.
sim_seconds_between_debugger_frames = 12

# Minimum time a green light should remain green.  We can't just keep switching lights.
minimum_seconds_for_green_light = 60

# Left turn light timeout.
timeout_left_turn_signal = 150

# Green light timeout for the 'stock' intersection implementation.
timeout_green_light_stock = 350

# Number of 'sim seconds' of advancement between each environment 'step'.
sim_timestep_seconds = 3

# Constant allowing us to control the level of traffic at a global level.
traffic_volume_factor = 0.085

# Time ('sim' seconds) between intersection camera snapshots.
seconds_between_camera_snapshots = 15
```

For example, the **`traffic_volume_factor`** value was right-sized to a value that the simulation could handle because the initially defined values generated too much traffic to be practical.

RESULTS

A number of metrics are kept internally but there is one that is the most straightforward measurement: throughput of the intersections. The following table is a comparison of the ML-based performance over the "stock" implementation:

Intersection	Crossings	9am	10am	11am	12pm	1pm	2pm	3pm	4pm	5pm	6pm	7pm	8pm
I1	0%	-1%	7%	-7%	-5%	-2%	6%	0%	-1%	0%	8%	-7%	3%
I2	16%	12%	11%	26%	14%	15%	10%	27%	19%	26%	19%	0%	17%
I3	18%	22%	26%	19%	9%	20%	15%	34%	4%	29%	31%	8%	9%
I4	-1%	-1%	0%	-3%	-3%	-2%	1%	0%	-6%	4%	3%	-5%	-1%
I5	12%	4%	8%	13%	15%	9%	11%	19%	19%	10%	16%	13%	1%
I6	15%	20%	16%	10%	9%	9%	17%	12%	18%	7%	5%	27%	41%
I7	0%	-2%	-1%	-1%	1%	-1%	3%	-2%	-3%	1%	5%	1%	4%
I8	19%	3%	19%	32%	15%	14%	23%	33%	23%	24%	21%	18%	-3%
I9	20%	15%	25%	25%	22%	13%	23%	18%	14%	31%	12%	17%	36%

In all cases the ML intersections either beat the stock ones by a significant amount or performed the same or only slightly worse. The overall picture is that the application of ML had a positive impact.

CONCLUSION

The final conclusion of this report is that application of ML techniques to traffic problems would likely result in considerable efficiency gains. The approach developed here is one of many that are possible, but hardware upgrades of various types would likely be required.

REFLECTION

This was a very challenging!!! Traffic optimization via AI is something I've wondered about for a long time whenever I have been stuck in a long line at an intersection.

I believe that the final implementation could be improved greatly. It uses a bounding box technique but it needs to be run several times just for a single frame. Also, the car counts that the algorithm as a whole produce are accurate enough but still fairly rough. The current implementation results in the following limitations:

- Car counts may be off by as much as 50%! This is because the CNN is a binary classifier applied to the presence of one or more cars and the sliding window covers 2 lanes minimum. This is mitigated by the fact that the same is true for all oncoming directions so a rough estimate is enough.
- The intersection cannot distinguish which lane a car is in. This prevents some types of optimizations that might be made with respect to handling the left turn signal.
- The code is not necessarily efficient and the simulation is slow when the ML agent is running. A "sim hour" under the "stock" implementation takes around 2 minutes in wall clock time, while the same ML implementation takes around 5-10 minutes wall clock time.

Even with these significant issues the results for applying ML to traffic problems are promising.