Capstone Project Report

David Forester

Machine Learning Engineer Nanodegree

29 May 2017

Crossing Busy Intersections Using a Deep-Q Network

DEFINITION

Project Overview

Reinforcement learning (RL) methods have been successfully employed to learn rules, called policies, to solve tasks such as navigate a maze, play poker, auto-pilot an RC helicopter, and even play video games better than humans (https://arxiv.org/abs/1312.5602)[1]. This last feat was accomplished in 2013 by DeepMind who were quickly bought by Google. Their paper "Playing Atari with Deep Reinforcement Learning", describes how a computer can learn to play video games by taking the screen pixels and associated user actions (such as "move left", "move right", "fire laser", etc.) as input and and receiving a reward when the game score increased. Their approach employs a "Deep-Q Network" (DQN) to learn game-winning policies.

This project was inspired by DeepMind's paper and by example programs shared by researchers and enthusiasts such as Eder Santana's Keras Plays Catch (https://gist.github.com/EderSantana/c7222daa328f0e885093)[2]. I began wondering if a pre-trained network might be of use to a self-driving vehicle (sdv) attempting to cross an intersection. In a possible future in which all driving is done by the AI-enhanced vehicles, there would be no need for traffic lights or stop signs, only intelligent policy or coordination amongst the vehicles.

Problem Statement

Let us imagine that a vehicle, which we will call our "agent" is required to traverse a busy

two-lane highway. If it happens to sense that by the time it reaches the intersection a sufficiently large gap in the traffic in both lanes will exist, then it should proceed forward without stopping. If however, there will be no such gap, then it must stop at the intersection and then wait only long enough for a valid gap to present itself before crossing. This is the crossing policy that we seek to learn through deep reinforcement learning.

In this project I sought to design and implement a Python class which could be invoked to train an agent to cross busy intersections by learning a policy through q-learning as is presented in [1] and [2]. Once a policy has been learnt, the same Python class will be used to test the agent using the policy within that environment. Once tested and shown successful, this same policy, saved as weights of our particular neural network, may be used in the future by the agent to cross any such intersection.

Metrics

From the above statement of the problem depend the two indicators of the quality of our learned policy: *safety* and *efficiency*. By "safety" is meant traversing the lanes without crashing into another vehicle. In our simulated environment a crash would occur if any part of our agent and any part of another vehicle occupy the same position in the two-dimensional world in the same time step. By "efficiency" is meant that in attempting to cross the lanes of traffic, the vehicle should take the first-available, valid gap in traffic instead of waiting for a later one to appear.

ANALYSIS

Data Exploration and Visualization

In reinforcement learning, our data is usually the state of the environment. In our case, the environment is a 21x21 world with a vertical road along which our agent moves, and horizontal lanes on which simulated vehicles move. The image below (Figure 1) shows a world with two lanes to traverse. Arrows have been added to indicate the direction of traffic flow. When a grey car has fully emerged from the side of the world and moved one additional time step so that a gap of one unit is behind it and the world border, a variable random number is calculated to determine whether another car will begin emerging on the next time step. This variable may be increased to lengthen the average gap size and decreased to shorten it.

The simulation begins with a randomly generated traffic pattern on both lanes and with

the agent emerging from the bottom border of the world. It ends when the agent has either successfully traversed the intersection or has crashed into another vehicle. Most of the vertical space in the world is for display purposes, to show the vehicle approaching or passing the intersection, though the neural network may learn salient spatial relationships from this otherwise wasted space. The possible actions that the agent may take are only two: *go forward*, and *stay in place*. Until the agent has reaches the position just before the intersection, it has only one valid move, to go forward. At the intersection however, it may either continue to go forward or stay in place as long as necessary before continuing to ensure a safe crossing.

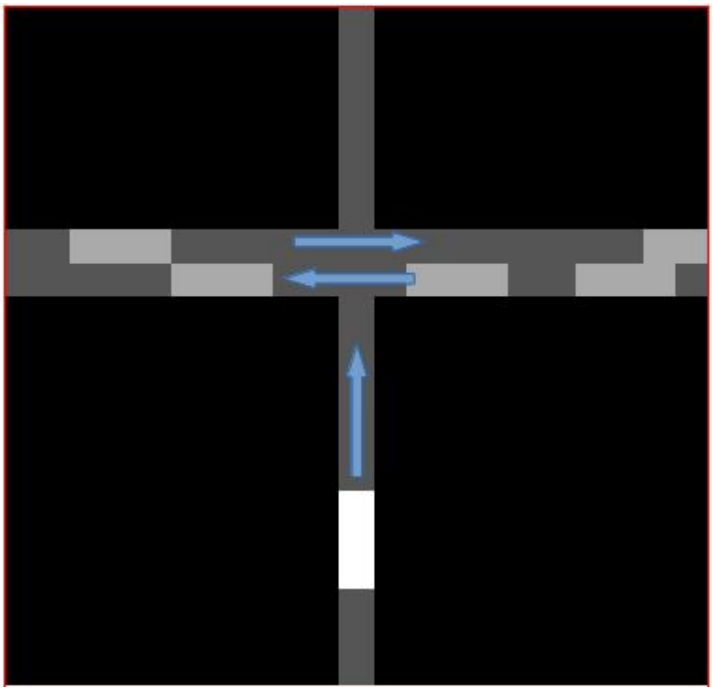


Figure 1. A two-lane 21x21 grid world. The white car on the vertical road must safely and quickly cross the two lanes of traffic on the horizontal road.

Algorithms and Techniques

In Q-learning, one runs many trials of an agent acting/reacting within the environment, some elements of which will exhibit random behaviour, such as position, size, color, etc. The randomness, or stochasticity, of our environment is the placement of vehicles in the lanes and the corresponding gaps between them. Using a deep-Q network, as we do here, means that state of our environment is completely defined by the pixels in the environment

image. A deep neural network is then trained over many trials taking these images as input and the corresponding actions as labels. The output of the neural network is the action policy; there is an output value for each possible action. For our case then, the output of the trained neural network is binary, indicating that, given the input image, the agent should either go forward or stay in place.

To describe the q-learning variant called deep-Q networks, I borrow from the excellent introduction by Tambet Matiisen on the Nervana Systems website (https://www.nervanasys.com/demystifying-deep-reinforcement-learning/)[3].

Quoting the above reference:

In Q-learning we define a function Q(s, a) representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$\mathbf{Q(s_t,a_t) = max(R_{t+1})}$$

The way to think about Q(s, a) is that it is "the best possible score at the end of the game after performing action a in state s". It is called Q-function, because it represents the "quality" of a certain action in a given state.

If the Q-function were known, then to select the action which will result in the best result, we simply pick the action with the highest Q-value. If π represents the policy, then

$$\mathbf{\pi = argmax_a\, Q(s,a)}$$

We can express the Q-value of state *s* and action *a* in terms of the Q-value of the next state *s'* using the Bellman equation:

$$\mathbf{Q(s,a) = r + \gamma\, max_{a'} Q(s,,a,)}$$

We can iteratively approximate the Q-function using the Bellman equation. Using DQNs, we estimate the future reward in each state using Q-learning and approximate the Q-function using a neural network. Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss. The network is trained with a loss function like:

$$\mathbf{L = \frac{1}{2}\,[r + max_{a'} Q(s',a') - Q(s,a)]^2}$$

where $\mathbf{r + max_{a'} Q(s',a')}$ is the target and $\mathbf{Q(s,a)}$ is the prediction. During training of the network, before the best Q-values are learnt, it is important that the sometimes the actions chosen do not come from the policy, but are rather chosen at random. In this way the agent will make new mistakes and get penalized and occasionally take actions for which it will be rewarded. Without this exploration of the action space, the policy would never improve. The variable controlling the frequency of exploration is called ε.

So we estimate the future reward in each state using Q-learning and approximate the Q-function using a neural network. But it turns out that approximation of Q-values using

non-linear functions is not very stable and it can be difficult to make it converge. Quoting again from Matiisen:

> The most important trick is **experience replay**. During gameplay all the experiences < *s, a, r, s'* > are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm.

The author of this report wrote all of the code and designed the simple environment Drive class and methods, however the ExperienceReplay class was taken directly from Eder Santana's Q-learning example code, with only one small modification. His Python code can be found here: https://gist.github.com/EderSantana/c7222daa328f0e885093

Benchmark

Since we define the environment which produces the data for training and testing, how should the results be interpreted? As a lower bound on our benchmark, we could say that the learned model must perform better than random chance. If the model performs better than random chance then we can claim that some learning was achieved. Shouldn't it do much better though? Should it perform better at crossing the simulated intersection than an average human driver crossing an actual busy intersection? The comparison between a human driver and the simulated road is not useful for the following reason: our computer agent moves exactly one unit forward in one discrete time step, even when moving from a stop, whereas a car must often accelerate from a stop, building up speed over continuously flowing time. This could be approximated with a much more sophisticated simulation perhaps, but there are doubtless many other differences which make such a comparison problematic. Let us say then that that this agent should be so trained so that it only rarely fails out of thousands of trials. Just to assign a number, let us say that out of 1000 trial runs, we should observe at most 1 failure.

METHODOLOGY

Data Preprocessing

No data preprocessing was required since the model is trained on the raw image pixels and the accompanying agent action.

Implementation

A version of the environment with two lanes was implemented in three Python files:

- qlearn_crossing.py

  In this file the Drive class is defined and the training iteration loop implemented.

- test_drive.py

  This file is for testing the trained model using methods of the Drive class.

- custom_plots.py

  This file facilitates plotting of the world images for either display or saving to file.

All code was written in Python 3.6 and depends on the following imports: NumPy, SciPy, Matplotlib, Keras, and TensorFlow.

The Training Code -

Pseudocode for training the model as implemented in qlearn_crossing.py is as follows:

```
for e in epochs:
    instantiate environment as env
    while trial not terminated:
        initialize env
        check if agent at intersection
        if at intersection:
            move forward
        else:
            if random int < epsilon:
                explore by choosing an action at random
            else:
                chose the policy model
        experience replay to get previous experiences
        update the model based on experience replay
```

To instantiate the environment means to create a zeros 21x21 uint8 NumPy array. The elements representing road locations are set to value 1. Traffic is randomly generated in the horizontal lanes by repeatedly calculating a random integer to determine whether a car will emerge from the side, and using SciPy's ndimage.interpolation.shift module to propagate any cars along its respective lane. With the environment thus initialized, the agent vehicle moves forward until the intersection is reached. At the intersection the agent must take either the "go forward" or "stay put" action (1 or 0 respectively). Since the q-values represented by the weights of the network are initially randomized and since learning good q-values may take hundreds or thousands of iterations, the agent should not yet choose its action by means of prediction from the model, rather some of its actions should be chosen randomly so that weights have a chance to be updated for all the input elements (the actions) and given as many environment states (configurations of the traffic) as might be encountered in actual use of the model. An epsilon value was used during training, so that an action will be taken at random if the following condition is true np.random.rand() <= epsilon, but that the policy determined by the model (still undergoing training) will be used otherwise like so:

q = model.predict(input)

action = np.argmax(q[0])

where input is the flattened environment NumPy array. This input image array, the action taken, the resultant reward, and the resultant image array are then stored into ExperienceReplay memory. Next a random minibatch from ExperienceReplay memory is selected to make a prediction using this updated model. A training iteration ends either when the agent crosses the intersection or when a crash occurs.

The Testing Code -

Pseudocode for the code to test the trained model, test_drive.py, is:

```
load pre-trained model
for t in trials:
    instantiate environment
    while trial not terminated:
        initialize environment
        check if agent at intersection
        if at intersection:
            use q-values from model to determine action
```

```
else:
     move agent forward
keep count of results used for evaluation metrics
```

Testing begins by loading the model into system memory. The same calls to environment methods used in the training code are used in the testing code for initializing the environment, moving the traffic forward, updating the system state, etc. In testing, of course, model.predict() is always used at the intersection to determine the agent action.

Refinement - two intersections

Final parameters used for training are shown below:

network optimizer	rmsprop
network loss function	mse
epsilon	0.3
replay batch_size	50
training epochs	3000

The final network architecture is shown below, printed by means of a call to keras.model.summary():

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 441)               194922
dense_2 (Dense)              (None, 441)               194922
dropout_1 (Dropout)          (None, 441)               0
dense_3 (Dense)              (None, 441)               194922
dense_4 (Dense)              (None, 2)                 884
=================================================================
Total params: 585,650.0
Trainable params: 585,650.0
Non-trainable params: 0.0
```

I could not begin deciding on an epsilon value until I had identified a neural network architecture and associated optimizer and loss function that would converge during training. I initially defined the model just as in [2]:

```
model = Sequential()
model.add(Dense(hidden_size, input_shape=(grid_size**2,), activation='relu'))
```

```
model.add(Dense(hidden_size, activation='relu'))
model.add(Dense(num_actions))
model.compile(sgd(lr=.2), "mse")
```

When the training did not converge, I re-read the Keras Guide to the Sequential Model (https://keras.io/getting-started/sequential-model-guide) and changed the optimizer from sgd to rmsprop as in the "MLP for binary classification" example. After further unsuccessful attempts, I added a dropout layer, which can also be found in the binary classification example. The final epsilon value of 0.3 was chosen by beginning at 0.1 and training over a few hundred epochs, watching whether the loss value decreased sufficiently. Convergence alone however is not proof of a valid model. So I would then test the model for a few hundred iterations, observing the "Success Rate" metric to determine whether to try another epsilon value.

Multiple lanes -

It was found that a model trained for the two-lane case could be used in an environment with multiple two-lane intersections as long as they are separated by at least one car length (since the agent may need to come to a stop). In the same way, if the environment contains both a single lane and a two-lane intersection, each lane may be safely traversed by using the appropriate pre-trained model. The double, two-lane case will be discussed below in the Results section.

RESULTS

Model Evaluation and Validation

The two-lane intersection model was trained on an environment with average gap length variable equal to 6 (or 2 x car_length). This means that for each move forward of a car on the horizontal road, the chance of the gap length growing by another unit in length is 1-in-6. This value was chosen in an attempt to minimize training time by presenting enough crossing opportunities while keeping enough cars on the road at any given time. Sometimes during training it would be necessary for the agent to continue driving through the intersection to gain a positive reward. Sometimes the agent would need to crash in the intersection, in either lane, to earn the negative rewards. Sometimes the agent would need to stay put during one or more time steps before continuing across to gain the reward. It

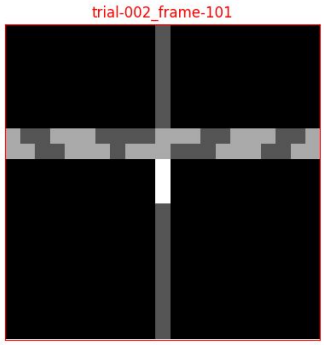
was assumed (hoped) that when the model was fully trained that it would be just as accurate no matter the gap length setting used during testing.

The reward values used during training are shown below in the Drive class _get_reward() internal method:

```
47
48      def _get_reward(self):
49          if self.crossed:
50              reward = 5
51          elif self.crashed:
52              reward = -10
53          else:
54              reward = 0
55          return reward
56
```

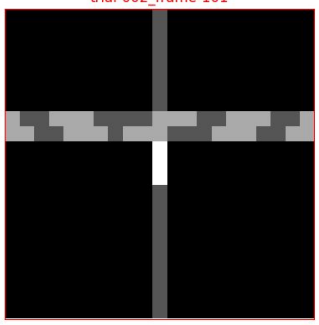
Training was initially attempted with line 54 in the code snippet above replaced by reward = -1, to “encourage” the agent to finish in as few time steps as possible. However, it was observed to have a negative impact on convergence and was changed to the value seen above. The agent using the trained model only rarely misses a crossing opportunity.

Below is six frames from a testing sequence using the trained model. The agent identifies a gap in both lanes only just large enough to pass through.

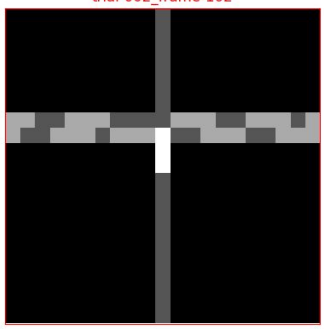


Agent has stopped at intersection

trial-002_frame-101

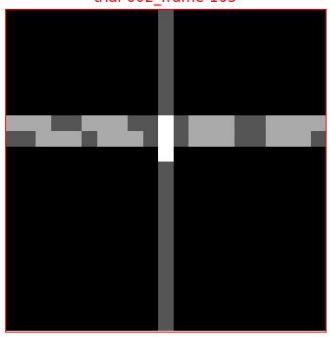


trial-002_frame-102



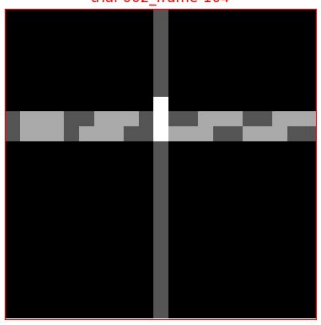
Agent moved forward one step as soon as gap in first land appeared

trial-002_frame-103

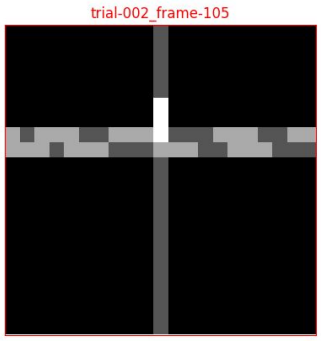


Next step

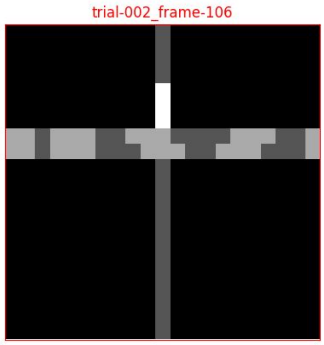
trial-002_frame-104



A car in first lane moves adjacent to agent



Agent moves forward just before a collision with the approaching car in the first lane



Agent moves out of second lane just before collision with approaching car

Besides determining if a model could be learnt to enable the agent to cross intersections safely and efficiently, it was thought to be of interest to measure how well this model performs when used for environments with gap length variables different from that on which the model was trained. Results are shown in the table below. Recall that the model was trained with the average gap length variable set to 6. Setting this variable to 1 always produces gaps of length 1 in both lanes (no more randomness) so that the agent can never traverse and always chooses to stay put, so 2 is the smallest possible value.

Single, two-lane intersection		
avg. gap length variable	Missed opportunities (%)	Success Rate (%)
2	4.73	92.70
3	0.99	98.49
4	0.36	99.54
5	0.17	99.74
6	0.13	99.92
7	0.09	99.97
8	0.06	100.00
9	0.06	99.99
10	0.04	100.00
11	0.04	100.00

These results were encouraging. As long this trained model is used in an environment with

the average gap length variable set above 2, the agent is able to traverse the environment safely and efficiently. What if the agent encountered an environment with multiple 2-lane intersections? Could this same model be applied to each intersection independently so that we could avoid training on every unique environment we could imagine? It certainly seems that we could convince the agent, with a little world map manipulation, that when it had reached one of these intersections, it was viewing the same kind of environment on which its model had been trained.

Take, for instance, the double intersection environment below. When the agent reaches, say, the bottom intersection, we could make a copy of the environment in which the second intersection was missing. Then, by translating the single, remaining intersection to the same location in the grid as the environment on which the model was trained, and placing the agent just below it, it would make no difference to the model that this scene was from a double intersection environment.

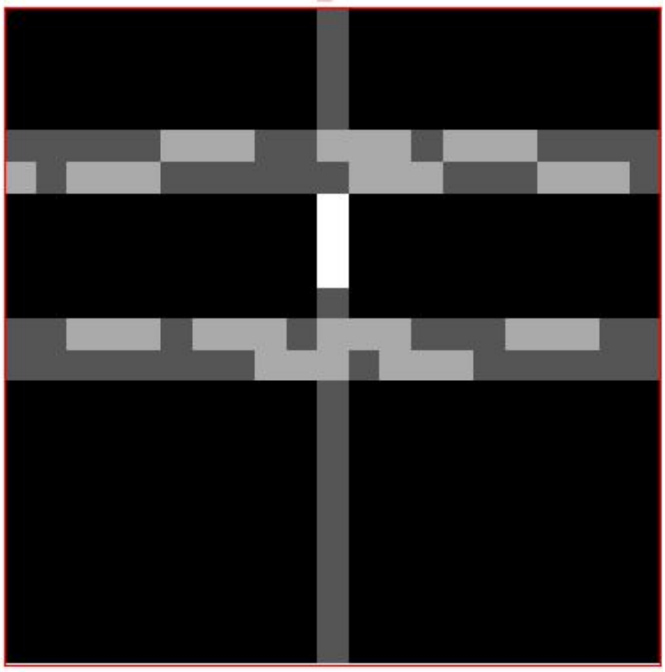


Figure 2. The agent, having crossed the first intersection, waits to cross the second (avg. gap length variable = 4).

I had assumed that though the second intersection had been added, by treating it separately as described above, this would be equivalent to doubling the number of trials, but that the missed opportunities and success rate percentages would be unchanged. I was

therefore surprised to see that not only the missed opportunity had changed nonlinearly with gap length variable, but so had the success rate. See table below.

Double, two-lane intersection		
avg. gap length variable	Missed opportunities (%)	Success Rate (%)
2	6.92	84.39
3	2.11	96.52
4	1.26	99.05
5	0.88	99.66
6	0.66	99.88
7	0.60	99.92
8	0.45	99.97
9	0.36	99.96
10	0.43	100.00
11	0.29	100.00

Though the performance in the double intersection environment was appreciably different for low gap length variables, most of the tests yielded similar, though still slightly lower, results.

Justification

With missed crossing opportunities near zero and successful crossings near 100 percent, it seems certain that our model performs better than the random policy benchmark described earlier, but performing such a check is easy enough. Taking the single 2-lane intersection environment with gap length variable set to 6, and changing test_drive.py to use a random policy at the intersection instead of the learned policy yields the following:

Missed Opportunities: 6.14 %, Success Rate: 21.30 %

Comparing this to the corresponding entry in the green results table above:

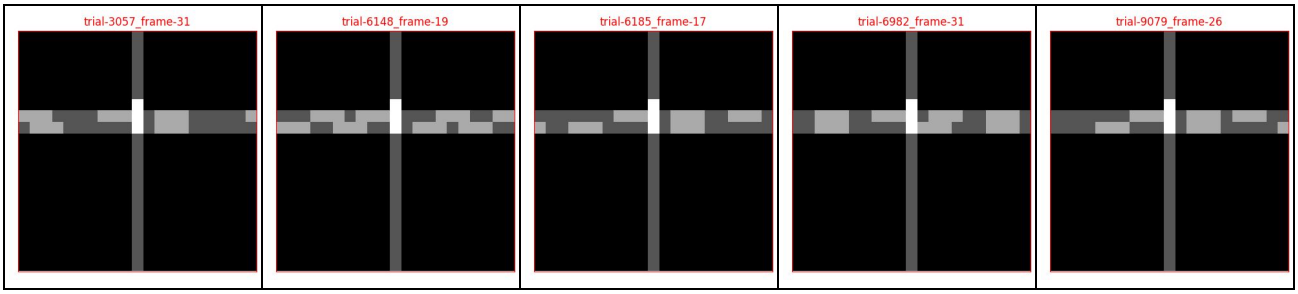
Missed Opportunities: 0.13 %, Success Rate: 99.92 %,

we may justly say that a good crossing policy was learnt.

CONCLUSION

Free-Form Visualization

To see better when, if not why, the model fails some few times, I ran test_drive.py on the single, 2-lane intersection case for 10,000 iterations, saving the environment image to disk if a crash occurred. Five crashes were recorded and shown below.



Every crash involved a car in the upper lane, and with the agent one unit past the intersection. To see if either of these observations might be a rule rather than mere chance, I ran the script again, this time using 50,000 trials. Twenty-nine crashes were recorded. All but two of these crashes were like the five seen previously, with the agent one unit past the intersection, having collided with a car in the upper lane. This seems to indicate a weakness in the model, but I do not know at present how to correct it.

Reflection

To summarize my solution to the problem of learning a successful policy to cross a busy intersection, I have done the following:

- Created an environment class “Drive” which defines and implements the intersection and traffic features of the world in which our agent must act.
- Q-learning is implemented by an ExperienceReplay class which stores and retrieves batches of previous states, actions, and rewards.
- The methods of these classes are used first in qlearn_crossing.py to train the model and then in test_drive.py to test this trained model and to gauge model accuracy.

All code for this project can be found in the Github repository https://github.com/drforester/Q-learning-to-cross-intersection.

Though this environment was a simple one, and though the actions available to the agent were limited to two, this project does show that q-learning can enable an agent to make nearly perfect decisions within that environment. It might be tempting to dismiss the intersection examples discussed herein as too simplistic for application to real-world decision-making, but there are many real-world cases where the environment can be

downcast and approximated by such a simplistic environment. Imagine that you want to simulate an actual intersection as accurately as possible. Most of the information would be contained in such a simplistic environment as we have been using here. For greater accuracy we could of course increase the definition of our world by representing it with many more pixels and much shorter time-steps. We could also include such real-world constraints as acceleration so that the agent must gradually come up to speed as it moves forward from a stop at the intersection. However I believe that these would be but refinements on the policy encapsulated by the model we have already developed.

Improvement

The binary regression neural network used here began to work well after 50% dropout was added in one of the dense layers. Would a convolutional neural network, such as described in reference 1, yield an even more accurate model?

Some little testing was performed with a decaying epsilon, so that the agent would be more likely to explore, i.e. use a random action, at the beginning of training and less likely as more training iterations were performed and better q-values were learnt. However, not enough work was done in this area to judge the effectiveness. Use of a decaying epsilon should result in a fewer required training epochs.