
Reinforcement Learning for Autonomous Driving Obstacle Avoidance using LIDAR

Lucas Manuelli

Massachusetts Institute of Technology

MANUELLI@MIT.EDU

Pete Florence

Massachusetts Institute of Technology

PETEFLO@MIT.EDU

1. Introduction

In this project we evaluate the ability of several reinforcement learning methods to autonomously drive a car through a cluttered environment. We don't assume a map of the environment a priori but rather rely on our sensor (in this case a LIDAR) to inform control decisions. A standard approach to solving this problem would require building up a map of the environment, possibly using an occupancy grid, and then planning a collision free path through this estimated map using a planner such as RRT*. On the other hand one notes that simple output feedback controllers, such as Braitenberg controllers, can be quite effective for obstacle avoidance tasks. Guided by this example we want to see how reinforcement learning performs in learning an output feedback controller for obstacle avoidance.

Our initial goal was to implement a value-based learning method, and we were recommended to start with SARSA. After some initial encouraging results with SARSA, we decided implemented Q-learning and a continuous state-space version SARSA using function approximation. In addition we also tried policy search methods.

In short, there were distinct advantages and concerns found for the different approaches. The value-function-based approaches benefit greatly from their dynamic programming formulation, but it is also their limitation, particularly due to the difficulties of discretization. The policy-search-based approaches have some nice properties including that they are easy to "seed" with a favorable initialization, and do not need to be discretized, but do not benefit from Bellman's optimality principle (even if it does not hold, given the non-Markov observable state). In most cases, the ability of the reinforcement learning agents to improve their performance, even if not without "hand-holding" from us, is quite remarkable to watch. That said, significant effort and tuning is required, and in the end, we were not able to satisfiably outperform intuitive hand-designed controllers. The dynamic programming techniques had a ceiling of performance imposed by the discretization, and the policy search

methods were not able to improve upon essentially-perfect hand-designed controllers. Although this was the case for the study presented, it seems that in scenarios where intuition is less readily applied, and/or with more patient tuning of parameters, even better results may be obtained for the reinforcement learning controllers.

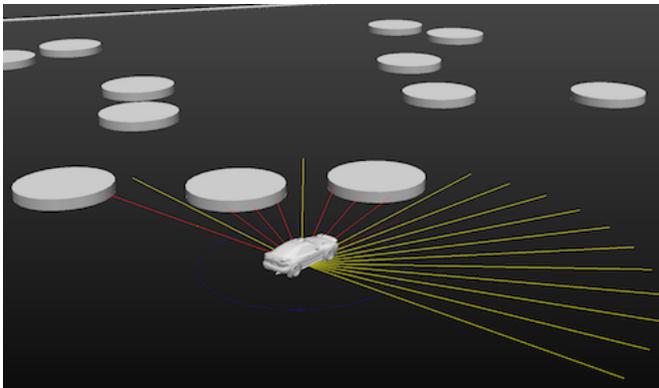


Figure 1. Screen capture from simulation setup: a car in a circular obstacle field with an array of point LIDAR measurements.

2. Experiment Setup and Simulation Environment

2.1. Car Model

For our car model we consider a simplified Dubin's car model. The location of the vehicle is described by (x, y, θ) , where (x, y) denote Cartesian position and θ denotes the orientation. For our purposes we suppose that we can control the derivative of the orientation, namely our control is $u = \dot{\theta}$. We also suppose a fixed speed v . Then the dynam-

ics of the vehicle are given by

$$\dot{x} = v \cos(\theta) \quad (1)$$

$$\dot{y} = v \sin(\theta) \quad (2)$$

$$\dot{\theta} = u \quad (3)$$

2.2. Sensor Model

For our sensor model we use a small array of point LIDARs. In particular, N beams are spread evenly over the field of view $[\theta_{min}, \theta_{max}]$. All experiments shown use $N = 20$ and $[\theta_{min} = -\pi, \theta_{max} = \pi]$. Each laser has a maximum range of d_{max} . The n^{th} laser then returns x_n , which is the distance to the first obstacle it encounters, or d_{max} if no obstacle is detected. Then one laser measurement can be summarized as $X = (x_1, \dots, x_N)$.

2.3. Obstacle Environment

Picking the right obstacle environment for our task required some trial-and-error. Obstacles that were too thin along one direction were not a good choice, as they could hide in between the “array of point LIDARs” as the robot approached. Thus circular obstacles were chosen, as they circumvented this problem. Picking the right density of obstacles and obstacle size was another trial-and-error process. A square border prevented the car from escaping. A listing of approximate parameters for the obstacle environments used is provided in Table 1. Generalizations across obstacle environments is outside the scope of this project.

Car parameters

Car velocity, v	16 m/s
Maximum turning rate, u_{max}	4 rad/s

Sensor parameters

Number of point LIDARs, N	20
Maximum laser range, d_{max}	10 m
Field of view $[\theta_{min}, \theta_{max}]$	$[-\pi, \pi]$

Obstacle environment parameters

Area of square world	250 m^2
Obstacle density	0.18 / m^2 (45 obstacles in 250 m^2)
Circular obstacle radius	1.75 m

Table 1. Simulation parameters used for the experiments, unless otherwise noted.

2.4. Simulation Environment

Our software stack is integrated with a robotics visualization tool called Director (Marion, 2015). We use the Director code for visualization and raycasting. This is the only external code that we use in addition to standard Python modules. Our simulation environment is written in Python. We have a main class called `Simulator` which combines together several other modules, e.g. `Car`, `Sensor`, `World`, `Controller`, `Reward`, `Sarsa`, etc, to construct a working simulation class. All of this code was

written by us for the project. As mentioned before, the only fundamental pieces of code that we didn’t write were the visualization tools and the raycasting method which we use for computing our sensor model. One nice feature of the Director application is that it uses Visualization Toolkit as the underlying graphics engine, and thus our ray-cast calls are actually in C++ and thus very efficient. We use a timestep of $dt = 0.05$ in all our experiments, and `scipy.integrate` is used for integrating forward the dynamics.

All of the code for this project is open-source and may be found at github.com/peteflorence/Machine-Learning-6.867-homework. We also strongly encourage readers to watch our YouTube video, linked from the GitHub repo, which displays the controllers in action, from the learning phase through to final learned controllers.

2.5. Braitenberg (Hand-Designed) Controller

As a baseline against which to compare our learned controllers, we used a simple hand-designed controller inspired by the controllers of Valentino Braitenberg. The most basic controller we tried simply counts the number of non-max sensor returns on the left ($n_{left} = \sum_i^{N/2} \mathbb{1}[x_i < d_{max}]$) and compares with the number of non-max sensor returns on the right ($n_{right} = \sum_{i=N/2+1}^N \mathbb{1}[x_i < d_{max}]$), and then selects the action to turn towards the direction $\min(n_{left}, n_{right})$. If no obstacles are seen ($n_{left} = n_{right} = 0$), then the controller selects to go straight. Thus the action space for this simple controller is $A = \{u_{max}, 0, -u_{max}\}$. This controller actually works surprisingly well.

The best Braitenberg-style controller we used is a slight improvement on the above version: it actually takes into the account the distances measured. This controller uses the squared inverse of each of the sensor measurements as its features: $\phi_B(x_i) = \frac{1}{x_i^2}$. The sum of these features is then computed for the left and right, and again the action is selected to turn towards the direction $\min(n_{left}, n_{right})$. An algorithm description is provided (Algorithm 1).

Algorithm 1 Braitenberg Squared Inverse Controller (BSIC)

Input: sensor data x_i , size N
Output: u from $\mathcal{A} = \{u_{max}, 0, -u_{max}\}$
 Initialize $n_{left}, n_{right} = 0$
for $i = 1$ **to** $N/2$ **do**
 $n_{left} = n_{left} + 1/(x_i)^2$
end for
for $i = N/2 + 1$ **to** N **do**
 $n_{right} = n_{right} + 1/(x_i)^2$
end for
if $n_{left} > n_{right}$ **then**
 $u = u_{max}$ (TURN LEFT)
else
 if $n_{left} < n_{right}$ **then**
 $u = -u_{max}$ (TURN RIGHT)
 else
 $u = 0$ (GO STRAIGHT)
 end if
end if

3. Dynamic Programming: SARSA and Q-Learning

Two of the most popular methods in reinforcement learning are SARSA and Q-Learning. These methods both aim to find the optimal policy of a dynamic programming problem. Thus the first step is to reformulate our problem as a dynamic programming problem.

3.1. Dynamic Programming Formulation

To formulate our problem as a dynamic programming problem we need to define a “state.” Let $S_c = (x, y, \theta)$ be the state of the car, and let $X = (x_1, \dots, x_n)$ be the sensor state. What we can observe is S_c and X . On the other hand the “true state” includes a complete description of the global map. If we don’t give our learning algorithm a prior map and only let it have measurements of S_c and X , then it is clear that the “true state” is partially observable. To make our problem similar to the limitations of a real car robot with just the described LIDAR sensor model, we only allow our reinforcement learning agent to access X , and so we consider $S = X$ (this is because the dynamics of the car are trivial here). Since this is not a state in the true sense of the word let us call it a “reward state” (really it’s just the observation). Next we need to define our action set. In principle our model has a continuous action set. However, for the purposes of SARSA and Q-Learning it is much easier to have a discrete action space. Thus we restrict ourselves to $a \in \{u_{max}, 0, -u_{max}\} = \mathcal{A}$, which correspond to left, straight, and right actions. Let θ_n be the angle of n^{th} laser where $\theta_n = 0$ corresponds to straight ahead. Now we need

to define the reward function. First define weights

$$w_n = C_0 + \cos(\theta_n) \quad (4)$$

Also define

$$\phi(x_n) = \begin{cases} \min(1/x_n, C_{max}) & x_n < d_{max} \\ 0 & x_n = d_{max} \end{cases} \quad (5)$$

Then ϕ has the effect of amplifying short distance measurements. Let $W = \frac{C_{raycast}}{\sum_n w_n} \cdot (w_1, \dots, w_N)$, $\phi(X) = (\phi(x_1), \dots, \phi(x_n))$. Then define the reward $R(S, a)$ as

$$R(S, a) = C_{action}|a| + \langle W, \phi(X) \rangle \quad (6)$$

where C_{action} is an action cost. Finally if the car is currently in collision with an obstacle, i.e. some laser x_n is reading less than d_{min} then we set $R(S, a) = C_{collision}$.

Now let us think about why this may be a reasonable reward function for our objective, which is obstacle avoidance. One should think of $C_{raycast}, C_{action}, C_{collision} < 0$ and $C_0, C_{max} > 0$. Then we see that the components of $\phi(X)$ increase as we get close to obstacles. And they increase as the inverse of the distance to the obstacle. Thus this is penalizing getting too close to obstacles. Now consider the weights W show in Figure 2.

This means that it’s worse to have obstacles in front of you than off to the side. The action cost is to encourage the car to drive straight rather than spin around in circles. Finally since we are interested in obstacle avoidance we impose a large penalty if the car does crash into an obstacle.

The reason we have $\phi(X)$ and the weights W is reward shaping. In reality all we care about is not crashing into obstacles, but it helps the algorithms to converge if the rewards “guide” them towards staying away from obstacles, e.g. if the reward is higher the further you are from the obstacles.

Now that we have the reward function we can formulate our problem as a dynamic program. The problem is to find the policy $\pi : S \rightarrow \mathcal{A}$ to solve

$$\max_{\pi} E_{\pi} \left[\sum_{t \geq 0} \gamma^t R(S_{t+1}, a_t) \right] \quad (7)$$

The expectation is over the reward states that result from taking actions according to policy π . Here $\gamma \in (0, 1)$ is a discount factor. In a standard dynamic programming framework the state S_t would be Markov. That is, future states depend only on the current state and future actions. In our case however our reward state is non-Markov since the map of the world is not included as part of our state.

3.2. SARSA(λ)

The difficulty with applying standard dynamic programming techniques to the current problem is that it is hard to write down the transition law $(S_t, a_t) \rightarrow S_{t+1}$ since this requires knowing how the sensor measurements will evolve if we take a particular control action. The SARSA technique allows us to circumvent this problem if we have access to runs/simulations of the true system. The Q-Values are defined as

$$q_\pi(S, a) = E_\pi \left[\sum_{k \geq 0} \gamma^k R(S_{t+k+1}, a_{t+k}) | S_t = s, A_t = a \right] \quad (8)$$

If we knew the Q-values we could choose the best policy as

$$a = \arg \max_{a'} q_\pi(s, a) \quad (9)$$

The ultimate goal of SARSA(λ) is to find the optimal policy π^* . Intuitively the approach is to estimate q_π and then slowly improve the policy π towards the optimal policy by using greedy policy selection based on the current q values. The algorithm is always estimating q_π for the current policy π and hence the q-values ultimately converge to q_{π^*} , and the greedy policy π also converge to π^* .

Algorithm 2 SARSA(λ)

```

Initialize  $Q(S, a)$  arbitrarily for  $S \in \mathcal{S}, a \in \mathcal{A}$ 
repeat
   $Z(S, a) = 0$  for all  $S \in \mathcal{S}, a \in \mathcal{A}$ 
  for each step of episode do
    Take action  $A$ , observe  $R, S'$ 
    Choose action  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.  $\epsilon$ -greedy)
     $A^* \leftarrow \arg \max_a Q(S', a)$  (if  $A'$  ties for max then  $A^* \leftarrow A'$ )
     $\delta \leftarrow R + \gamma(Q(S', A') - Q(S, A))$ 
     $Z(S, A) \leftarrow Z(S, A) + 1$ 
    for all  $s \in \mathcal{S}, a \in \mathcal{A}$  do
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta Z(s, a)$ 
      if  $A' = A^*$  then
         $Z(s, a) \leftarrow \gamma \lambda Z(s, a)$ 
      else
         $Z(s, a) \leftarrow 0$ 
      end if
    end for
   $S \leftarrow S', A \leftarrow A'$ 
end for
until end of episode
    
```

In order to apply SARSA in the form described above we need to discretize the state-action space. In section 3.4 we describe an approach that doesn't require discretizing the

state. In order for the SARSA algorithm to be effective without drastically increasing the training time, we must keep the size of the discretization relatively small. With this in mind we chose a discretization based on bins. The discretization is governed by three parameters. N_{inner} , the number of inner bins, N_{outer} the number of outer bins and β_{cutoff} , the cutoff distance (as a fraction of the maximum laser range) between inner and outer bins. A bin is defined by two angles θ_0, θ_1 and two distances d_0, d_1 . Then this bin is occupied if any laser whose angle lies in $[\theta_0, \theta_1]$ returns a distance in $[d_0, d_1]$. For example suppose $N_{inner} = 5$ and $N = 20$. Then the lasers corresponding to this bin are x_1, \dots, x_4 . This bin would be occupied if $x_i \in [d_{min}, \beta_{cutoff} * d_{max}]$ for some $i \in \{1, \dots, 4\}$. If a bin is occupied then it is labeled with a 1, if it is empty is is labeled with a zero. We have already discretized the action space into $\mathcal{A} = \{-4, 0, 4\}$. Thus the discretization of the state-action space lies in the space $\{0, 1\}^{N_{inner} + N_{outer}} \times \mathcal{A}$. Hence the size of the discretization is $2^{N_{inner} + N_{outer}} \times |\mathcal{A}|$.

3.2.1. RESULTS

There are many parameters that must be chosen in order to run the SARSA lambda algorithm. We discuss some of the most important here and give intuition on how we chose them. Then we will analyse how varying a few key parameters around our baseline setup affects performance.

Two of the most important parameters are the number of bins N_{inner}, N_{outer} in the discretization. You need enough bins that so that the state is informative enough to pass through relatively narrow gaps between obstacles. However adding more bins comes at the cost of increasing the size of the state space which can increase training time. In addition if a particular action-state pair is not visited sufficiently many times then that Q-value $Q(s, a)$ will not be a good approximation to the true Q-value, and hence control decisions taken based on that Q-value will not be optimal (and in fact will be quite poor usually). Through many experiments we found that $N_{inner} = 5, N_{outer} = 4$ provided good performance while keeping the size of the discretized state-action space relatively small at $2^9 * 3 = 1536$. Another set of parameters that had a large impact on performance were those of the cost function. It turned out to be important to balance the reward for staying away from obstacles with the penalty for using large control action, C_{action} . If C_{action} was too small the controller could end up learning to spin in circles, see Section 3.2.2 for more discussion of this problem. Alternatively if C_{action} was too high the behavior would bias too much towards driving straight and wouldn't do enough to avoid obstacles. Another important parameter was the step size α in the SARSA update. It needs to be small enough so that the Q-Value estimates don't diverge. However, if it is too small then the Q-values are updated only a small amount in each

iteration and convergence would take a long time. For the discrete state-action case we found that $\alpha = 0.2$ worked well, although it is difficult to evaluate what the “correct” step size is. See Table 2 for our preferred set of default parameters. First we provide a qualitative discussion of the results of the Q-Learning using parameters from Table 2. We initialize all of the Q-values to zero and run the SARSA algorithm for 8000 seconds. Since our simulation timestep is $dt = 0.05$ this amounts to $8000/dt = 160,000$ iterations of the SARSA update. Our simulation runs at approximately 700 Hz. Since it takes 20 ticks of the simulator to complete one second this amounts to a simulation of $35\times$ normal speed. Thus this training takes approximately 3.8 minutes. The controller that we get out of this performs reasonably well. Specifically it manages to drive around the obstacle field shown in Figure 3 at a reasonably high speed with mean time between crashes of approximately 30 seconds. One thing to note is that this is much worse than the performance of our default controller described in section 2.5. A comparison of performance against the default controller is given in Figure 6. It is apparent the the default controller provides highly superior performance in run duration, and to a lesser extent discounted reward. At the given car speed in this run the default controller can essentially drive around indefinitely without crashing. However, it’s behavior is to make a hard turn as soon as it detects an obstacle. Thus it has a tendency to get trapped within an area just going in circles. The default controller always turns as soon as it sees anything, and even if it goes through a gap, will do so by constantly alternating between left and right control actions. On the other hand since our reward function penalizes taking control actions (by C_{action}) it rewards our controller for going straight. Thus the learned controller will continue to go straight in some situations where we are detecting obstacles, but they are not in our direct path as in Figure 4. This is an interesting behavior that is learned. Thus qualitatively our learned controller turns much less than the default controller and hence doesn’t tend to get stuck circling in one spot. Another interesting feature of the controller is that learns how to drive through narrow gaps such as in Figure 5. For a more quantitative analysis we can consider the duration and discounted reward over time (see Figure 7). The upper plot shows that as time progresses the average duration of an episode increases. Similarly the lower plot show that as time progresses the algorithm also improves in terms of discounted reward. This clearly shows how the learning algorithm improves its performance over time, as is to be expected.

3.2.2. FAILURE MODES

Occasionally our learned controller would end up spinning us in circles, quite literally. Here we try to un-

derstand why this might have happened. Let $X_{max} = (d_{max}, \dots, d_{max})$ which corresponds to a laser measurement where no obstacles are detected. Then what is happening is that $Q(X, 0) < \max_{a \in \{-u_{max}, u_{max}\}} Q(X, a)$. Thus we end up turning a given direction (suppose it is left without loss of generality) even when we don’t see an obstacle. The reason this actually works quite well is that in our randomly generated maps there tends to be pockets of space that are obstacle free. If the vehicle happens to enter or be initialized into one of these “obstacle free” zones then by aggressively turning a given direction it can keep itself inside the “obstacle free” zone and avoid ever running into something. Although this controller actually produces a high reward (since it never crashes) it is not what one would consider a “good” controller. There are several reasons that SARSA may end up learning this behavior. The most important is that our reward-state is not Markov. This means that when we are in reward state X_{max} we don’t know our transition probabilities to the next reward state. This is because we have only local information in the reward-state and have no information about the global map. This is an example of our learning algorithm exploiting structure in the problem that we didn’t mean for it to take advantage of.

A possible solution to this problem is to redesign the reward function. We could include reaching a goal as part of the reward function, so then the learning algorithm would have an incentive to make progress towards this goal rather than spin in circles. Another solution approach is to increase the number of obstacles in the world to eliminate these “obstacle free zones.”

3.2.3. ANALYSIS

In this section we perform two experiments to further characterize the performance of our SARSA algorithm. In the first experiment we hold all the parameters fixed except λ . The λ parameter affects the eligibility traces and controls the balance between a TD (temporal difference) update and an MC (Monte Carlo update). At the two extremes, $\lambda = 0$ corresponds to a pure TD update and $\lambda = 1$ corresponds to a pure MC update. Intuitively TD updates rely heavily on the Markov property and the Bellman equation for the Q-Values. On the other hand MC updates simply try to estimate $Q(S, a)$ using the empirically observed rewards that followed the state action pair (S, a) and don’t rely on the Markov property or the Bellman equation. It is known (Ch. 7 of (Sutton & Barto, 1998)) that $\lambda > 0$ can help convergence speed and may improve performance in non-Markov domains. In Figure 9 we see a plot of performance data for several different λ values. Since each run takes 5 minutes to train we are limited in the number of simulations we can perform. Overall the trend seems to be that higher lambda values provide better performance. Since our observation state is highly non-Markov this coincides with the

fact that SARSA(λ) with λ close to 1 is less sensitive to violations of the Markov property since each update step is more similar to a MC update.

Another interesting fact that we noticed is that, even with the same set of parameters, different runs of the learning algorithm can produce quite varied performance. Figure 10 shows several runs of the learning algorithm using the same set of parameters. Out of the 10 trials, runs 5 and 9 stand out. They have extremely long average run durations which are an order of magnitude larger than all the rest. Investigating these runs more carefully they both succumbed to the “drive in circles” failure mode. Thus our learning algorithm is sensitive to this failure mode, with about a 20% failure rate. We believe that with a more carefully designed reward function we could take care of this failure mode.

3.3. Watkins Q-Learning

Q-Learning is almost the same as SARSA but with a slight variation in the update. While SARSA is an on-policy learning method (that is we are learning the Q-value for the current policy), Q-Learning is an off-policy method (we are learning the Q-Values under the optimal policy (even though we not currently controlling according that policy)).

Algorithm 3 Watkins Q-Learning(λ)

```

Initialize  $Q(S, a)$  arbitrarily for  $S \in \mathcal{S}, a \in \mathcal{A}$ 
repeat
   $Z(S, a) = 0$  for all  $S \in \mathcal{S}, a \in \mathcal{A}$ 
  for each step of episode do
    Take action  $A$ , observe  $R, S'$ 
    Choose action  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.  $\epsilon$ -greedy)
     $A^* \leftarrow \arg \max_a Q(S', a)$  (if  $A'$  ties for max then  $A^* \leftarrow A'$ )
     $\delta \leftarrow R + \gamma(Q(S', A^*) - Q(S, A))$ 
     $Z(S, A) \leftarrow Z(S, A) + 1$ 
    for all  $s \in \mathcal{S}, a \in \mathcal{A}$  do
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta Z(s, a)$ 
      if  $A' = A^*$  then
         $Z(s, a) \leftarrow \gamma \lambda Z(s, a)$ 
      else
         $Z(s, a) \leftarrow 0$ 
      end if
    end for
     $S \leftarrow S', A \leftarrow A'$ 
  end for
until end of episode
    
```

3.3.1. RESULTS

The same preferred parameter set for SARSA given in Table 2 also works well for Q-Learning. The resulting

controller exhibits similar qualitative performance to the SARSA controller described above and the plot of performance over time is almost identical to Figure 7 so we omit it here. We do however repeat the two experiments from section 3.2.3. The first is to vary the λ parameter while holding everything else fixed. The results are shown in Figure 11 and are very similar to those for SARSA. There seems to be a weak trend of increasing performance as we increase λ . Again this makes sense since a higher λ corresponds to less reliance on the Markov and Bellman properties (which we know are violated in this case). Unfortunately Q-Learning does not resolve the “driving in circles” failure mode, and possibly is even more sensitive to it. It does happen occasionally and for essentially the same reasons as in SARSA. We repeated the experiment from section 3.2.3 that performed multiple runs with the same parameter set. The results are shown in Figure 12. In this case you can see that runs 2, 6, 9 have an order of magnitude longer duration than the rest. These are exactly the runs where we ran into the “drive in circles” failure mode. Thus for Q-Learning the failure rate was 30%, which is slightly higher than the 20% for SARSA.

Ultimately there wasn’t too much difference in the performance of SARSA versus Q-Learning. Looking closely at the duration and reward in Figures 9 and 11 the performance difference between SARSA and Q-Learning is relatively negligible.

3.4. SARSA with function approximation

One of the major drawbacks of the methods presented in sections 3.2 and 3.3 is that they require discretizing the state-action space. In reality however our state is really continuous since the laser measurements lie in \mathbb{R}_+^N . An alternative to discretizing the state space is to use function approximation to approximate the Q-Values. As described in Chapter 9 of Sutton and Barto (Sutton & Barto, 1998) this still requires a discrete action space. We consider a linear function approximation to the Q-Values given by

$$Q(S, a, \mathbf{w}) = w_a(0) + \sum_{n=1}^N w_a(n) \phi(x_n) \quad (10)$$

where \mathbf{w} is the matrix of weights. We think this is a reasonable approximation to the Q-values since it has almost the same form as the reward function. We include an intercept term $w_a(0)$ to capture the fact that even when the feature vector $\phi(X)$ is identically zero we prefer action $a = 0$ to actions $a = \{-u_{max}, u_{max}\}$. Given the above form, we implemented gradient descent SARSA(λ).

Algorithm 4 Gradient Descent SARSA(λ)

```

Initialize weights  $\mathbf{w}$  to 0.
repeat
   $\mathbf{z} = 0$ 
   $S, A \leftarrow$  initial state and action of episode
  for each step of episode do
    Take action  $A$ , observe reward  $R$  and next state  $S'$ 
     $A' \leftarrow$  policy derived from Q-values at state  $S'$  (using  $\epsilon$ -greedy)
     $\delta \leftarrow R + \gamma Q(S', A', \mathbf{w}) - Q(S, A, \mathbf{w})$ 
     $\mathbf{w}_{old} \leftarrow \mathbf{w}$ 
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$ 
     $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla_{\mathbf{w}} Q(S, A, \mathbf{w}_{old})$ 
     $S \leftarrow S', A \leftarrow A'$ 
  end for
until end of episode
    
```

Intuitively this corresponds to adjusting the weights $w_a(n)$ to get the best approximation to the Q-Values in a least squares sense (see Sutton-Sarto Ch. 9 (Sutton & Barto, 1998) for a precise statement).

3.4.1. RESULTS

It is notoriously difficult to get function approximation methods for SARSA to perform well. Performance fundamentally depends on whether the chosen parametric form, given by (10), provides a good approximation of the Q-Values. The first thing we noticed was that a much smaller step-size α was needed. After experimenting we found that $\alpha = 1e - 4$ was small enough that the weights wouldn't diverge to ∞ . For the training in sections 3.2 and 3.3 we initialized the Q-Values to zero. The analog in this case corresponds to setting the weights $w_a(n) = 0$. This approach didn't yield very good results for the function approximation SARSA method. An alternative, as outlined in (Smart & Kaelbling, 2000), is to help the learning algorithms perform better by running a known controller to help learn some baseline Q-Values. Essentially during a supervised learning period we are learning weights $\{w_a(n)\}$ which approximate the Q-Values under the known policy $\pi_{default}$. During this phase we are performing the weight updates from Algorithm 4 but the policy is chosen according to $\pi_{default}$ rather than the epsilon greedy policy. In this setup we run this supervised training for a while before switching to the standard SARSA(λ) updates specified in Algorithm 4. If we use the standard environment used in sections 3.2 and 3.3 we end up falling into the "drive in circles" failure mode outline in 3.2.2. Increasing the obstacle density to eliminate the "obstacle free" zones resolves this problem and we are able to get a functioning controller. The controller performs reasonably well, but still considerably worse than the Q-Learning controller on the same en-

vironment. The interesting thing is to attempt to understand the weights $\{w_a(n)\}$ that we are learning and why they might be working. The weights are plotted in Figure 8. The actions $\{4, 0, -4\}$ correspond to Left, Straight, and Right, respectively. Also the laser measurements go from -10 to 10 with 0 corresponding to straight ahead. Thus x_{-10} is our leftmost laser measurement. Not shown are the intercept terms $w_a(0)$. The greedy controller that can be derived from the Q-Values is $\pi(S) = \arg \max_{a'} Q(S, a')$. In other words we should choose the action that maximizes the Q-Value for the current state. Suppose there is an obstacle to our right, then the feature vector $\phi(X)$ will have $\phi(x_n)$ small for $n < 0$ (i.e. range measurements to our left are large) and $\phi(x_n)$ large for $n > 0$, i.e. measurements to our right where the obstacle is. Since the weights $w_4(n)$ corresponding to action $a = 4$, which is Left, are mostly positive for $n > 00$, and since the only components of $\phi(X)$ which are large are those where the obstacle is (i.e. $n > 0$) we get that $w_4(n)\phi(X)$ is moderately positive. On the other hand $w_{-4}(n)$ is exactly the opposite. For $n > 0$ we mostly have $w_{-4}(n) < 0$ and so $w_{-4}(n)\phi(X)$ will most likely be negative. Thus we have that $Q(S, 4) > Q(S, -4)$. So we prefer to go left rather than right. A similar logic applies to understanding why $Q(S, -4)$ could be larger than $Q(S, 0)$. Thus in this case we would choose left.

The above exposition gets the general idea across. However, as can be seen in Figure 8 the actual weights, while exhibiting some general trends, are quite messy and hard to parse.

4. Policy Search

As an alternative method to the value-function-based approaches presented in Section 3, we also investigated the use of policy search methods. The key difference with policy search approaches versus value-function-based approaches is that they are not based on dynamic programming, and instead are local optimization methods of a parametric policy. The basic idea is very simple: if we can represent the behavior of the robot parametrically, then we can just vary the parameters and attempt to optimize the total reward for the robot.

A number of policy search methods were investigated. Some successful results were obtained, but as with the difficulties encountered for function approximation, in general the learned weights are messy and performance was somewhat inconsistent: sometimes significant improvements in performance were observed during the course of training, but at times the researcher is led to believe that luck is entirely too dominant. Empirically, it was found that the variant of Episodic REINFORCE as presented in the following subsection performed the best on the given obstacle environments. A second approach, using a Beta distribu-

tion, is discussed as well, and although it did not produce universally successful obstacle avoidance maneuvers, it is nonetheless interesting to analyze.

It is interesting to note the list of differences with the methods of the last section. For one, policy search is able to handle continuous state and action spaces without discretization. The function approximation approach discussed in Section 3.4 is able to consume a continuous state space, but still is limited to a discrete action space, whereas policy search methods may be continuous in both. Also in contrast with the dynamic programming methods, these gradient-descent-based policy search methods are inherently local search, and so we have no guarantees on finding global maxima even if we had access to a full Markov state. The methods investigated all used at least 10 parameters, and so it is very likely that local maxima were abundant in these high-dimensional spaces. Stochasticity in gradient descent may have contributed to escaping local maxima, but it is difficult to say. It is also useful to note that the policy search methods discussed were all episodic in nature – i.e., there was no TD (temporal difference) analog used here for policy search. Although such methods exist, it was easier in initial investigations to focus on the episodic versions.

4.1. Episodic REINFORCE with “logistic” control function

Many different variants of gradient-descent-based optimization of a parametric policy were investigated. The following was the algorithm with which the most empirical success was achieved. It has been able to repeatedly find parameters that seem capable of developing smart behavior, and in particular is able to increase, stochastically, the average duration which the robot is able to avoid crashing. It is different in a few ways from what might be considered a standard REINFORCE algorithm, and the reasons for these differences will be discussed. The parametric form of the policy (for commanding the steering rate u) was chosen to take the following continuous form:

$$u = 2u_{max} \left(\sigma(\theta^T \phi_B(X)) - \frac{1}{2} \right) \quad (11)$$

where σ is the logistic sigmoid function, and the features $\phi_B(x_i) = \frac{1}{x_i^2}$ are the same as for the Braitenberg vehicle. (Empirically, these were found to work well. Squaring the inverse serves to increase the emphasis even more on close obstacles.) The logistic sigmoid function is pushed down by $1/2$ so that it is an odd function, and is then scaled by $2u_{max}$. Note that a bias term has purposely been left out, because it would only serve to bias left/right turning, whereas our desired controller should be symmetric. We also note that the above is a deterministic function of the parameters, so that the stochasticity of the policy is on the parameters rather than the output. The REINFORCE up-

date is adapted from Roberts et al. (Roberts et al., 2011):

$$\theta_{i+1} = \theta_i + \eta(R_{total}(\theta_i + \theta_{p_i}) - R_{total}(\theta_i))\theta_{p_i}, \quad (12)$$

where θ_{p_i} is a sampled zero-mean Gaussian perturbation, and the update is such that if total reward R_{total} is increased due to the perturbation, then the parameter update will move in that direction. We note from Roberts et al. that this matches the form of the gradient update for REINFORCE since $\frac{\partial}{\partial \theta_i} \ln(p(\theta_{p_i})) \propto (\phi'_i - \phi)$, and the scalar may be absorbed into the learning rate, η , term. After exploring different options, the same reward function as in Section 3 was used (a combination of collision penalty, action cost, and large-sensor reward). Although we have $N = 20$ parameters in the vector θ in Equation 11, we recognize that the ideal behavior of the vehicle is symmetric, and so we reduce our model to just 10 parameters such that $\theta_{N-i} = -\theta_i$ for $i = 1, 2, \dots, \frac{N}{2}$. Also rather than sampling from a true zero-mean multivariate Gaussian, we take a page from Kiefer-Wolfowitz’s finite-difference method and only perturb one parameter at a time. This was found to be preferable since the contribution of one of the point LIDARs to the control action could be evaluated in isolation. The implementation is to randomly select one of the 10 indices on each iteration, and perturb that one parameter by a zero-mean single-variate Gaussian with variance σ_p^2 . This may break the algorithm’s exact resemblance of REINFORCE, but it is clearly very similar. We note that this formulation is essentially stochastic gradient descent using finite differences. An algorithm description is provided below.

Algorithm 5 Variant of Episodic REINFORCE inspired by Roberts et al.

Initialize policy parameterization $\theta_i = \theta_0$

repeat

 compute R_{total} for ∞ -horizon roll-out with $u(\theta_i)$

 randomly select index j for one of θ_i in θ_{left}

$\theta_{p_i} =$ zero vector, same length as θ

$\theta_{p_i}[j] = \mathcal{N}(0, \sigma_p^2)$

 compute R'_{total} for ∞ -horizon roll-out with $u(\theta_i + \theta_{p_i})$

 update policy:

$$\theta_{i+1} = \theta_i + \eta(R_{total} - R'_{total})\theta_{p_i}$$

until end of training time

4.1.1. RESULTS

Three scenarios for the performance of policy search are discussed: (i) starting from a zero-vector θ , (iii) starting from hand-chosen parameters that already give near-perfect performance, and (ii) starting from hand-chosen parameters that roughly work well already.

Before we analyze each of the three scenarios, though, we note observations that applied in all cases. For one, because the sigmoid function $\sigma(x)$ is very flat for $|x| \gg 0$, we expected it to be difficult to effectively change parameters usefully unless $\theta^T \phi_B(X)$ was close to 0. Thus we expected the algorithm to be able to successfully descend away from a parameter initialization of all zeros (as in case (i)), but did not expect to as easily demonstrate unambiguous improvement for already $|\theta^T \phi_B(X)| \gg 0$ policies. Indeed, that is the case we observed in case (ii).

(i) For starting with $\theta_0 = \mathbf{0}$, a significant improvement in controller performance was able to be learned with a surprisingly small amount of training time, as is shown in Figure 13. Only 1-2 minutes of simulation (approximately 30-60 minutes of “car” time) were required to produce controllers that were capable of surviving in the obstacle field for 400 or more time steps (20 seconds “car” time). To be clear, θ as a zero vector corresponds to ignoring all sensor measurements and going perfectly straight forward at every time step. This is clearly not optimal controller performance. In practice when starting from zeroed initial weights, the controller learns by having a large enough perturbation θ_{p_i} in the correct direction to be able to dodge an obstacle that it wasn’t able to before. Obviously when starting from the initial zero vector, it helps to use significantly large variance σ_p^2 for the perturbation sampling such that something actually capable of causing the car to dodge the “straight into brick wall” scenario. In practice it seemed that σ_p^2 could be increased as long as η , the learning rate, was decreased further to compensate. In addition to providing satisfactory performance, the final weights (Figure 14) make sense: for obstacles that are in front of the vehicle the $\theta_{left,i}$ for $i = 9, 8, \dots$, a strong turn away from the obstacle is preferred (which corresponds to negative weights, for θ_{left} (We remind the reader that the weights are mirrored, so we only need to consider the left weights).

In general the weights θ_{left} found by the policy search were more strongly negative. In closer detail, it is interesting to note some surprising parameter settings are found by policy search. For example, having small negative parameters for the left lasers out in front of you, and one very large parameter for out to the left side, would produce interesting behavior. With this controller, as the robot approached obstacles from afar, it would only be slightly turning away from them. This has the benefit of not incurring a large action cost and allowing the robot to drive more straight. When the robot comes very close to the obstacle, though, its measurement for out to the side will kick in, and it will very sharply turn away from the obstacle. Since the other weights were in the right direction but small, this prevented the robot from actually moving towards an obstacle head on, and so the “out to the side” measurement was able to help kick it away from the obstacle.

Another unexpected behavior was that for the weights for the lasers pointing out to the sides of the car, having them be positive in θ_{left} could actually give qualitatively better controller performance. The policy search parameters shown in Figure 14 show this characteristic. With a significant positive weight for example for the farthest left lasers x_1, x_2 , the the car would turn slightly around obstacles after it passed them. This had the effect of helping the car not get stuck in the square boundary corners of the world. It turns out that this is a key strategy for the obstacle environments that were used. It also had the capability of causing ‘circling’. As discussed before with the DP approaches, a cost on action helped negate the circling behavior. This did, however, unsurprisingly make it much harder to learn “dodging the brick wall” for the initially-zeroed parameters case. To verify, most learned controllers, including the one in Figures 13 and 14, did not produce circling behavior but instead produced desirable ‘dodge obstacles’ behavior.

(ii) We also tried initializing the policy with parameters that already gave impressive obstacle-avoidance performance. This is indeed an ideal property of policy search: it is easy to formulate “seeding” the policy search with a policy that is known or expected from other means to already be a good policy. This may come, for example, from trajectory optimization techniques, but for us this ideal “seeded” policy was designed by hand. It was not hard to think about the 10 θ parameters and design ones that would work very well. A sensible setting of the parameters was given by $\theta_{left,ideal} = [-10, -10, -10, -10, -10, -100, -100, -100, -100, -100]$, and θ_{right} was imposed to be $-\theta_{left}^R$ as discussed. These settings cause sharp turning motions away from obstacles out in front of the vehicle, and only slightly turning away from the obstacles out to the sides. The smaller weights for the lasers pointing out to the sides cause less erratic turning when driving parallel to a wall or obstacles, and because there is an action cost on the control input, this seems to positively impact the total reward of a roll-out. Indeed, this $\theta_{left,ideal}$, with zero optimization, was found to give a controller that did not crash on many environments. To increase the challenge, the obstacle density and the car’s velocity were increased. It was difficult, however, to find a setting where the car still “had a chance”, but just was not making the best control decisions. In other words, these initial parameters are near perfect.

In the end, at least with the parameters attempted, the ability of the policy gradient descent to “improve” this already-stellar policy seemed tenuous. Settings of σ_p^2 and η either seemed to produce θ that would barely change from the initial condition, or if the updates were too large, would diverge randomly. This seems plausible if the controller’s already stellar performance represents a local maxima. Additionally, as mentioned the effects of perturbations are ex-

pected to be small when the input to the sigmoid already contains values far from zero, as is the case with these large initial parameters. It is possible that this is primarily a result of using the sigmoid function. In short, if the policy is already really good, then policy search may not surprisingly have a difficult job of improving performance.

(iii) Finally, we investigate a third scenario, where the initial parameters are not stellar, but they are also not zero. For this purpose, $\theta_{left,okay} = [-1, -1, -1, -1, -1, -10, -10, -10, -10, -10]$ sufficed. This is similar to the parameters just discussed, but with less strong turning. In this case, the policy gradient descent was able to find improved parameterizations, as shown in Figure 15, although of course the increase in performance was not as drastic as it was for the zeroed initial parameters. The parameters found by policy search (Figure 16) were sensibly more negative than the initial parameters. It is interesting to note that although the weights appear to be very “noisy” from θ_i to θ_{i+1} , if we think about this, it is not too surprising. Having a sequence of three weights be $[-10, -40, -10]$ is not all that different from $[-20, -20, -20]$ since in these obstacle environments, if one laser is intersecting an obstacle, then the chances are that its neighbor is as well.

4.2. Beta distribution

As an alternative option, a second formulation of the policy parameterization was used. The largest difference here was that the policy is actually probabilistic, even given a specific parameterization. (In the previous subsection, the controller was only stochastic due to a probability distribution on the parameters.) It was hypothesized that a beta distribution (scaled by u_{max}) might offer a useful way to encode a probability distribution over $[-u_{max}, u_{max}]$. In particular the formulation was:

$$u \sim 2u_{max}(\text{beta}(a, b) - 1/2) \quad (13)$$

$$a = \theta_a^T \phi_B(X) + \theta_{a,0} \quad (14)$$

$$b = \theta_b^T \phi_B(X) + \theta_{b,0} \quad (15)$$

Whereas the previous formulation has 20 parameters (only 10 independent due to symmetry), this formulation has 42 parameters, with $\theta_b = \theta_a^R$ reducing this to 21 by symmetry.

With some thought, one can design parameters for this controller that seem to make sense. As with all the other controllers studied, if all of the weights for the lasers to one side of the car has the same, appropriate sign, then performance will generally be good.

A detailed case-by-case for all cases of interest are omitted for brevity, but the failure mode is particularly interesting and so we will highlight it here. The case that is difficult for this controller to handle is where the robot

is headed “straight towards a brick wall”. Although this might sensibly seem to correspond well to a case such as $\text{beta}(0.01, 0.01)$ where there will be pdf density for “either turning left or right, but not straight,” this does not work because the robot does not “commit” to either side, and so actually ends up going straight (alternating left and right) into the wall. Having very large weights that correspond to “always turn as soon as I see something” is one way around this problem, but this gives very jittery, non-ideal performance.

5. Conclusion

In conclusion, we have investigated a number of RL methods for learning controllers for our obstacle-avoidance task. To summarize briefly, we highlight again the most notable differences between the methods.

The dynamic programming methods were able to learn controllers that would improve performance with training time, as expected. Even though the “state” of the robot is highly non-Markov, the value function estimation seems to still work well, as evidenced by the controller performance. Discretizing the state and action space, even for this small toy robot, was a limitation that we expect prevented the robot from achieving optimal performance given the robot’s limitations. Both SARSA and Q-Learning worked well in the discretized domain, except for occasional failure modes that were described in section 3.2.2. Ultimately there wasn’t much difference in performance between SARSA and Q-Learning. Function approximation was investigated in order to avoid discretizing the state space. The function approximation method was more sensitive to divergence and failure modes, but with careful parameter choices could be made to work.

The policy search methods offered a different set of advantages and disadvantages. One clear advantage is that the robot does not have to discretize either the state or action space: both can be continuous. This allows the robot to make control decisions based on much richer information, and execute smoother maneuvers. Another advantage is that it was very easy to seed the policy parameterization with a highly functional policy. Policy search is also known, because it avoids the curse of dimensionality from discretization, to scale to higher dimensional systems better. Although “not dynamic programming” is an advantage of the policy search methods, it is also a major disadvantage. The search through policy space has no guarantees other than to find local optima.

While the results of some of the learned controllers were impressive, they were also humbling. In this game of human design versus reinforcement learning agent, the game was tipped in favor of human design. The controllers de-

signed by hand worked significantly better with significantly less effort: the value function controllers were not able to beat the performance of the Braitenberg controller, and the policy search did not significantly improve upon a parameterization that was chosen by design in a couple of minutes.

In theory there seems to be no reason, for example, that given sufficiently small step sizes, and sufficient random initializations, that a policy search method can't verifiably improve the performance of a hand-designed parameterization. In practice, though, this is not that easy!

6. Division of Labor

Lucas implemented the value function based controllers (SARSA, and Q-Learning, both discrete and function approximation). He also helped build out the simulation environment, and wrote the utility methods for logging, playback and plot generation. Pete built out the library of object-oriented modules and pulled together the simulation environment, designed the Braitenberg controllers, and implemented the policy search methods. The overall simulation architecture was designed together.

7. Acknowledgements

The authors would like to thank Professor Leslie Kaelbling for her enthusiasm, for helping us with ideas, and for supporting us in developing the project. We also thank Pat Marion for his great help in developing the visualization environment.

References

- Marion, Pat. Director: A robotics interface and visualization framework, 2015. URL <http://github.com/RobotLocomotion/director>.
- Roberts, John, Manchester, Ian, and Tedrake, Russ. Feedback controller parameterizations for reinforcement learning. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2011.
- Smart, William D. and Kaelbling, Leslie Pack. Practical reinforcement learning in continuous spaces. pp. 903–910. Morgan Kaufmann, 2000.
- Sutton, R.S. and Barto, A.G. *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998.

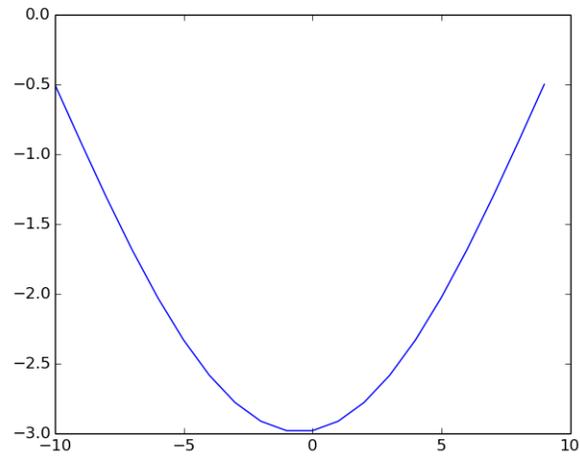


Figure 2. The weights W for computing the reward.

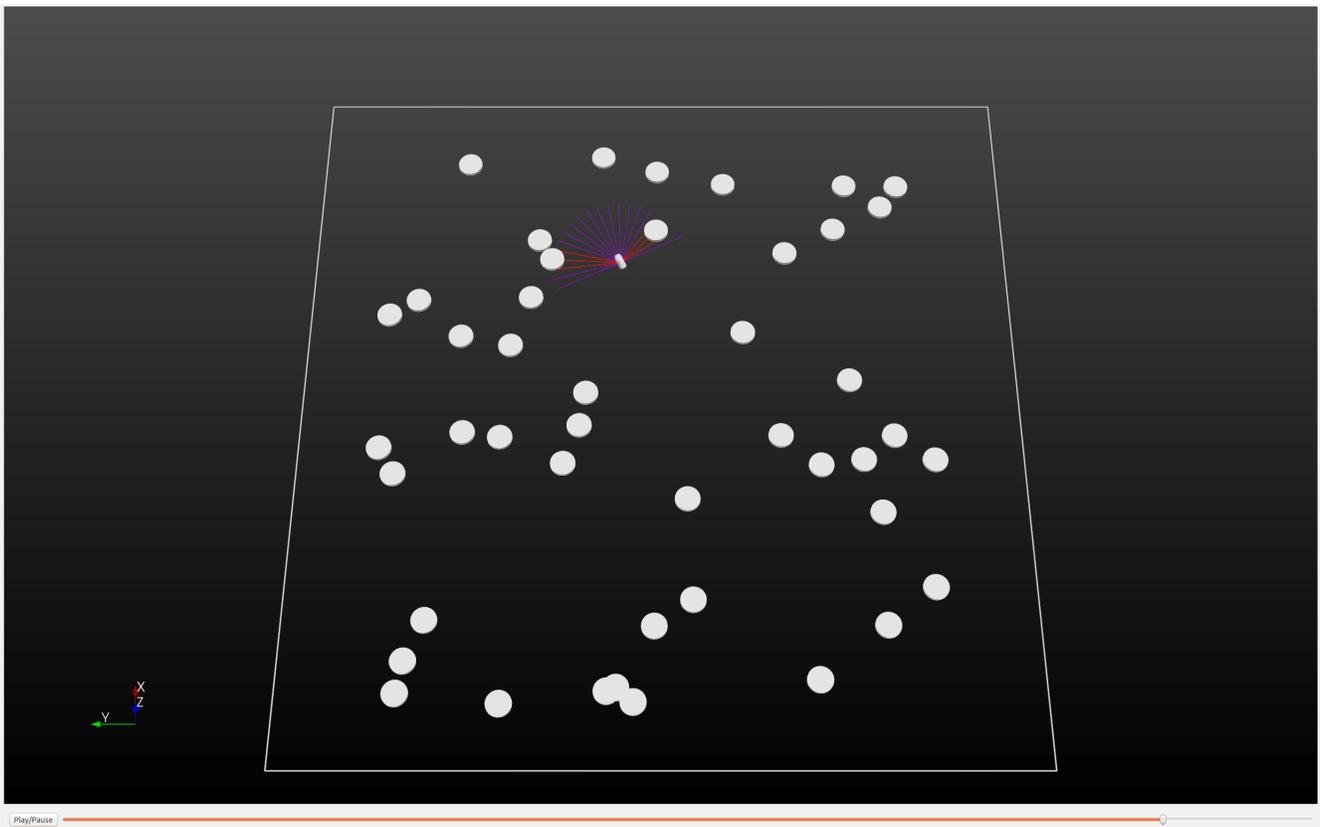


Figure 3. The obstacle field used for the majority of the learning runs

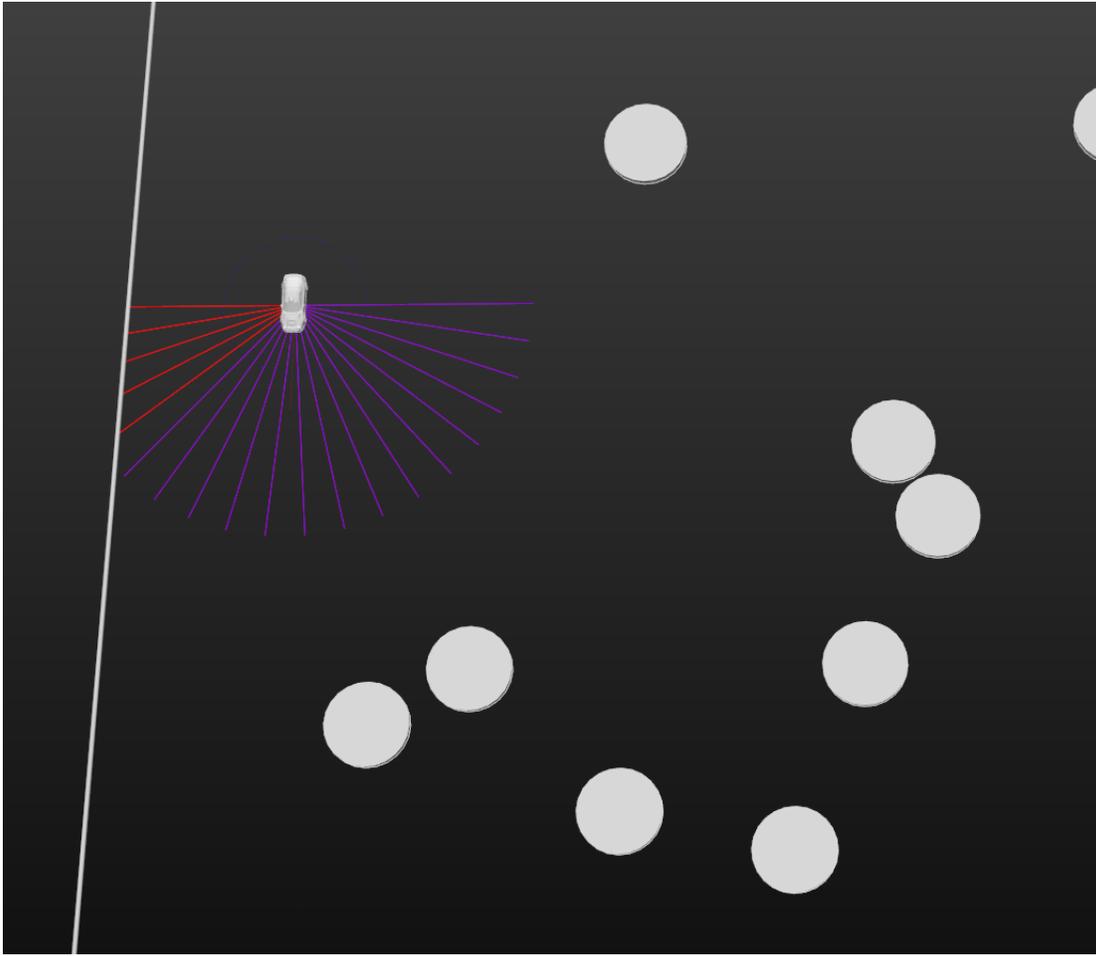


Figure 4. A situation where the SARSA controller chooses the straight control action, while the default controller would choose a hard turn to the left.

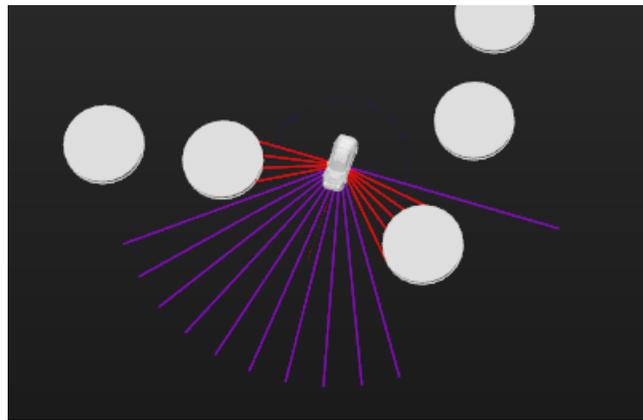


Figure 5. An example situation where the SARSA controller would drive through the gap.

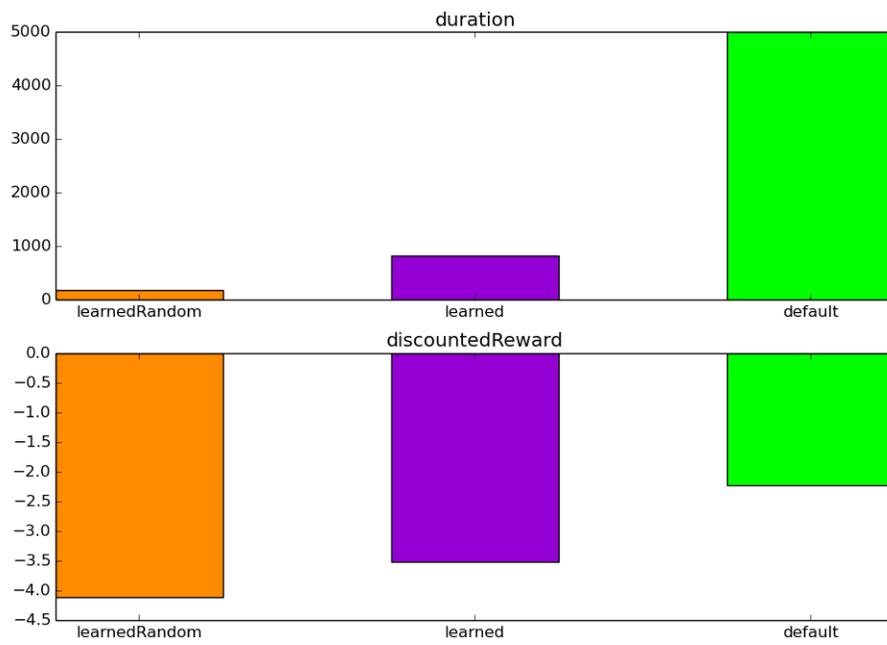


Figure 6. Performance of three different controllers using the parameters in Table 2. The duration is in simulator ticks, there are 20 ticks/second. learnedRandom corresponds to the SARSA controller during the learning phase where we are still randomizing the policy. learned is the actual SARSA with no randomization. And default is the Braitenberg controller from Algorithm 1.

RL for Obstacle Avoidance

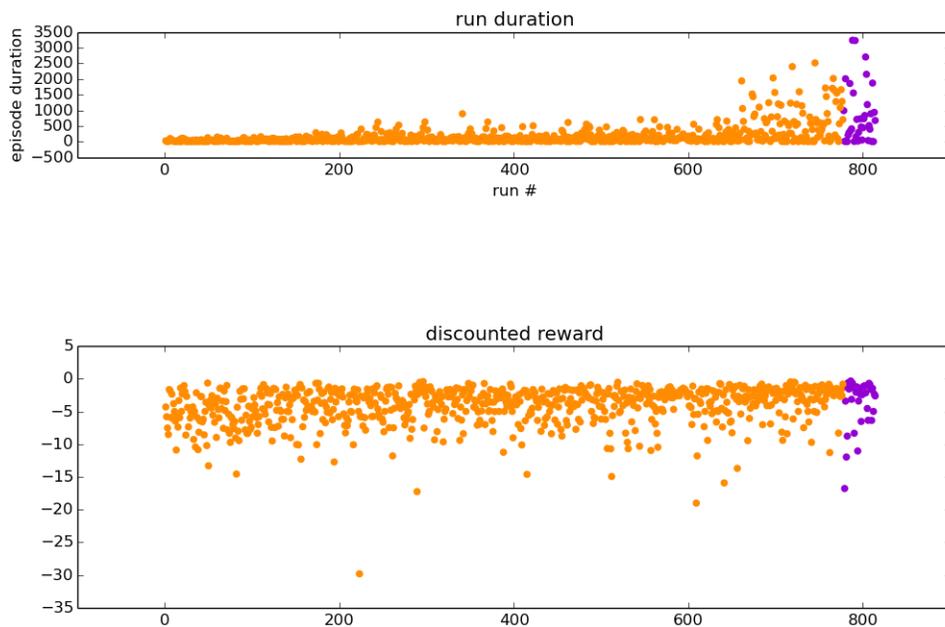


Figure 7. Performance over time of the SARSA learning controllers using parameters in Table 2. The duration is in simulator ticks, there are 20 ticks/second. The orange points correspond to the learning phase where the SARSA controller is using the ϵ -greedy control policy. Purple corresponds to the greedy control policy with no randomization.

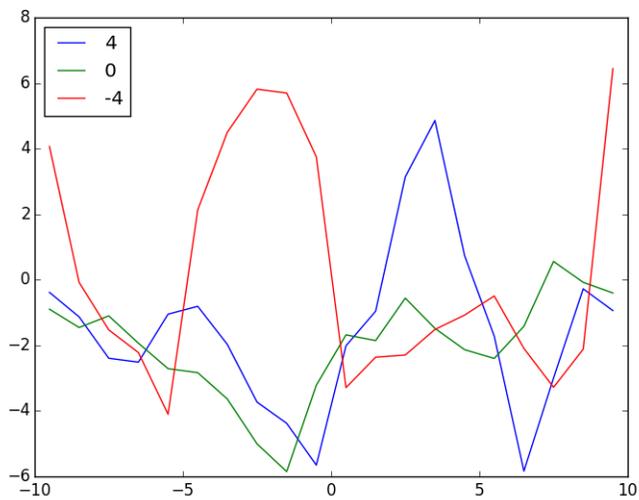


Figure 8. The weights $w_a(n)$ for function approximation version of SARSA(λ)

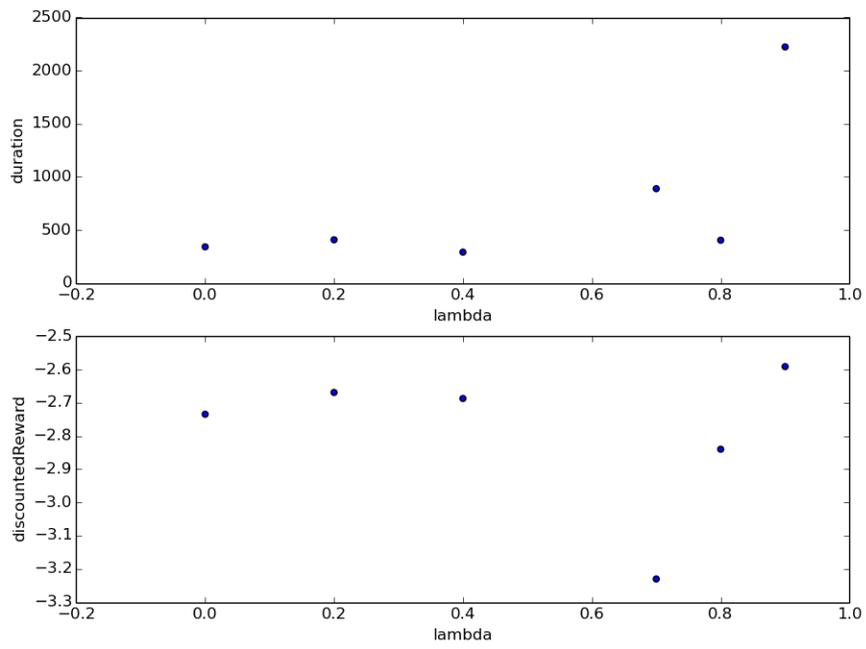


Figure 9. Comparison of performance of discrete SARSA(λ) for different λ values.

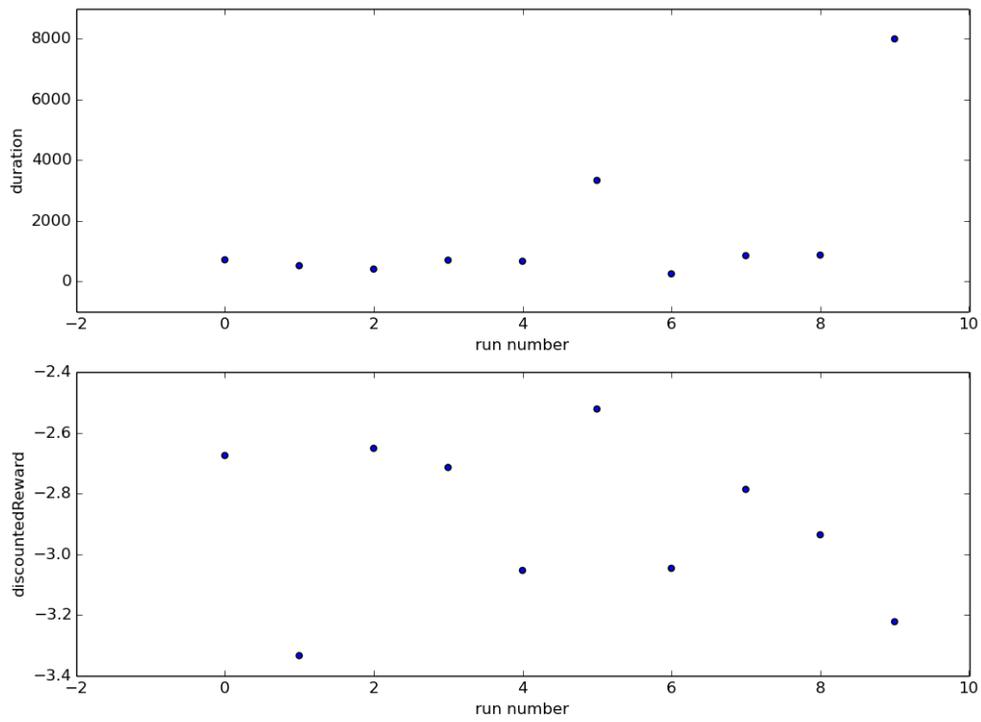


Figure 10. Comparison of performance across 10 runs of discrete SARSA using parameters in Table 2. The duration is in simulator ticks, there are 20 ticks/second.

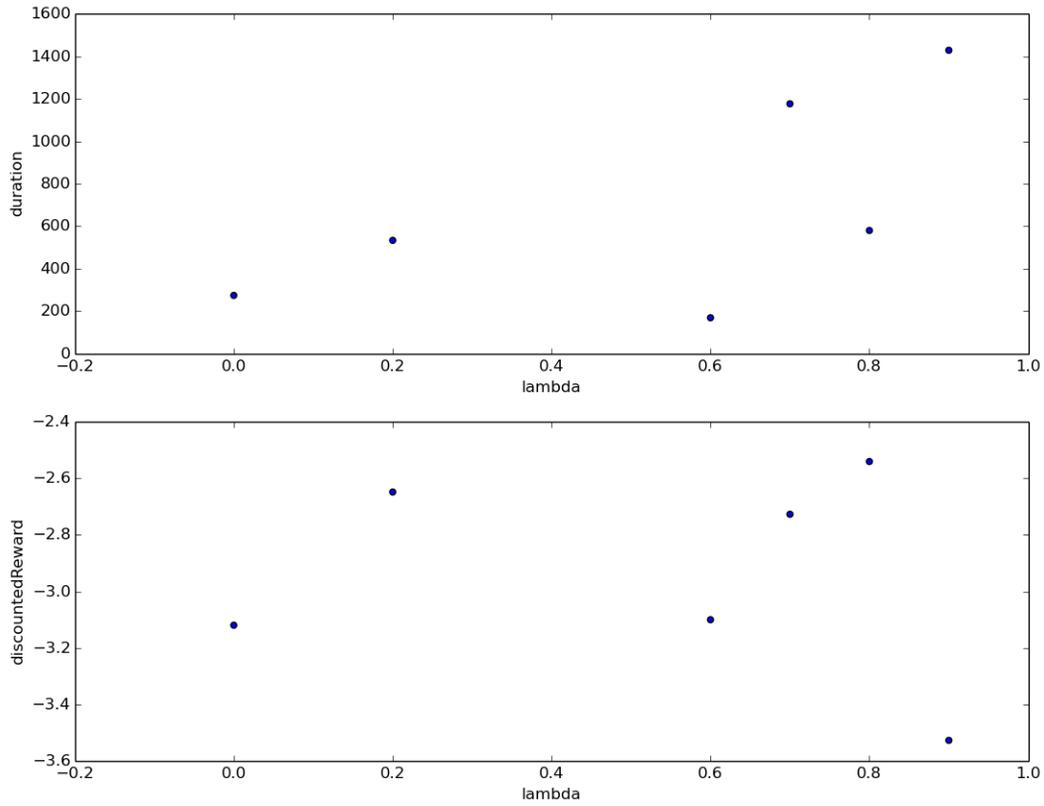


Figure 11. Comparison of performance of discrete Q-Learning(λ) for different λ values.

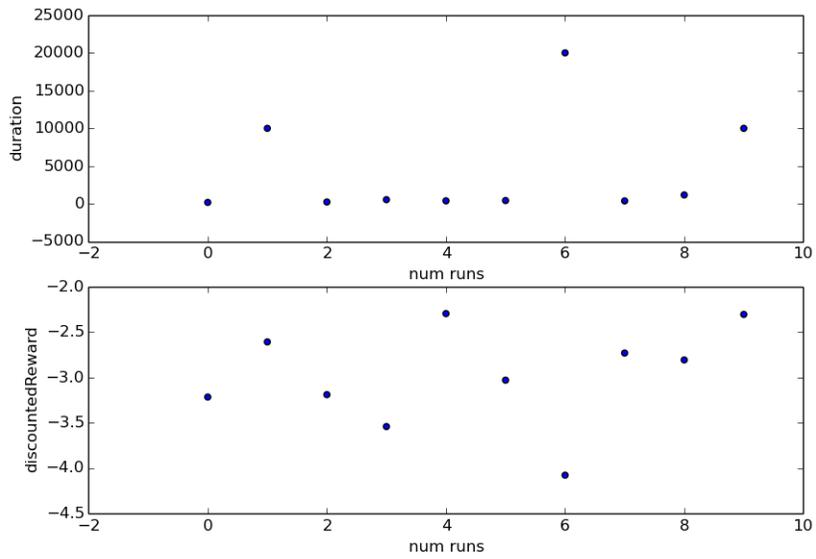


Figure 12. Comparison of performance across 10 runs of discrete Q-Learning using parameters in Table 2.

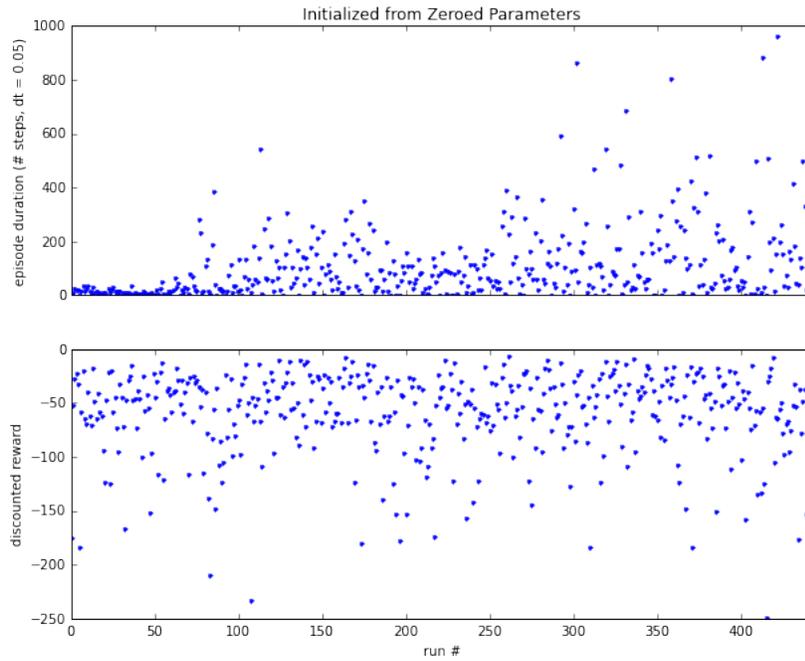


Figure 13. Performance (run duration and discounted reward) over time of the policy search controller (Episodic REINFORCE), initialized with all zero weights. Parameters for the gradient descent update were $\eta = 5e - 2$ and $\sigma_p^2 = 4.0$.

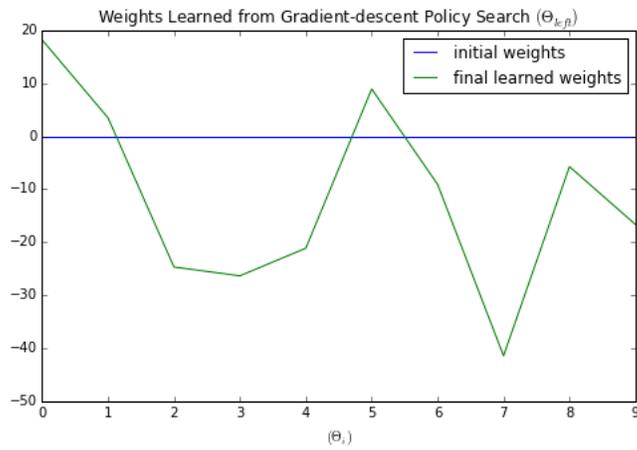


Figure 14. Initial (all zero) and final weights θ_{left} of the policy search controller (Episodic REINFORCE), for the same trained controller as in the figure above.

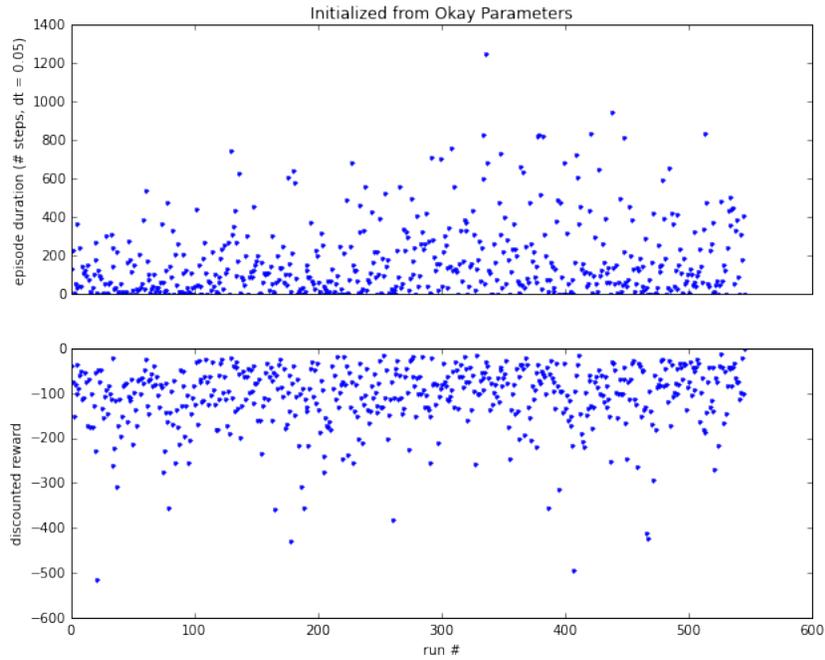


Figure 15. Performance (run duration and discounted reward) over time of the policy search controller (Episodic REINFORCE), initialized with sensible but imperfect weights (as plotted in the figure below). Parameters for the gradient descent update were $\eta = 2e - 2$ and $\sigma_p^2 = 6.0$.

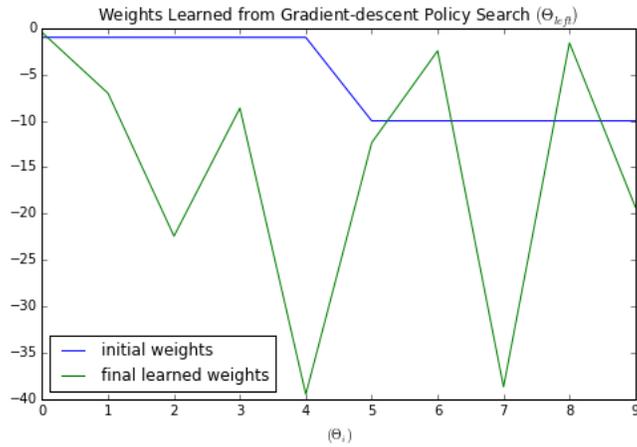


Figure 16. Initial and final weights θ_{left} of the policy search controller (Episodic REINFORCE), for the same trained controller as in the figure above.

RL for Obstacle Avoidance

Parameter	Value
N_{inner}	5
N_{outer}	4
β_{cutoff}	0.5
α	0.2
γ	0.95
λ	0.7
C_{action}	0.4
C_{max}	20
$C_{collision}$	100
C_0	0.3
$C_{raycast}$	40

Table 2. Default parameter values for SARSA(λ)