

Machine Learning Nanodegree Capstone Project: Training a Smartcab Planner

Lucas Murtinho
lucas.murtinho@gmail.com

August 7, 2016

Contents

1	Definition	2
1.1	Project Overview	2
1.2	Problem Statement	2
1.3	Metrics	2
2	Analysis	3
2.1	Input Space Description	3
2.1.1	Position and Heading	3
2.1.2	The PlannerWorld Class	4
2.2	Exploratory Visualization	5
2.3	Algorithms and Techniques	5
2.4	Benchmark	7
3	Methodology	9
3.1	Data Preprocessing	9
3.2	Implementation	9
3.2.1	Basic Planner	9
3.2.2	Operating Planners	10
3.3	Refinement	12
4	Results	12
5	Conclusion	13
5.1	Free-Form Visualization	13
5.2	Reflection	15
5.3	Improvement	15

1 Definition

1.1 Project Overview

In Project 4 of Udacity’s Machine Learning Engineer Nanodegree, I had to teach a smartcab in a grid-like world how to get to a destination on time by obeying traffic rules and following the directions passed by a planner. Without any “hardcode”, the agent should learn right-of-way rules and to go straight when the planner sent a “forward” input, for instance.

A natural extension of this project is, instead of relying on an outside planner, coming up with an agent that, given the smartcab’s location and destination, learns how to plan a route. This is the goal of my capstone project.

1.2 Problem Statement

The problem at hand is to program an agent that learns to identify what the next waypoint should be for a smartcab in a grid-like world to reach its destination as fast as possible. I’ll use an 8×6 grid for the project, similar to the one used for the Nanodegree Project 4, but in principle the solution should apply to a grid of any size.

It is possible to solve this problem through “hardcoding” — i.e., by developing a function that takes as inputs the smartcab’s location, heading, and destination, and returns the next move the agent should perform. This is, in fact, how the provided planner from the Nanodegree’s project 4 works, and it is also how I programmed the “perfect planner” described in the Benchmark section below. However, here I am interested in accomplish the same task through reinforcement learning, or, more specifically, Q -learning (see the Algorithms and Techniques section below for an explanation of this family of algorithms).

In this setting, the program is an agent that operates in a world defined by the `PlannerWorld` class. Upon performing an *action* in a given *state* of the world, the agent receives from the environment a *reward*, which is used to modify the agent’s *policy*, or the function that returns an action given a state.

The goal of this project is to implement a learning agent such as the one described above that can learn a feasible policy over time. An additional is to implement through reinforcement learning not only a feasible policy but an optimal one — that is, to create a learning agent whose behavior, after some exploration time, is indistinguishable from the perfect planner’s.

1.3 Metrics

I used the following metrics to gauge the performance of the learning planner:

- The number of **destinations reached** over all trials. I defined a deadline of 100 steps for the agent to reach its destinations, which is more than enough steps in an 8×6 toroidal grid. Therefore, reaching the destination within the deadline after a number of exploratory steps are taken is a very low bar for any learning agent to clear.
- The **number of steps needed** to reach the destination. This is a more demanding metric: a perfect planner, for instance, will need no more than 7 steps to reach its destination in a grid world such as the one I devised.¹ However, the minimum number of steps at any given trial will vary according to the agent’s initial position relative to its destination.

¹Since the agent can “go around the world”, it will need at most $(m + n)/2$ steps to reach any destination.

- The **sum of rewards** over each trial. The reward structure of the world is defined in the `PlannerWorld` class and it must be well defined for the agent to learn the desired policy. Accumulating rewards is the ultimate goal of the agent; if it can get a large sum of rewards without achieving an optimal policy, either the learning algorithm or the reward structure should be reexamined.
- The number of trials with **net positive rewards**. Again, the main goal here is to verify whether the reward structure is consistent with the agent's desired behavior.
- The number of **states visited** by the agent. As full an exploration of the state space is desirable, since this means the agent has acquired sufficient knowledge of the world for it to operate in the desired fashion. Once again, if the agent has explored much of the world but its policy does not approximate the perfect planner's, the learning algorithm or the structure reward must be reviewed.

I computed these metrics over 2000 trials for each of the planners presented in this report.

2 Analysis

2.1 Input Space Description

In Project 4 of Udacity's Machine Learning nanodegree, the world in which the smartcab existed was represented by a graphical 8×6 grid rendered in Pygame. I'll use an image of that world in the next section, but for now I'll describe the problem's input space abstractly.

2.1.1 Position and Heading

In the grid-like world of the smartcab, each position is defined by an (i, j) tuple, in which i is the longitude (the position across the East-West axis) and j is the latitude (the position across the North-South axis). In a $m \times n$ grid, $(1, 1)$ represents the northwesternmost position, while (m, n) represents the southeasternmost position.

The goal, then, is that, given a position tuple (i_{cab}, j_{cab}) , a destination tuple (i_{dest}, j_{dest}) , and a heading (described below), the planner should be able to come up with the best next action for the smartcab: forward, right, or left.

The results of an action taken by the smartcab depends on its heading. The heading is a tuple (i_{head}, j_{head}) , whose items represent East-West heading and North-South heading, respectively. The heading tuple must obey the following rules to be valid:

1. The value of a tuple element indicates how the smartcab will move along the element's axis if it moves forward. This value can be either -1, 1, or 0.
2. If the first element of the tuple is non-zero, the second element is zero and vice-versa. (This means the smartcab can only be headed in one direction at a time.)

These rules imply four valid headings: $\{(1, 0), (-1, 0), (0, 1), (0, -1)\}$. These headings indicate the smartcab is turned East, West, South, and North, respectively.

Therefore, if the smartcab is at position (i_{cab}, j_{cab}) and moves forward, at the next step it will be at position $(i_{cab} + i_{head}, j_{cab} + j_{head})$ - with an important exception: the grid world is *toroidal*,

which means going "over the border" will bring the smartcab to the other side. If the smartcab is headed East at the eastnorthernmost position $(8, 1)$ in an 8×6 grid, for instance, moving forward will bring it to the westnorthernmost position $(1, 1)$.

The more general formula for the smartcab's position at the next step after moving forward (considering, as stated above, that $(1, 1)$ represents the northwesternmost position in an $m \times n$ grid) is therefore:

$$((i_{cab} + i_{head} - 1) \% m + 1, (j_{cab} + j_{head} - 1) \% n + 1) \quad (1)$$

The above expression is used to update the smartcab's location in the current version of the environment class used for project 4 (see here).

2.1.2 The PlannerWorld Class

To define the input space in which the planner is to operate, I created the `PlannerWorld` class, with the following attributes and methods:

- `grid_size`: a tuple defining the size of the grid (East-West and North-South).
- `trials`: the number of trials to run.
- `deadline`: the number of steps an agent is allowed to take in each trial. If the agent takes `deadline` steps without reaching its destination, the trial is terminated.
- `get_delta()`: given the agent's location and its destination, this method returns a tuple with the shortest East-West and North-South distances between these two points. If the shortest distance is going East or South, the corresponding element in the returned tuple is positive; if the shortest distance is going West or North, the element is negative. (In other words, `delta` has the same sign as the heading the agent should be in order to take the shortest path to its destination.)
- `get_distance()`: given a `delta` tuple, this method returns the Manhattan distance between the agent's location and its destination. It will be used to compute the reward the agent will receive at each step of a trial.
- `get_reward()`: this method returns a reward according to the state the agent is in. If the agent has reached its destination, the reward is the number of steps the agent still could take before the termination of the trial (or `deadline` minus the number of steps taken in the trial). If the agent has not reached its destination, the reward is the negative of the Manhattan distance between the agent's location and its destination.
- `move_agent()`: this method will update the agent's position and heading according to the action taken by the agent and to the agent's heading when said action was performed. For instance, if the agent is at position $(1, 1)$, facing North, and goes right, its next position will be $(2, 1)$ and its next heading will be to the East. The method takes into account the toroidal nature of the world and allows the agent to "go across" the world's boundaries.
- `get_random_position()`: this method returns a random position in the world. For an $m \times n$ world, the northeasternmost position will be $(1, 1)$ and the southwesternmost position will be (m, n) .

- `simulate()`: this method simulates the agent's world. For as many trials as defined in the world's `trial` attribute, the method performs the following:
 - The agent's location and its destination are randomly initialized using `get_random_position()`, and the agent's heading is also randomly defined
 - The sum of rewards for the trial is defined as 0
 - For each step in the trial:
 - * The delta tuple is calculated using `get_delta()`
 - * The state is defined as a tuple containing the delta tuple and the agent's heading
 - * The state is passed to the planner, which uses its `policy()` method (described below) to return an action
 - * The action picked by the agent is passed to `move_agent()`, which updates the agent's position and heading
 - * The new delta between the agent and its destination is calculated, and the new state in which the agent is in is defined as a tuple with the updated delta and heading
 - * The `get_reward()` method calculates the reward associated to the agent's new state
 - * An experience tuple with the previous state, the action performed, the reward obtained and the new state is passed to the planner's `update_qval()` method (explained below), which updates the Q -value associated to taking the action performed at the previous state the agent was in
 - * The reward perceived is added to the overall sum of rewards
 - * If the agent's location is equal to the destination, the trial is terminated
 - The number of steps taken by the agent during the trial and the overall sum of rewards it received are appended to a `trial_results` list

The function returns a dataframe with the results of each trial and the planner's final Q -table.

2.2 Exploratory Visualization

Figure 1 shows the grid world of Project 4 without the lights and the other ("dummy") agents. This is the world in which the planner must find the best route toward the destination.

2.3 Algorithms and Techniques

Q-learning description

For the learning planner of this project, I'll be using the Q -learning family of algorithms described in the Reinforcement Learning section of Udacity's Machine Learning Engineer nanodegree. A Q -learning algorithm allows an agent to learn an optimal policy from sets of states, actions, next states, and rewards, without being explicitly told which actions are optimal in which circumstances.

The general form of the Q -function is:

$$Q(s, a) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a') \quad (2)$$

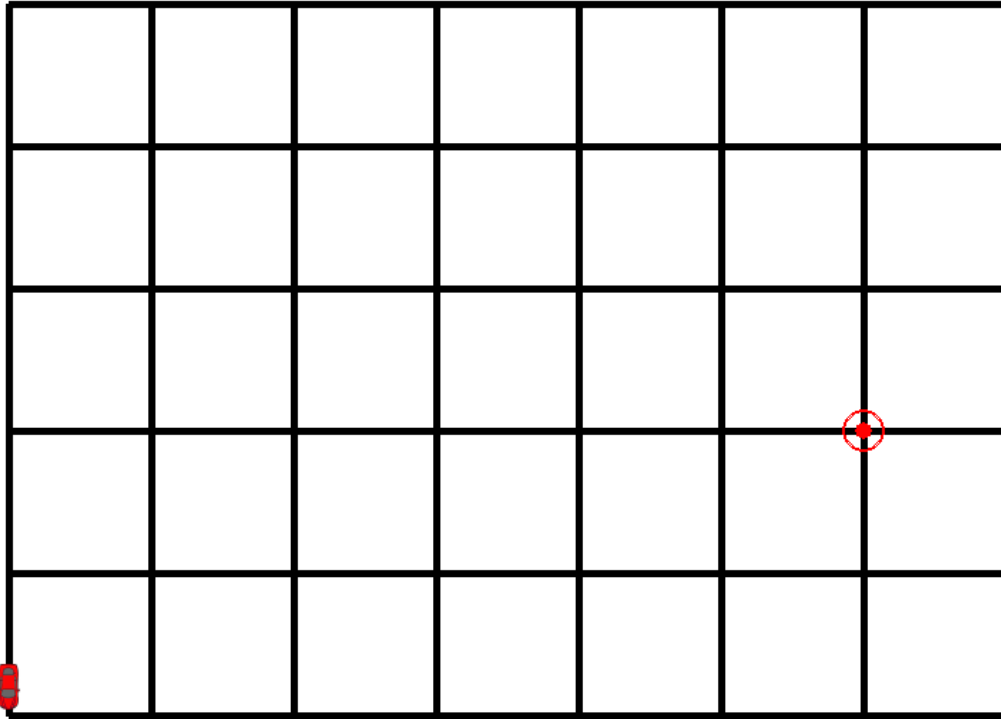


Figure 1: The planner's world. The smartcab is at position (1,6), facing South, and a forward move would therefore take it to position (1,1). The destination is at position (7,4). Due to the toroidal nature of the world, the smartcab's next move should be right, taking it to position (8,6) and 3 moves away from its destination.

That is, the Q -value for a given $(state, action)$ pair is the the reward for that state, $R(s)$, plus the discounted (by the discount factor γ) expected value of Q for the next state the agent lands in, considering the transition function $T(s, a, s') = Pr(s'|s, a)$ (the probability of landing on state s' coming from state s and performing action a) and assuming that, whatever s' is, the agent will pick a' so as to maximize Q from there on.

The Q -learning update function is given by:

$$\hat{Q}_t(s, a) = (1 - \alpha_t)\hat{Q}_{t-1}(s, a) + \alpha_t(r + \gamma \max_{a'} \hat{Q}_{t-1}(s', a')) \quad (3)$$

That is, our estimate of the Q -value for the $(state, action)$ pair is updated with the learning rate (α_t , which varies over time) by the observed reward (r) and our previous estimate of the Q -value for the observed next state (s'), discounted by γ and assuming the agent will pick the action a' that maximizes Q from the next state on.

To implement a Q -learning algorithm we need to define the following:

- **A discount factor**, γ . This will define how future rewards are to be valued in comparison to present rewards.
- **A learning rate**, α , and its decay rule. This defines how quickly the agent updates the Q -value of each $(state, action)$ pair as it explores the state space.
- **Initial Q -values** for the $(state, action)$ pairs. A common strategy to promote exploration by the agent is *optimism in the face of uncertainty*, initializing high Q -values that will decrease as the agent explores the world. If the agent always picks the action that yields the largest Q -value (see below), this guarantees the agent will be compelled to pick previously unperformed actions.
- **A policy**, or a way for the agent to pick actions based on the updated Q -function. Most of the times, the agent should pick the action that yields the largest Q -values, thus *exploiting* the knowledge of the state space previously acquired. However, it is also important for the agent to *explore* the state space in order to acquire this knowledge. A compromise may be found by defining a **decaying exploration rate**, ϵ , and setting the agent to perform a previously unperformed action (typically chosen at random) with probability ϵ_t .

The implementation of the Q -learning algorithm in this setting will be presented in the Implementation section.

2.4 Benchmark

Before implementing the learning planner, I implemented two other planners, whose results will be compared to the learning planner's:

- A **random planner** that selects an action at random at each step.
- A **perfect planner** that always selects the best possible action, defined as an action that reduces the distance between the agent and the destination. (Depending on the agent's position, more than one action can be the "best" under this definition: for instance, if the shortest path to the destination involves going forward and taking a right turn, both actions would be considered as the "best" one.)

See the implementation of these agents, as well as the learning agent, in the Implementation section.

The tables below summarize these two planners' performance in a simulation with 2000 trials.

Random Planner Results

Metric	Sum of rewards	Steps taken
Mean	-99.2	47.9
Standard deviation	160.9	35.1
Minimum	-460.0	1.0
Q1	-216.0	17.0
Median	- 59.5	39.5
Q3	40.0	84.0
Maximum	100.0	100.0

Destinations reached: 1752/2000

Trials with positive sum of rewards: 741/2000

States visited: 744

The random planner reached its destination in approximately 87.5% of the trials - a good result achieved due to the generous deadline defined: in over 75% of the time, the random planner needs more than 17 steps to reach the destination, which is more than double the worst case for the perfect planner (see below). Only 37% of the trials end with a positive sum of rewards for the random planner.

Perfect Planner Results

Metric	Sum of rewards	Steps taken
Mean	91.2	3.7
Standard deviation	6.0	1.4
Minimum	73.0	1.0
Q1	86.0	3.0
Median	91.0	4.0
Q3	95.0	5.0
Maximum	100.0	7.0

Destinations reached: 2000/2000

Trials with positive sum of rewards: 2000/2000

States visited: 244

By contrast, the perfect planner always reaches its destination and always nets a positive sum of rewards. As mentioned in the Metrics section, it takes at most 7 steps to reach its destination, which makes sense considering the dimensions of the state and its toroidal nature.

One interesting difference between these planners is the number of states visited, much larger for the random planner. This makes sense because, with a perfect planner, there is no exploitation: the

correct action will always be taken, so any state involving a wrong action (i.e., an action that leads to an increase of the distance between the agent and its destination) is automatically discarded. This amounts to a significant reduction of the state space the agent will experience, but it does not detract in any way from the agent’s performance.

3 Methodology

3.1 Data Preprocessing

There is no data processing in this project, since the input space is created “from scratch”, based on the code for project 4.

The only step that may be considered as data processing is the function `get_distance()`, which returns the absolute sum of a tuple of North-South and East-West distances. This is necessary to calculate the planner’s reward at states in which the agent has not reached its destination, which is defined as the negative of the Manhattan distance between the agent and the destination (see below).

3.2 Implementation

Each agent was implemented as a subclass of a basic `Planner` class. This way, attributes and methods that are common to all planners were implemented only once.

I decided on this implementation after verifying that treating each planner as independent classes or functions would lead to code repetition and potential mistakes. Other than that, no complications arose during the implementation of the project.

3.2.1 Basic Planner

The basic planner is defined in the `Planner` class, with the following attributes and methods:

- `initial_val`: the initial Q -value for a previously unknown $(state, action)$ pair. The default value for this attribute is 0.
- `disc_rate`: the rate used to discount future rewards. The default value is 0.5.
- `learn_rate_func`: a function that takes the number of steps already taken by the agent as an argument and returns the learning rate to be used in the next update of the Q -table. Its default is $1.0/(x + 1)$, where x is the number of steps taken by the agent.
- `q_vals`: the Q -table, initialized as an empty dictionary.
- `possible_actions`: a list of the possible actions the agent can perform: `forward`, `left`, or `right`.

- `update_qval()`: a method that updates the agent's Q -table:

```
def update_qval(self, experience, t):
    """
    Returns an updated version of the dict
    with Q-values for (state, action) pairs.
    """
    # learning rate decreases over time
    # according to a predefined function
    learn_rate = self.learn_rate_func(t)

    state, action, reward, next_state = experience
    # get the current Q-value for the (state, action) pair
    current_qval = self.qvals.get((state, action), 0)
    # get the best action for the next state
    # given the information in the qvals dict
    next_action = self.policy(next_state)
    # update the Q-value for the (state, action) pair
    update_qval = (1-learn_rate) * current_qval + \
        learn_rate * (reward + self.disc_rate * \
            self.qvals.get((next_state, next_action),
                           self.initial_val))
    self.qvals[(state, action)] = update_qval
```

This function could have been implemented only for the learning planner; however, implementing it in the basic planner guarantees that we can check how many states every planner visited.

3.2.2 Operating Planners

The basic planner described above lacks a `policy()` method - that is, a method to define the action to be performed given a state. I describe below the policy that defines each type of planner: the perfect and random planners whose results were presented in the Benchmark section, and the learning planner whose results will be presented in the Results section.

- **Random planner:** The policy for this planner consists of randomly picking one of the three possible actions:

```
def policy(self, state):
    return random.choice(self.possible_actions)
```

- **Perfect planner:** The policy consists of always picking an action that reduces the Manhattan distance between the agent and its destination:

```

def policy(self, state):
    """
    Returns the best action given the agent's
    position relative to its destination as well as
    its heading.
    """
    delta, heading = state
    # if agent is turned to the east/west axis
    if heading[0]:
        # if it needs to go forward, do it
        if delta[0] * heading[0] > 0:
            return 'forward'
        # else check if it needs to go right or left
        elif delta[1] * heading[0] > 0:
            return 'right'
        else:
            return 'left'
    # if agent is turned to the north/south axis
    else:
        # if it needs to go forward, do it
        if delta[1] * heading[1] > 0:
            return 'forward'
        # else check if it needs to go right or left
        elif delta[0] * heading[1] < 0:
            return 'right'
        else:
            return 'left'

```

- **Learning planner:** The policy consists of picking one of the best actions according to the Q -table. Given a state s , the learning planner checks the Q -value of all $(state, action)$ pairs for which the state is s , and picks the action that maximizes this value, breaking ties randomly.

```

def policy(self, state):
    # get the value of each action given the
    # current state
    possible_vals = {action: self.qvals.get((state, action),
                                             self.initial_val)
                     for action in self.possible_actions}

    # find the actions that yield the largest value
    best_actions = [action for action in possible_vals.keys()
                     if possible_vals[action] == \
                        max(possible_vals.values())]

    # randomly pick one of the best actions
    return random.choice(best_actions)

```

3.3 Refinement

The refinement of the model involved adjusting the parameters of the Q -update function. After a learner with the default parameters presented above was tested, the learning function rate was modified to return $1.0/\ln(x + 1)$. The results of both models are presented in the Results section.

4 Results

The tables below summarize the two learners' performance in a simulation with 2000 trials.

Original Learner Results

Metric	Sum of rewards	Steps taken
Mean	80.3	5.7
Standard deviation	23.3	4.4
Minimum	-195.0	1.0
Q1	73.0	3.0
Median	86.0	5.0
Q3	95.0	7.0
Maximum	100.0	59.0
Destinations reached: 2000/2000		
Trials with positive sum of rewards: 1974/2000		
States visited: 727		

The original learner, with the learning rate $1.0/(x + 1)$, had very good results, reaching the destination in all trials and netting a positive sum of rewards over 98% of the time. However, its results are not as good as the perfect planner's; its median sum of rewards is somewhat smaller, for instance, while the median number of steps taken is a little larger. In other words, this learner converges to a good policy, but not a perfect one.

Final Learner Results

Metric	Sum of rewards	Steps taken
Mean	89.2	4.1
Standard deviation	17.0	3.5
Minimum	-376.0	1.0
Q1	86.0	3.0
Median	91.0	4.0
Q3	95.0	5.0
Maximum	100.0	100.0
Destinations reached: 1999/2000		
Trials with positive sum of rewards: 1993/2000		
States visited: 716		

The final learner, with the learning rate $1.0/\ln(x+1)$, missed its destination in 1 of the 2000 trials simulated, and netted a negative sum of rewards in 7 trials. However, the percentiles of its results are identical to the perfect planner’s. This indicates that a simple change in the learning rate decay was sufficient to achieve a perfect planning strategy through reinforcement learning.

In summary, my original learning agent was able to comfortably beat the random agent, or the “lower bound” of the benchmark. But it was still below the benchmark’s “upper bound”, defined by the perfect agent. The final learning agent, on the other hand, does achieve results comparable to the perfect agent’s, thus reaching the benchmark’s “upper bound”.

Results (2000 Trials)				
Metric	Random Planner	Perfect Planner	Original Learner	Final Learner
Mean steps taken	47.9	3.7	5.7	4.1
Mean sum of rewards	-99.2	91.2	80.3	89.2
Destinations reached	1752	2000	2000	1999
Trials with positive sum of rewards	741	2000	1974	1993
States visited	744	244	727	716

5 Conclusion

5.1 Free-Form Visualization

Figure 2 shows the rolling means, over 100 trials, of each planner’s number of steps taken per trial. The figure shows how each learner’s behavior changes over time:

- The **perfect planner**’s behavior is the same throughout the trials. This is to be expected, since this instance does not learn from experience. Since the perfect planner always performs an action that leads to the decrease of the distance between the agent and its destination, its behavior is ideal in all trials. The small fluctuations that happen are due to different initial positions of the agent and its destination, which modify the distance between these two points and consequently the number of steps taken by the agent in a given trial.
- The **random planner**’s behavior also does not change over time, again because the planner does not learn from experience. But its results fluctuate wildly over time: the random nature of the planner allows for “lucky” or “unlucky” streaks, during which it may seem as if the agent’s behavior is improving or worsening. However, there is no trend in the planner’s erratic behavior, which remains fluctuating around a stationary value for the number of steps taken at each trial.
- The **original learner**’s results improve markedly at the beginning of the simulation, over the first 200 trials or so, and then become stationary slightly below the perfect planner’s results. This is in line with what we saw from the statistics of the original planner’s results: the planner converges to a good, but suboptimal, behavior.

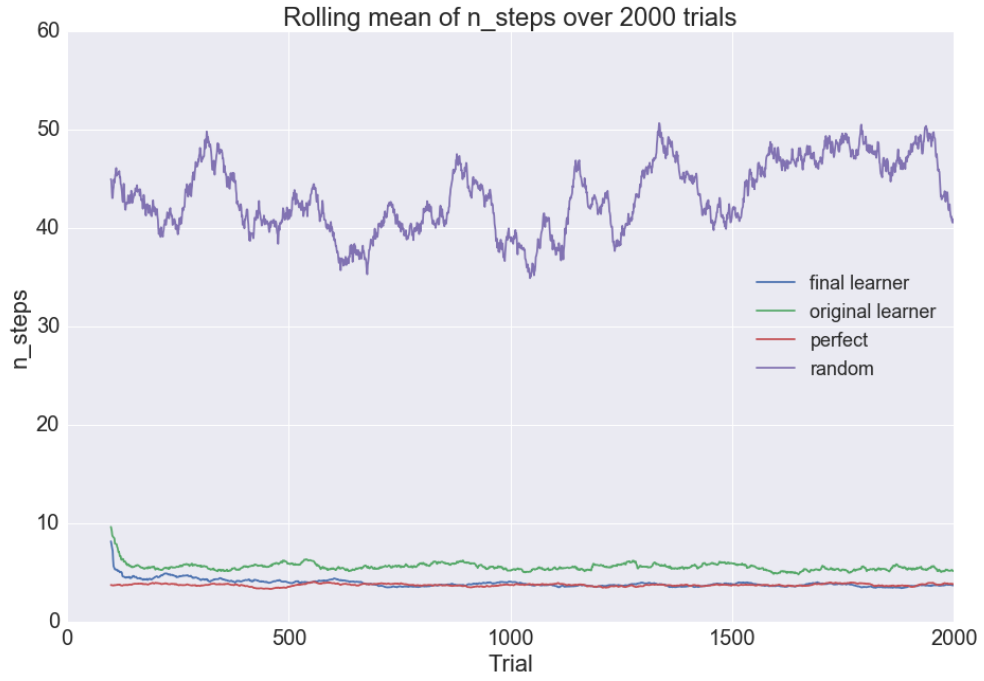


Figure 2: The rolling means over 100 trials of the number of steps taken at each trial by the planner. Taking more than 7 steps necessarily indicates a less than perfect strategy, since, in an 8×6 grid world, any position may be reached from any other in 7 steps or less.

- The **final learner**'s results also improve markedly at the beginning of the trial; but they continue to improve, at a smaller rate, until a little after the 500th trial they become indistinguishable from the perfect planner's results. This is also in line with my previous analysis of the final learner's statistics: after a while, this learner converges to an optimal policy.

5.2 Reflection

When I started this project, the grid world and the planners were all defined as a series of functions. Revisiting the project after a while, it became clear to me that this would lead to code repetition and confusion, so I decided to implement the `PlannerWorld` and `Planner` classes described above. This made for a much simpler and clearer code.

It was also after some time away from the project that I decided to modify the learning rate decay, and I was surprised by, and very satisfied with, the results. Since the original learner already had results close to the perfect planner, I believed there was very little room for improvement in the learner, and that any improvement would be difficult to perceive. However, it was clear from the start that the optimized learner had achieved a policy comparable to that of the perfect planner.

My first version of the final planner also included a modification of the Q -values assigned to unseen (*state, action*) pairs. Following the "optimism under uncertainty" method, I increased this initial value to make the learner more exploratory. However, once I modified the learning rate decay I saw this was the only modification necessary, so I returned the initial value to 0 for my final planner.

5.3 Improvement

In the Machine Learning Engineer's project 4, I had to teach a smartcab to respect traffic rules and follow the waypoints provided by a planner. In this project, I came up with a learning planner that would be able to pass these waypoints along to an agent.

However, the planner I implemented here does not consider what the environment around the smartcab is. Adding this dimension to the planner may improve its results, since the planner may realize, for instance, that an action is better than another because, while both reduce the agent's distance to its destination, only one of them can be performed immediately. This way the agent may be able to escape traffic and red lights.²

A more prosaic improvement of the project would be leveraging modular arithmetic to write cleaner versions of the planner world's `get_delta()` method and the perfect planner's `policy()` method, both of which seem to me unnecessary long and convoluted.

²For instance, suppose the destination is Northeast of the agent and that the agent is facing North. In this situation, moving forward and turning right would both decrease the agent's distance to its destination. However, if the light is red at the agent's intersection and there's no more traffic, it can only move right. A planner that was aware of the environment around the agent could, in this situation, prefer to tell the smartcab to turn right, an action that can be immediately performed.