Teach a smartcab to drive

Project 4 for the Udacity Machine Learning Nanodegree

In this project, we will use Q-learning to train a smartcab to follow traffic rules and reach it's destination in a timely manner.

All code was developed in the accompanying Jupiter notebook, and exported directly to the agent.py, located in the smartcab directory. The project report may be viewed with or without code included inline, by choosing the appropriate pdf, or directly from the ipynb file as a notebook.

Setup

You need Python 2.7 and pygame for this project: https://www.pygame.org/wiki/GettingStarted For help with installation, it is best to reach out to the pygame community [help page, Google group, reddit].

In [21]:

```
# Import what we need, and setup the basic function to run from later.

import math
import string
import sys
import os
import random

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
try: # attempt to determine if we are running within a notebook or on the console.
    cfg = get_ipython().config
    if cfg['IPKernelApp']['parent_appname'] == 'ipython-notebook':
        print "in notebook"
        from IPython.display import display # Allows the use of display() for DataFrames
        %matplotlib inline
    else:
        print "in notebook"
        from IPython.display import display # Allows the use of display() for DataFrames
        %matplotlib inline
    console=False # either condition above means that we are in a Notebook
except NameError:
    print "in console"
```

```
    console=True

sys.path.append("./smartcab/")
from environment import Agent, Environment
from planner import RoutePlanner
from simulator import Simulator

print "Environment ready"
```

```
in notebook
Environment ready
```

In [22]:

```
# Several of the provided modules output unuseful information during each run.
#  Here we provide a way to supress that output as needed.
class outputRedirect():
    def __init__(self):
        self.stout_orig=sys.stdout  # save the current state

    def reset(self): #restore to the original state when initiated
        sys.stdout = self.stout_orig
        print "stdout restored!"

    def suppress_output(self): # a well formed name for the default of the
redirect_output
        self.redirect_output()

    def redirect_output(self,f= open(os.devnull, 'w')): # redirect to f, if provided,
otherwise to null
        try:
            print "redirecting stdout...."
            sys.stdout = f
        except:
            return "couldn't open destination..."
            self.reset = f

redirector=outputRedirect()

print "Redirector ready"
```

```
Redirector ready
```

In [23]:

```
def run(agentType,trials=10, gui=False, deadline=True, delay=0):
    """Run the agent for a finite number of trials."""

    # Set up environment and agent

    if gui ==False:
        redirector=outputRedirect()
        redirector.suppress_output()
        delay=0

    e = Environment()  # create environment (also adds some dummy traffic)
    a = e.create_agent(agentType)  # create agent
    e.set_primary_agent(a, enforce_deadline=deadline)  # specify agent to track
    # NOTE: You can set enforce_deadline=False while debugging to allow longer trials

    # Now simulate it
    sim = Simulator(e, update_delay=delay, display=gui)  # create simulator (uses pygame
when display=True, if available)
    # NOTE: To speed up simulation, reduce update_delay and/or set display=False

    sim.run(n_trials=trials)  # run for a specified number of trials
    # NOTE: To quit midway, press Esc or close pygame window, or hit Ctrl+C on the
command-line

    if gui ==False:
        redirector.reset()

    print "Successful runs = {}".format(a.goal)
    print "----------------------------------------------------------"
    features= [] # the state at each turn in the run
    deadlines = [] # the deadline at each turn in the run
    for i in range(len(a.features)):
        features.append(pd.DataFrame(a.features[i]).T)
        deadlines.append(a.deadline[i])

    rewards=[] # the total reward for each run
    for i in range(len(a.total_reward)):
        rewards.append(a.total_reward[i])



    try:
        print "Qtable: ",len(a.Qtable)
        print "state=light, oncoming, right, left, next_waypoint  / actions=
```

```
{}\n".format(a.availableAction)
        for r in a.Qtable:
            print "state={}, {}, {}, {}, {} / action={}".format(r[0],r[1],r[2],r[3],r[4],
a.Qtable[r])
    except:
        print "no Qtable"

    return features,deadlines,rewards


print "run ready"
```

```
run ready
```

In [24]:

```
# display the feedback from the prior runs graphically
def statsFromRun(feat,DL,RW):
    left=pd.Series()
    light=pd.Series()
    next_waypoint=pd.Series()
    oncoming=pd.Series()
    right=pd.Series()
    for f in feat:
        left= left.add(pd.value_counts(f.left.ravel()), fill_value=0)
        light= light.add(pd.value_counts(f.light.ravel()), fill_value=0)
        next_waypoint= next_waypoint.add(pd.value_counts(f.next_waypoint.ravel()),
fill_value=0)
        oncoming= oncoming.add(pd.value_counts(f.oncoming.ravel()), fill_value=0)
        right= right.add(pd.value_counts(f.right.ravel()), fill_value=0)

    fig, axes = plt.subplots(nrows=2, ncols=3,figsize=(14,6))
    fig.suptitle( "Runs:{}".format(len(feat)))

    left.plot(kind='bar', title="Left",ax=axes[0,0])
    light.plot(kind='bar', title="light",ax=axes[0,1])
    next_waypoint.plot(kind='bar', title="next_waypoint",ax=axes[0,2])
    oncoming.plot(kind='bar', title="oncoming",ax=axes[1,0])
    right.plot(kind='bar', title="right",ax=axes[1,2])
    axes[1,1].plot(DL,label="Deadlines")
    axes[1,1].plot(RW,label="Rewards")
    avgDist=3
    axes[1,1].plot(     #add a line to the graph representing the avg of all point within
avgDist of the current run.
```

```
        [(np.mean(DL[i-avgDist:i+avgDist])+np.mean(RW[i-avgDist:i+avgDist]))/2 for i in
range(len(DL))],
        label="Avg {:2.2f}".format( # use the last half avg in the label
            (np.mean(DL[len(DL)/2:len(DL)])+np.mean(RW[len(DL)/2:len(DL)]))/2))
    #axes[1,1].xlabel('Run')
    axes[1,1].legend(loc=2)
    #axes[1,1].title("Deadline and Rewards per Run")

    plt.show()
    plt.close()


def scorePerRun(DL,RW):
    plt.figure(figsize=(14,6))
    plt.plot(DL,label="Deadlines")
    plt.plot(RW,label="Rewards")
    avgDist=3
    plt.plot(     #add a line to the graph representing the avg of all point within
avgDist of the current run.
        [(np.mean(DL[i-avgDist:i+avgDist])+np.mean(RW[i-avgDist:i+avgDist]))/2 for i in
range(len(DL))],
        label="Avg {:2.2f}".format( # use the last half avg in the label
            (np.mean(DL[len(DL)/2:len(DL)])+np.mean(RW[len(RW)/2:len(RW)]))/2))
    avgDist=6
    plt.plot(     #add a line to the graph representing the avg of DL within avgDist of
the current run.
        [np.mean(DL[i-avgDist:i+avgDist]) for i in range(len(DL))],
        label="DL Avg {:2.2f}".format( # use the last half avg in the label
            np.mean(DL[len(DL)/2:len(DL)])))
    plt.plot(     #add a line to the graph representing the avg of RW within avgDist of
the current run.
        [np.mean(RW[i-avgDist:i+avgDist]) for i in range(len(RW))],
        label="RW Avg {:2.2f}".format( # use the last half avg in the label
            np.mean(RW[len(RW)/2:len(RW)])))

    plt.xlabel('Run')
    plt.legend()
    plt.title("Deadline and Rewards per Run")
    plt.show()
    plt.close()

print "Graph display ready"
```

```
Graph display ready
```

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

Next waypoint location, relative to its current location and heading, Intersection state (traffic light and presence of cars), and, Current deadline value (time steps remaining), And produces some random move/action (None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce_deadline set to False (see run function in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

In [25]:

```
class RandomAgent(Agent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
        super(RandomAgent, self).__init__(env)  # sets self.env = env, state = None,
next_waypoint = None, and a default color
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner to get
next_waypoint
        # TODO: Initialize any additional variables here
        self.availableAction = ['forward', 'left', 'right',None]
        self.goal=0
        self.steps=0
        self.features=[]
        self.deadline=[]
        self.total_reward=[0]

    def reset(self, destination=None):
        self.planner.route_to(destination)
        # TODO: Prepare for a new trip; reset any variables here, if required
        #print"RESET, Final state:\n", self.state
        try:
            if self.deadline[len(self.features)-1] >0: #deadline less than zero
                self.goal+=1 #FIXME - order
                print "PASS! {} steps to goal,Goal reached {} times out of {}!".format(

self.deadline[len(self.features)-1],self.goal,len(self.features))
```

```
            else:
                print "FAIL! {} steps to goal,Goal reached {} times out of {}!".format(

self.deadline[len(self.features)-1],self.goal,len(self.features))
                pass
        except:
            print "Trial 0 - Goal reached {} times out of
{}!".format(self.goal,len(self.features))
            pass
        print "-----------------------------------------------------------"
        self.features.append({})
        self.deadline.append(None)
        self.total_reward.append(0)
        self.steps=0

    def update(self, t):
        # Gather inputs
        self.steps+=1
        self.next_waypoint = self.planner.next_waypoint()  # from route planner, also
displayed by simulator
        inputs = self.env.sense(self)
        #self.deadline[len(self.features)] = self.env.get_deadline(self)
        self.state=inputs
        self.features[len(self.features)-1][self.steps]=inputs
        self.deadline[len(self.deadline)-1] = self.env.get_deadline(self)

        # TODO: Select action according to your policy
        action = self.availableAction[random.randint(0,3)]

        # Execute action and get reward
        reward = self.env.act(self, action)
        self.lastReward=reward
        # TODO: Learn policy based on state, action, reward
        self.total_reward[len(self.total_reward)-1]
=self.total_reward[len(self.total_reward)-1]+reward
        #print "LearningAgent.update():deadline{}, inputs{}, action = {}, reward = {},
next_waypoint = {}".format(
        #                                          deadline, inputs, action,
reward,self.next_waypoint, )  # [debug]
print "RandomAgent ready"
```

```
RandomAgent ready
```

In [26]:

```
if console == False:
    features,deadlines, rewards=run(agentType=RandomAgent,trials=2, deadline=False)
#Example of a random run, with no deadline
    print "RandomAgent, no deadlines, Done"
```

```
redirecting stdout....
stdout restored!
Successful runs = 0
------------------------------------------------------------
Qtable:  no Qtable
RandomAgent, no deadlines, Done
```

In [27]:

```
if console == False:
    features,deadlines, rewards=run(agentType=RandomAgent,trials=2, deadline=True)
#Example of a random run
    print "RandomAgent, with deadlines, Done"
```

```
redirecting stdout....
stdout restored!
Successful runs = 0
------------------------------------------------------------
Qtable:  no Qtable
RandomAgent, with deadlines, Done
```

Random Agent - Discussion:

When we run an agent with a random action policy, we see that it will move about the board with no pattern, and will eventually reach the destination, in most cases. If we allow the use of deadlines, we see that the agent rarely reaches the destination in time, although it may still occur.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

In [28]:

```
class StateAgent(RandomAgent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
        super(StateAgent, self).__init__(env)  # sets self.env = env, state = None,
next_waypoint = None, and a default color
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner to get
next_waypoint
        # TODO: Initialize any additional variables here
        self.availableAction = ['forward', 'left', 'right',None]
        self.next_waypoint   = None
        self.goal=0
        self.steps=0
        self.features=[]
        self.total_reward=[0]


    def update(self, t):
        # Gather inputs
        self.steps+=1

        self.lastWaypoint = self.next_waypoint
        self.next_waypoint = self.planner.next_waypoint()  # from route planner, also
displayed by simulator
        inputs = self.env.sense(self)

        deadline = self.env.get_deadline(self)

        # TODO: Update state

        inputs['next_waypoint']=self.next_waypoint
        self.state= inputs
        self.deadline[len(self.deadline)-1] = self.env.get_deadline(self)
        self.features[len(self.features)-1][self.steps]=inputs
        # TODO: Select action according to your policy

        action = self.availableAction[random.randint(0,3)]

        # Execute action and get reward
        reward = self.env.act(self, action)
        # TODO: Learn policy based on state, action, reward

        self.total_reward[len(self.total_reward)-1]
```

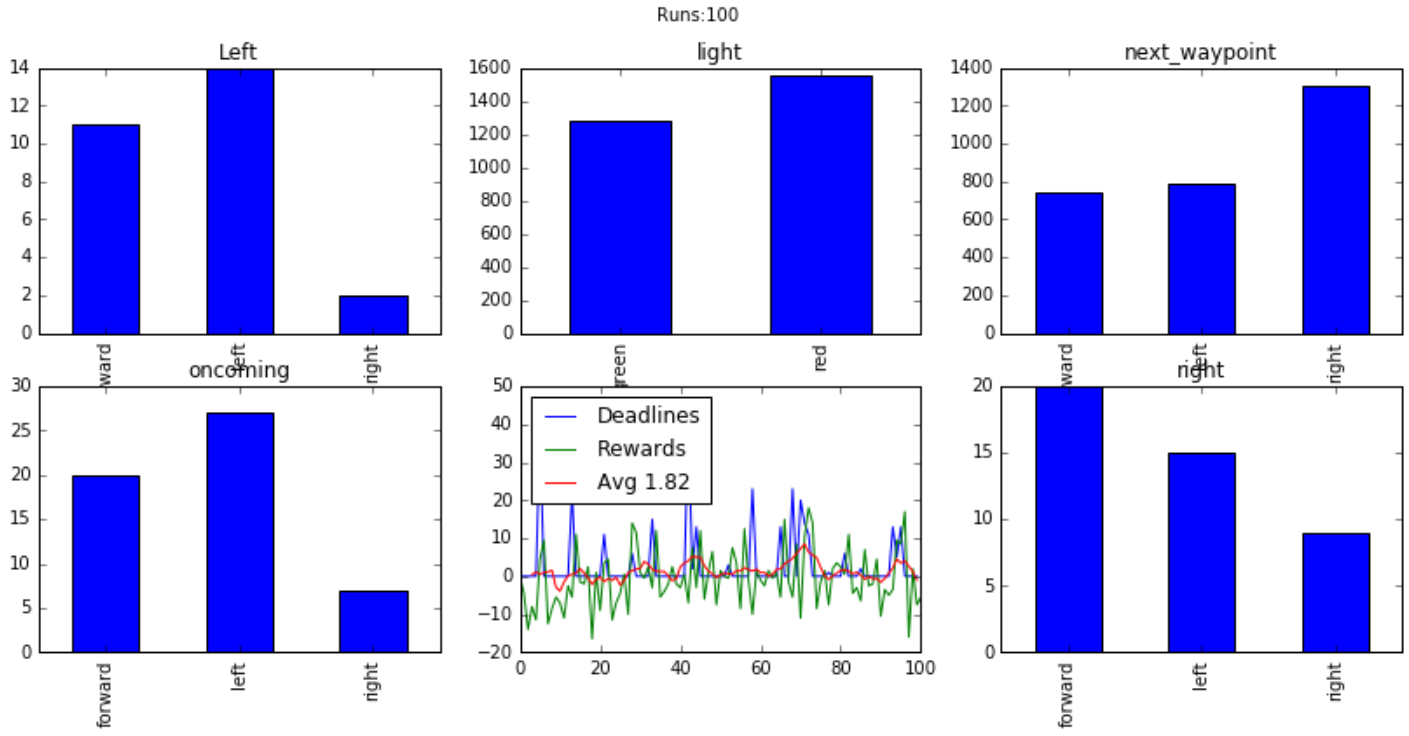
```
=self.total_reward[len(self.total_reward)-1]+reward

        #print "LearningAgent.update(): self.state{}, action = {}, reward = {},
next_waypoint = {}".format(
        #                                        self.state, action,
reward,self.next_waypoint, )  # [debug]
print "StateAgent Ready"
```

```
StateAgent Ready
```

In [29]:

```
if console == False:
    # run the trials for the state
    stateFeatures,StateDeadlines,StateRewards=run(agentType=StateAgent,trials=100)
    statsFromRun(stateFeatures,StateDeadlines,StateRewards)
    #scorePerRun(StateDeadlines,StateRewards)
    print "StateAgent done"
```

```
redirecting stdout....
stdout restored!
Successful runs = 21
------------------------------------------------------------
Qtable:  no Qtable
```




```
StateAgent done
```

Identify and update state - Discussion.

When we sense our environment, we perceive 5 variables, with several possible states These include: left, light, next_waypoint, oncoming, and right. We can see right away that light and next_waypoint contains new information at every poll, while the others usually have no value.

It's not readily apparent that the direction of travel information of the other cars (described by left/right/oncoming) is relevant to our agent. A case could be made to remove the direction information, and only retain information about another car being present at the light. This would have the benefit of reducing the number of possible states, increasing the learning speed of the agent. This may be a valuable approach in resource constrained environments.

The downside is that the agent may pick an action that causes a longer trip. Early in the learning phase, it could also pick an action incorrectly. For instance, by proceeding through a light when the opposite car is turning left. In this case, it may have previously seen a positive reward for moving through the light, because the opposite car was not turning. This time through, it will receive a negative reward, and in the future when a car is at the oncoming light, it will always wait till the intersection is clear.

In the interest of correct action, we will choose to use the state as returned from the sensor, with the addition of the next_waypoint.

While I have tracked the deadline, it is not apparent that it will provide useful information to the agent. It is useful to note that the agent does not see any change in the deadline value yet. We may expect this to adapt as we implement learning.

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

In [30]:

```
class BasicLearningAgent(RandomAgent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
        super(BasicLearningAgent, self).__init__(env)  # sets self.env = env, state =
None, next_waypoint = None
        self.color = 'red'  # override color
```

```
        self.planner = RoutePlanner(self.env, self)  # simple route planner to get
next_waypoint
        # TODO: Initialize any additional variables here
        self.availableAction = [None,'forward', 'left', 'right']
        self.next_waypoint   = None
        self.goal=0
        self.steps=0
        self.features=[]
        self.Qtable={}
        self.epsilon=0.0
        self.gamma=0.0
        self.total_reward=[0]
        self.alpha = .2

    def get_state(self):
        inputs = self.env.sense(self)
        inputs['next_waypoint']=self.planner.next_waypoint()
        return (inputs['light'], inputs['oncoming'], inputs['right'],
inputs['left'],inputs['next_waypoint'])

    def set_action(self):
        action = self.availableAction[random.randint(0,3)]    #take a random action

        # 1-epsilon % of time, refer to the q-table for an action. take the max value
from the available actions
        if self.epsilon < random.random() and  self.Qtable.has_key(self.state):

action=self.availableAction[self.Qtable[self.state].index(max(self.Qtable[self.state]))]
        return action

    def update_q_table(self,action,reward):
        if not self.Qtable.has_key(self.state):
            self.Qtable[self.state]=[0,0,0,0]

        new_state=self.get_state()
        if not self.Qtable.has_key(new_state):
            self.Qtable[new_state]=[0,0,0,0]

        #Set the new value in the Q table based on the Q-learning method
        #Q[s,a] ←Q[s,a] + α(r+ γ*maxQ[s',a'] - Q[s,a])

        self.Qtable[self.state][self.availableAction.index(action)]=(
                        self.Qtable[self.state][self.availableAction.index(action)]+ #
Q[s,a] +
```

```
                        self.alpha*   # α
                            (reward+self.gamma*max(self.Qtable[new_state])-   #
r+maxQ[s',a'] -
                             self.Qtable[self.state][self.availableAction.index(action)])
# Q[s,a]
                        )

    def update(self, t):

        # Gather inputs
        self.steps+=1
        self.next_waypoint = self.planner.next_waypoint()  # from route planner, also
displayed by simulator

        # TODO: Update state
        self.state = self.get_state()

        #store details of this run
        self.deadline[len(self.deadline)-1] = self.env.get_deadline(self)
        self.features[len(self.features)-1][self.steps]=self.env.sense(self)
        self.features[len(self.features)-1][self.steps]
['next_waypoint']=self.planner.next_waypoint()

        # TODO: Select action according to your policy
        action = self.set_action()
        # Execute action and get reward
        reward = self.env.act(self, action)

        # TODO: Learn policy based on state, action, reward
        self.update_q_table(action,reward)

        #store details about our rewards
        self.total_reward[len(self.total_reward)-1]
=self.total_reward[len(self.total_reward)-1]+reward



print "BasicLearningAgent Ready"
```

```
BasicLearningAgent Ready
```

In [31]:

```
if console == False:
    # run the trials for the Basic Q learning agent
```

```
    basicLearnFeatures,BLdeadlines,BLrewards=run(agentType=BasicLearningAgent,trials=100,
deadline=True)
    #statsFromRun(basicLearnFeatures,BLdeadlines,BLrewards)
    scorePerRun(BLdeadlines,BLrewards)
    print "Basic Q Learning Agent done"
```

```
redirecting stdout....
stdout restored!
Successful runs = 0
------------------------------------------------------------
Qtable:  50
state=light, oncoming, right, left, next_waypoint  / actions= [None, 'forward', 'left',
'right']

state=green, None, None, right, forward / action=[0, 0.4, 0, 0]
state=green, None, None, left, right / action=[0.0, 0, -0.1, 0]
state=red, left, None, None, left / action=[0.0, 0, -0.2, 0]
state=green, None, forward, None, left / action=[0, -0.1, 0, 0]
state=green, None, right, None, right / action=[0.0, 0, -0.1, 0]
state=green, None, None, None, left / action=[0.0, 0, 0, -0.1]
state=red, left, None, None, forward / action=[0, -0.2, 0, 0]
state=green, None, None, left, forward / action=[0, 0.976, 0, 0]
state=red, None, None, left, forward / action=[0.0, 0, -0.2, 0]
state=red, left, None, None, right / action=[0.0, 0, -0.2, 0]
state=red, None, forward, None, left / action=[0, -0.2, 0, 0]
state=red, None, right, None, left / action=[0.0, 0, 0, 0]
state=red, None, None, right, left / action=[0.0, 0, 0, -0.1]
state=red, right, None, None, left / action=[0.0, 0, 0, 0]
state=red, None, None, None, left / action=[0.0, 0, -0.2, 0]
state=green, None, None, None, forward / action=[0.0, 0, 0, 0]
state=green, None, right, None, left / action=[0.0, -0.1, 0, 0]
state=green, None, forward, None, forward / action=[0, 0.7200000000000001, 0, 0]
state=red, None, None, forward, right / action=[0.0, -0.2, 0, 0]
state=red, None, None, right, right / action=[0, 0, -0.2, 0]
state=red, None, left, None, right / action=[0.0, 0, 0, 0]
state=red, None, forward, None, forward / action=[0.0, -0.2, 0, 0]
state=red, forward, None, None, left / action=[0.0, -0.2, 0, 0]
state=red, None, None, None, right / action=[0.0, -0.2, 0, 0]
state=green, None, None, right, left / action=[0.0, 0, 0, 0]
state=green, None, left, None, right / action=[0.0, -0.1, 0, 0]
state=red, forward, None, None, right / action=[0, 0, 0, 0.7200000000000001]
state=green, None, None, left, left / action=[0.0, 0, 0, -0.1]
state=green, None, None, forward, right / action=[0.0, 0, 0, 0]
state=green, None, None, right, right / action=[0.0, 0, 0, 0]
```

```
state=green, None, left, None, forward / action=[0, 0.4, 0, 0]
state=green, None, left, None, left / action=[0.0, 0, 0, -0.1]
state=red, None, right, None, right / action=[0.0, -0.2, 0, 0]
state=green, left, None, None, forward / action=[0, 0, -0.1, 0]
state=red, None, None, left, left / action=[0.0, 0, 0, -0.1]
state=red, None, None, None, forward / action=[0.0, -0.2, 0, 0]
state=red, right, None, None, forward / action=[0.0, 0, 0, -0.1]
state=red, None, left, None, left / action=[0.0, -0.2, 0, 0]
state=red, None, None, left, right / action=[0, -0.2, 0, 0]
state=green, None, None, forward, left / action=[0, 0, 0.4, 0]
state=green, right, None, None, left / action=[0, 0, -0.2, 0]
state=red, None, None, right, forward / action=[0.0, 0, 0, 0]
state=red, None, forward, None, right / action=[0.0, 0, 0, 0]
state=red, None, right, None, forward / action=[0.0, 0, 0, 0]
state=red, right, None, None, right / action=[0.0, 0, -0.2, 0]
state=green, None, forward, None, right / action=[0.0, 0, 0, 0]
state=green, None, None, None, right / action=[0.0, 0, 0, 0]
state=green, forward, None, None, right / action=[0, -0.1, 0, 0]
state=green, left, None, None, right / action=[0.0, 0, 0, 0]
state=green, None, right, None, forward / action=[0.0, 0, 0, -0.1]
```


```
Basic Q Learning Agent done
```

In [33]:

```
class BasicLearningAgent2(BasicLearningAgent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
```

```
        super(BasicLearningAgent2, self).__init__(env)  # sets self.env = env, state =
None, next_waypoint = None
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner to get
next_waypoint
        # TODO: Initialize any additional variables here
        self.availableAction = ['forward', 'left', 'right',None]
        self.next_waypoint   = None
        self.goal=0
        self.steps=0
        self.features=[]
        self.Qtable={}
        self.epsilon=0.0
        self.gamma=0.0
        self.total_reward=[0]

if console == False:
    # run the trials for the Basic Q learning agent

basicLearn2Features,BL2deadlines,BL2rewards=run(agentType=BasicLearningAgent2,trials=100,
deadline=True)
    #statsFromRun(basicLearn2Features,BL2deadlines,BL2rewards)
    scorePerRun(BL2deadlines,BL2rewards)
    print "Basic Q Learning Agent2 done"
```

```
redirecting stdout....
stdout restored!
Successful runs = 99
----------------------------------------------------------
Qtable:  40
state=light, oncoming, right, left, next_waypoint  / actions= ['forward', 'left',
'right', None]

state=green, None, forward, None, forward / action=[0.7200000000000001, 0, 0, 0]
state=green, None, right, None, forward / action=[0, 0, 0, 0.0]
state=red, None, right, None, None / action=[0, 0, 0, 0]
state=red, left, None, None, left / action=[0, 0, 0, 0]
state=red, forward, None, None, forward / action=[-0.2, -0.2, 0, 0.0]
state=green, None, None, None, left / action=[0, 3.184926837290606, 0, 0]
state=red, left, None, None, forward / action=[-0.2, -0.2, -0.1, 0]
state=green, None, left, None, forward / action=[0.4, 0, 0, 0]
state=red, None, None, left, forward / action=[-0.2, -0.2, 0, 0]
state=red, left, None, None, right / action=[-0.2, -0.2, 0, 0]
state=red, None, None, right, left / action=[0, -0.2, 0, 0]
```

```
state=green, right, None, None, forward / action=[0.4, -0.2, 0, 0]
state=red, left, None, None, None / action=[0, 0, 0, 0]
state=red, None, None, None, left / action=[-0.2, -0.2, -0.1, 0.0]
state=red, left, None, left, forward / action=[0, 0, 0, 0]
state=green, None, None, None, forward / action=[4.730763038064051, 0, 0, 0]
state=green, None, None, forward, forward / action=[0.7200000000000001, 0, -0.1, 0]
state=red, None, None, forward, forward / action=[0, 0, 0, 0]
state=green, None, None, None, None / action=[0, 0, 0, 0]
state=red, None, None, None, right / action=[-0.2, -0.2, 4.437750845382559, 0]
state=green, None, None, right, left / action=[0, 0, -0.1, 0]
state=green, None, left, None, right / action=[0, 0, 0, 0]
state=red, forward, None, None, right / action=[-0.2, -0.2, 2.2560000000000002, 0]
state=green, None, left, None, left / action=[-0.1, 0, -0.1, 0]
state=green, None, None, right, forward / action=[0, 0, 0, 0]
state=green, None, None, left, forward / action=[3.8560000000000003, 0, 0, 0]
state=red, None, None, None, None / action=[0, 0, 0, 0]
state=green, None, None, left, left / action=[0, 0, 0, 0]
state=green, left, None, None, forward / action=[4.5760000000000005, 0, 0, 0]
state=green, forward, None, None, forward / action=[0.7200000000000001, 0, 0, 0]
state=red, None, None, None, forward / action=[-0.2, -0.2, -0.1, 0.0]
state=red, right, None, None, forward / action=[-0.2, -0.2, 0, 0]
state=green, None, None, forward, None / action=[0, 0, 0, 0]
state=green, right, None, None, left / action=[0, 0, 0, 0.0]
state=red, None, forward, None, right / action=[0, -0.2, 0, 0]
state=red, None, None, right, forward / action=[0, 0, -0.1, 0]
state=green, left, None, None, left / action=[0, 0.4, 0, 0]
state=green, None, None, None, right / action=[-0.1, -0.1, 2.001528561117471, 0]
state=green, left, None, None, right / action=[-0.1, 0, 0, 0.0]
state=red, None, right, None, forward / action=[-0.2, -0.2, -0.1, 0]
```


```
Basic Q Learning Agent2 done
```

Implement Q-Learning - Discussion

With a basic Qlearning algorithm, we note that the agent quickly learns a set of rules that allow the agent to move toward the objective. Generally speaking the agent, is moving to the destination, but does not always make optimal choices. We see the agent begin to obey some traffic rules and make moves toward its destination. We can see that now the reward in each run is generally positive.

We don't see a large continued increase in the agent speed toward the destination after the initial runs. We still see strange behavior such as repeated right turns back to the original destination, or staying in a no action state for extended periods, without much change in pattern through the run. With Epsilon set to 0, we don't see the agent select new behaviors, and with Gamma at 0 we don't consider our future state.

One interesting note is the difference made by re-ordering the available action list. Placing the 'None' value at the end plays to the bias of the 'max' function, in which if all values are the same, will select the first match. Since we initialize the list to zero, if 'None' is first in the list, the agent tends to take no action, and never reaches the goal. By reordering the list, we bias the agent toward action, allowing the agent to perform significantly better.

Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the

agent. How well does it perform?

Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

In [34]:

```
class LearningAgent(BasicLearningAgent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
        super(LearningAgent, self).__init__(env)  # sets self.env = env, state = None,
next_waypoint = None
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner to get
next_waypoint
        # TODO: Initialize any additional variables here
        self.availableAction = [ None,'forward', 'left', 'right']
        self.next_waypoint = None
        self.goal=0
        self.steps=0
        self.features=[]
        self.Qtable={}
        self.epsilon=0.95
        self.epsilon_end=0.125
        self.gamma=0.05
        self.gamma_end=0.125
        self.total_reward=[0]
        self.alpha = 0.374625

    def set_action(self):
        #initially we want to prefer a random action, but later we would like to trust
our experience.
        exploration_rate=32 #rate at which we approach final epsilon-> higher is slower
        self.epsilon = self.epsilon-(self.epsilon-self.epsilon_end)/exploration_rate

        action = self.availableAction[random.randint(0,3)]    #take a random action
        # 1-epsilon % of time, refer to the q-table for an action. take the max value
from the available actions
        if self.epsilon < random.random() and  self.Qtable.has_key(self.state):

action=self.availableAction[self.Qtable[self.state].index(max(self.Qtable[self.state]))]
        return action

    def update_q_table(self,action,reward):
```

```
        # if we haven't seen this state yet, initialize it
        if not self.Qtable.has_key(self.state):
            self.Qtable[self.state]=
[random.uniform(-1,1),random.uniform(-1,1),random.uniform(-1,1),random.uniform(-1,1)]
        #we took an action, so can now observe our new state.
        new_state=self.get_state()
        if not self.Qtable.has_key(new_state):
            self.Qtable[new_state]=
[random.uniform(-1,1),random.uniform(-1,1),random.uniform(-1,1),random.uniform(-1,1)]

        #initially we should have a very low gamma, as we can't trust our knowledge.
        # as time goes by we should give more weight to our knowledge and grow gamma.
        rate=16 #-> higher is slower
        self.gamma=self.gamma+(self.gamma_end-self.gamma)/rate

        #Set the new value in the Q table based on the Q-learning method
        #Q[s,a] ←Q[s,a] + α(r+ γ*maxQ[s',a'] - Q[s,a])

        self.Qtable[self.state][self.availableAction.index(action)]=(
                        self.Qtable[self.state][self.availableAction.index(action)]+ #
Q[s,a] +
                        self.alpha*   # α
                            (reward+self.gamma*max(self.Qtable[new_state])-   #
r+maxQ[s',a'] -
                             self.Qtable[self.state][self.availableAction.index(action)])
# Q[s,a]
                        )

    def update(self, t):

        # Gather inputs
        self.steps+=1
        self.next_waypoint = self.planner.next_waypoint()  # from route planner, also
displayed by simulator

        # TODO: Update state
        self.state = self.get_state()

        #store details of this run
        self.deadline[len(self.deadline)-1] = self.env.get_deadline(self)
        self.features[len(self.features)-1][self.steps]=self.env.sense(self)
        self.features[len(self.features)-1][self.steps]
['next_waypoint']=self.planner.next_waypoint()
```

```
        # TODO: Select action according to your policy
        action = self.set_action()
        # Execute action and get reward
        reward = self.env.act(self, action)

        # TODO: Learn policy based on state, action, reward
        self.update_q_table(action,reward)

        #store details about our rewards
        self.total_reward[len(self.total_reward)-1]
=self.total_reward[len(self.total_reward)-1]+reward


print "LearningAgent Ready"
```

```
LearningAgent Ready
```

In [39]:

```
def runGrid(agentType,trials=10, gui=False, deadline=True, delay=0):
    """Run the agent for a finite number of trials."""

    # Set up environment and agent

    if gui ==False:
        redirector=outputRedirect()
        redirector.suppress_output()
        delay=0

    # NOTE: To speed up simulation, reduce update_delay and/or set display=False
    run = 16
    values=16
    iterations=10
    all_successcounts=[]
    for r in range(run):
        for v in range(values):
            for s in range(values):
                # Now simulate it
                success_counts=[]
                for i in range(iterations):
                    e = Environment()  # create environment (also adds some dummy
traffic)
                    a = e.create_agent(agentType)  # create agent
                    e.set_primary_agent(a, enforce_deadline=deadline)  # specify agent to
```

```
track

                    #agent parameter to evaluate
                    a.alpha=(1./run)*(r+1)
                    a.gamma_end=(1./values)*(v+1)
                    a.epsilon_end=(1./values)*(s+1)

                    sim = Simulator(e, update_delay=delay, display=gui)  # create
simulator (uses pygame when display=True, if available)
                    sim.run(n_trials=trials)  # run for a specified number of trials
                    success_counts.append(a.goal)

                all_successcounts.append({'success_counts_avg':np.mean(success_counts),
'alpha':a.alpha,'gamma':a.gamma,'epsilon' :a.epsilon_end})

    if gui ==False:
        redirector.reset()
    print "best success:"
    #print all_successcounts
    all_success_data=pd.DataFrame(all_successcounts)
    #display(all_success_data)

display(all_success_data[all_success_data.success_counts_avg==max(all_success_data.succes
s_counts_avg)])

print "run ready"
```

```
run ready
```

In [ ]:

```
if console == False:
    print "run some tests searching for a best value."
    runGrid(agentType=LearningAgent,trials=100, deadline=True,gui=console, delay=.1)
```

Enhance the driving agent - Discussion

We immediately see the agent begin learning when we begin using epsilon to explore new states. The addition of gamma provides many of the same benefits, and we see that the agent learns to reach the destination as quickly as the first or second run. Following this, the agent will quickly begin to reach it's destination well before the deadline, with a positive score, the majority of times. This is despite biasing the action list to 'None', as in the previous example. Even at the end of the run, we do still see odd behaviors, as the agent tries new methods according to epsilon, or encounters new states.

In addition to tuning the final epsilon and gamma, I have added the ability for each to adjust the amount they affect the outcome over time. Initially we want to prefer a random action, as our table is now initialized with random values, and we want to explore the available states, and record their effects. The opposite is true for gamma. In it's case, we would like to highly discount any initial knowledge initially, and over time grow to trust what we know. My implementation allows us to control the rate of change over time of each. I have selected starting/ending values and rates based on experimentation, and optimized these numbers further by implementing a grid search to test the values over multiple runs, etc. These were found to be:

alpha = 0.374625 epsilon = 0.125 gamma= 0.125 success_counts_avg = 91.92

We could also implement methods to adjust these rates based on factors other than time, for example based on the difference between the current and new values in the Q table for the state, but my initial attempts haven't been successful in finding a model that learns as fast as the current implementations.

I have also changed the Q-table initialization to use random values from -1 to 1. This is intended to assist early phase learning of each state taking random actions, increasing state exploration.

It should be noted that the change that most affects success from the initial algorithm was to re-order the selection of available actions, such that forward is preferred over inaction. This led to an agent that was successful in reaching it's goal at least as often as any other enhancement, usually reaching the goal more than 96% of the time. This may be a quirk of the environment, and not something that is true generally, especially when the learning is moved to environment's that may have a larger state space, or with a more complex reward system. We do see that the final model has a higher average reward per run.

In [38]:

```
if __name__ == '__main__':
    print  "running...."

    QLearnFeatures,QLdeadlines,QLrewards=run(agentType=LearningAgent,trials=100,
deadline=True,gui=console, delay=.1)
    #statsFromRun(QLearnFeatures,QLdeadlines,QLrewards)
    scorePerRun(QLdeadlines,QLrewards)
    print "\nQ Learning Agent done"
```

```
running....
redirecting stdout....
stdout restored!
Successful runs = 92
-----------------------------------------------------------
Qtable:  45
state=light, oncoming, right, left, next_waypoint  / actions= [None, 'forward', 'left',
'right']

state=green, None, forward, None, forward / action=[-0.28637861600804815,
```

```
5.382095180393977, -0.6398023452103123, -0.24715981401303244]
state=green, None, None, right, forward / action=[0.62670830450597, 1.3752279698566299,
-0.07379774660630004, -0.4127826891193236]
state=green, None, None, left, right / action=[-0.05349360588913843, 0.1257491237786934,
0.1931911284336038, -0.2089538235145192]
state=red, left, None, None, left / action=[0.3828742226088695, 0.27717277708561416,
-0.46745125256347736, -0.03605262459166281]
state=green, None, None, forward, left / action=[0.3517358913974602, -0.6494322559450854,
-0.8280030078958738, -0.5849741491943199]
state=red, forward, None, None, forward / action=[0.2435830504270794,
-0.9631005789522138, -0.02319954598698215, -0.2067360106611611]
state=green, None, None, None, left / action=[0.2361297949081176, -0.13003594268620433,
2.6350229033883084, -0.1415754831329289]
state=red, left, None, None, forward / action=[-0.8377969620638934, -0.7512611911212482,
-0.41719747876490804, -0.9879497219665783]
state=green, None, forward, None, None / action=[-0.431265598089571, -0.6137728980388744,
0.5994512617432861, -0.5233545527331214]
state=red, None, left, None, forward / action=[0.3495313049047167, 0.5725139629421878,
-0.7768735920224401, -0.0865916108609277]
state=green, None, None, right, left / action=[0.7795190565306642, -0.5325058129159763,
0.8183594831090361, 0.4287857947643876]
state=red, None, forward, None, left / action=[-0.9571176425370767, -0.8316893498114157,
-0.08012304827582595, 0.472949186956703]
state=red, None, right, None, left / action=[-0.09510783698738856, 0.5242903792241149,
-0.887145985674905, -0.9658542738084728]
state=green, None, right, None, forward / action=[-0.8773187383041803,
-0.5559770418771641, -0.7759507545552038, -0.05462434457800293]
state=green, right, None, None, forward / action=[0.560529194029932, -0.781336289865467,
0.7387724073497179, -0.17162462687573915]
state=red, None, None, None, left / action=[-1.1976315095214124e-20, -0.7562757939859666,
-0.9424821762815131, -0.2078880381023006]
state=red, None, forward, None, forward / action=[-0.09505186096912865,
-0.3702212017441882, -0.3416346826006987, -0.3190626533126053]
state=green, None, None, None, forward / action=[0.5911757979145313, 9.212880804350734,
-0.2230869351432529, -0.01539454346308558]
state=green, None, None, forward, forward / action=[-0.8513338268945243,
-0.4575612359571619, -0.5433012336874228, -0.16214091025422173]
state=red, None, None, forward, forward / action=[0.8891584597964426,
-0.07632871297445032, -0.6511533985146454, 0.5807962119359131]
state=green, None, None, None, None / action=[0.34085765183127714, -0.5414820084516812,
-0.044744967554727344, 0.5970417236774745]
state=red, forward, None, None, left / action=[-0.5402882262110988, 0.5199662653779633,
0.01368682234624985, -0.1575463122219491 5]
state=green, None, left, None, forward / action=[-0.3304427916720638, 1.9195408494550026,
```

```
-0.946381236190524, -0.6761738917385063]
state=red, None, None, None, right / action=[0.07082300035547853, -0.5664556318737085,
-0.6967699215272052, 2.2694892284176937]
state=green, None, right, None, left / action=[0.0608468220329581, -0.3801384707104232,
0.8633437573603349, -0.8663550760020851]
state=green, None, left, None, right / action=[0.16306795825370662, 0.1542301947963489,
0.060996008795279416, -0.8583804743350962]
state=green, None, left, None, left / action=[0.032590262671166714, -0.8602109898199279,
0.1412389427574814, -0.013848873870998618]
state=green, None, None, forward, right / action=[0.41286796941084236,
-0.29124597859651713, 0.2764808823955355, 0.3996299810784947]
state=green, None, None, right, right / action=[0.6293293828302866, 0.6978749995521987,
-0.8296235667509015, 1.0639359295044355]
state=green, None, None, left, forward / action=[-0.6428105116898091, 4.933767900740698,
0.058331244287824546, -0.6042638557782694]
state=red, None, None, None, None / action=[-0.29224264397663924, 0.3561219289265991,
0.10246754621834131, 0.46397066352108096]
state=red, None, None, right, None / action=[0.1955389031662178, -0.2595641903367445,
-0.362210624657884, -0.3333402516345454]
state=green, left, None, None, forward / action=[-0.6872802573221675,
-0.022085354448803995, -0.07165796321446724, -0.3349633602778459]
state=green, forward, None, None, forward / action=[-0.5323083090181484,
-0.8532569316734895, -0.5784256704237616, -0.13999579801660333]
state=red, None, None, None, forward / action=[1.2149670406685205e-81,
-0.9999845302260466, -0.999997654948867, -0.3173418708436298]
state=red, right, None, None, forward / action=[0.2025562834200077, -0.6962011916876905,
-0.7049832518673989, -0.05515559254731506]
state=green, None, forward, None, left / action=[-0.8048914662073439, 0.2682475262319439,
4.805317155196665, -0.6372854859562564]
state=red, None, None, left, forward / action=[0.3237131575955968, 0.0755463733165751,
0.27484089899357733, 0.32936893086451835]
state=red, None, None, forward, right / action=[0.18338275461193132, 0.24561640498065063,
0.42524799652990564, 1.2645871811287521]
state=red, left, None, None, right / action=[-0.6858843394850931, 0.10552443238775444,
-0.5493040570522435, -0.8171971839508263]
state=red, right, None, None, right / action=[-0.8918526674534564, -0.9155866374490758,
0.4885657104471841, 0.3212398417676037]
state=green, None, forward, None, right / action=[0.2736097528866843,
-0.9041676644171459, 0.16700942818833223, -0.8778564758585987]
state=green, None, None, None, right / action=[-0.021663438855974826,
-0.20848758727363267, -0.1916905669160083, 2.116660990298857]
state=green, left, None, None, right / action=[0.3674121857655166, -0.004138907637892819,
0.5299317032973543, 1.5458526170954427]
state=red, None, right, None, forward / action=[-0.425418922843122, -0.39715920197637855,
```

```
-0.305406400582579, 0.056465021892847966]
```


```
Q Learning Agent done
```

Conclusion

We can see that the fully implemented agent has learned a set of rules governing road travel, including stop light behavior, and how to follow the route planner. Our agent quickly approaches an optimal policy state, with the caveat that it retains the ability(via epsilon settings) to learn changes. Since we have concentrated on 100 run trials,we will not see all possible states, and so can't reach a fully optimized policy, wherein the agent would follow all rules perfectly and always take the shortest path that those rules allow. We currently see behaviors, such as circling around a block, that maximize the total reward, but don't serve any other purpose. The possibility exists to save the policy and continue to learn, eventually having an entry for each possible state, and minimizing the mistakes. Until then, our agent is able to perform in a very acceptable manner for this environment.

EOF