Machine Learning Engineer Nanodegree: "Tune the Mountain-Car" Capstone Project Report

Charlotte Dieter-Ridder, June 6th, 2017

History

June 6th	version 1.0	initial version
	version 1.1	

I. Definition

Project Overview

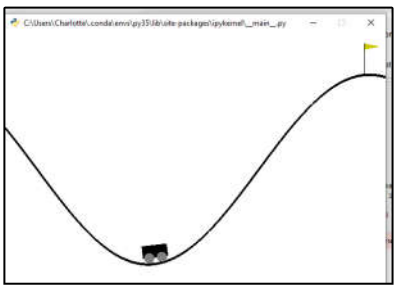
Reinforcement Learning (RL) is a fascinating area of machine learning, as the system learns "on-the-Job" from its environment. RL algorithms approximate complex optimization problems which cannot be solved otherwise because of their complexity (e.g. Backgammon has around 10^{20} states). RL got a lot of publicity when Google's DeepMind created AlphaGo won a Go game against the world greatest Go player.

To gain more insight to this area, I'll work on a classic RL problem, and I'll implement it in a standardized environment. As problem, I've chosen the "mountain car problem". As execution and evaluation environment I will use the implementation provided by [OpenAI].

[OpenAI] is an open source environment, providing standardized environments for RL problems. Their goal is to make implementations comparable, and so it is possible to upload an implementation and evaluate it in their environment.

Problem Statement

The "mountain-car" problem is a classic RL problem. In [Wiki_MountainCar] the problem and its history are described:
"*Mountain Car, a standard testing domain in reinforcement learning, is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill.*"



This problem has some interesting aspects:

- The status of the car is continuous: velocity and position. They must be converted first to a discrete number of states before known RL algorithms can be used.
- Before reaching the top of the hill the car must drive up the neighbor to get enough momentum. The agent must increase the distance to the goal before reaching it.

History of the problem according to [Wiki_MountainCar]:
The mountain car problem appeared first in Andrew Moore's PhD Thesis (1990) [Moore, 1990]. The problem

became more widely studied when Sutton and Barto added it to their book Reinforcement Learning: An Introduction (1998).

Metrics

[OpenAI] want to make implementations comparable. Therefore, I stick to their metrics:

MountainCar-v0 is considered "solved" when the agent obtains an average reward of at least -110.0 over 100 consecutive episodes. Evaluation will be done by uploading the agent to [OpenAI]. They will provide a standardized evaluation.

Project Outline

The challenge for me is the initial implementation and its evaluation, followed by improvements, and trying alternative concepts. So, the plan is:

- **Preparation:**
  Get the environment running and understand the basic approach for the solution.

- **Implement the simplest possible solution as a benchmark:**
  based on python and the typical libraries (numpy, ..) . Starting using TD(0)-Sarsa with single tiling I will have to add more complexity until the car reaches the mountain for the first time. I expect a steep learning curve when trying to understand what is happening, and to learn how to monitor and how to visualize the learning of the agent.

- **Improve the algorithm by tuning the parameters**
  until I'll arrive at the benchmark.

II. Analysis

Data Exploration: Model Exploration

As this is a RL problem, there are no data sets. The agent will learn from the environment. To base the project on a common environment I will use the implementation provided by [OpenAI]. So, exploration of the data is replaced by exploration of this model.

Interface between environment and agent

The interface of the environment is characterized by:

- The state of the car is characterized by velocity and position:
  velocity: range: [-0.07, 0.07],
  position: range: [-1.2, 0.6]
- Starting Condition:
  velocity =0.0,
  position: random between [-0.6, -0.4] (-0.5 is the deepest point in the valley)
- Termination Condition:
  position >= 0.5

The environment implements the update of the state of the car, based on the action taken by the car. Possible actions are:

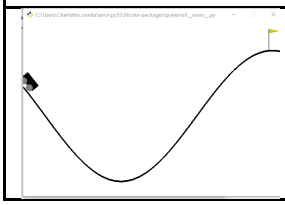
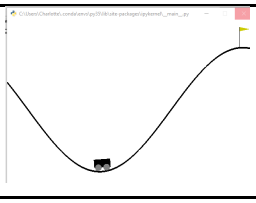
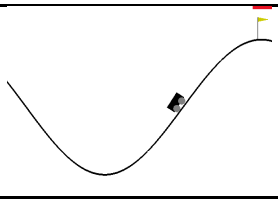
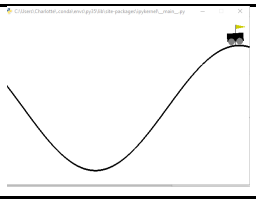
Action = [left, none, right] or [0,1,2]

Status is updated after each time step:

```
velocity += (action-1) *0.001 + math.cos(3*position)*(-0.0025)
velocity = np.clip(velocity, -self.max_speed, self.max_speed)
position += velocity
position = np.clip(position, self.min_position, self.max_position)
if (position==self.min_position and velocity<0): velocity = 0
```

Where is the position of the car?

Position of the car is in interval [-1.2, 0.6]:

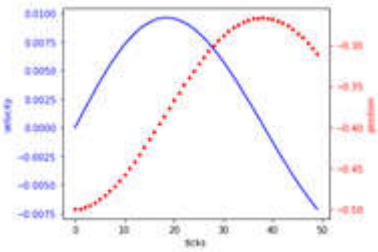
Position = -1.2	Position = -0.5	Position = 0.0	Position = 0.5
			

Start position of each episode is between -0.4 and – 0.6, so somewhere in the valley.
Goal position > 0.5, so we are up the right hill.

How powerful is the car?

What happens during the update? For better understanding of the mechanism of the environment, I run it brute force – initially starting in the valley, go first to the right, to see when velocity decreases to zero, then back to the left, finally go to the right again.

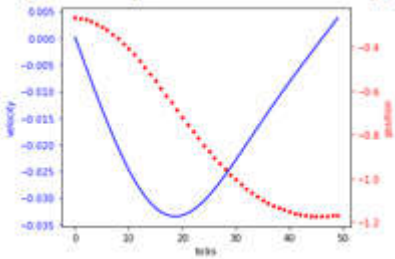
Step 1: "full power forward, starting in the valley"



Let us assume, we start in the middle of the valley and choose in any case "full power forward":
start: position: -0.5, velocity: 0.0
action: full power forward (2)
what happens?
max velocity: 0.0096 reached at position: -0.38
max position: -0.26 reached with velocity: 0.0002

With own power, we reach about a quarter of the hill. If we decide to switch to action "full power back" we get this figure:

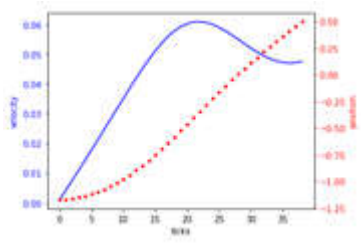
Step 2: "full power back, starting after 25% up the right hill"



start: position: -0.26, velocity: 0.0002
action: full power back (0)
what happens?
min velocity: -0.033 reached at position: -0.687
min position: -1.17 reached with velocity: -0.0001

When we reach the valley again, we have already some velocity (ca. -0.005 at position -0.5) which helps us to climb the left hill. Our maximal velocity is higher |0.033| instead of |0.001|, and until it is slowed down to zero, the car has reached the top of the left hill.

Step 3: "full power forward, starting on top of the left hill"



start: position: -1.17, velocity: 0.001
action: full power forward (2)
what happens?
max velocity: 0.061 reached at position: -0.343
max position: 0.498 reached with velocity: 0.04

and the car has reached the goal after 40 ticks.

Exploratory Visualization

See above – model exploration.

Algorithms and Techniques

Overview

There are 3 pieces of theory which will be used:

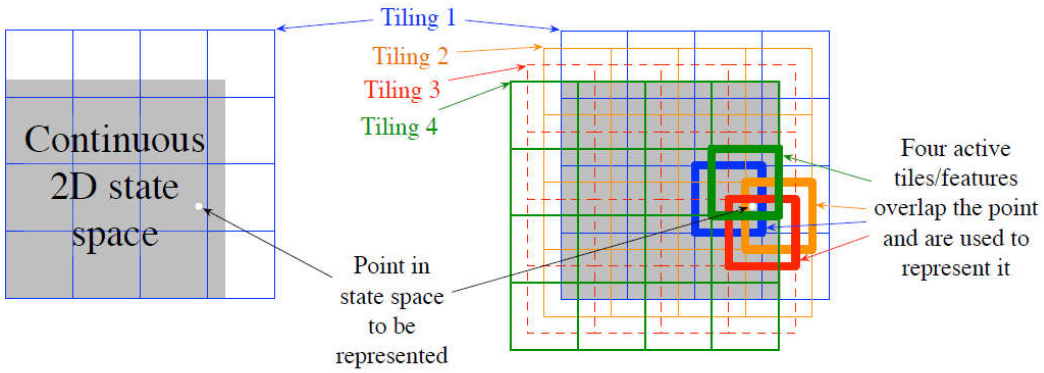
- Tile coding for discretization of the status
- TD(0) as basic implementation
- Q-Learning and TD(λ) for further improval

Tile coding

The environment delivers the state of the car as continuous variables. As all Markov-Models expect a finite state space this state must be discretized. Recommended algorithm for it is tile coding.

From [SuttonBarto, 2016]:

Tile coding:
This is a kind of coarse coding. There are some grids over a 2-dimensional layer defined, and according to the field which contains your status variable, the status is defined. So we map the input values (x,y) to discrete values (x', y'), and afterwards the Q((x',y'),a) is approximated.



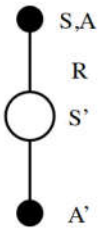
TD(0) – SARSA, Q-Learning

The "knowledge" of the car is a (State, Action)-table, which contains a value Q(S,A) for each element. As bigger Q(S,A) as more value (S,A) has for the final reward.

Our car learns after each time step. This is the "temporal difference (TD)" approach: the reward earned by each step is used immediately to improve the "knowledge" of the car. There are 2 similar algorithms which can be used: SARSA or Q-Learning:

From [Silver, 2015]:

TD(0)-Sarsa: $Q(S,A) \leftarrow Q(S,A) + \alpha (R + \gamma \mathbf{Q(S', A')} - Q(S,A))$



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma Q(S',A') - Q(S,A)\right)$$

During each timestep of an episode some decisions must be taken:

Choose next action:
Car is at state S, it chooses an action A and passes A to its environment.
The next action is chosen according to "**ε-greedy**" policy: Normally the action with best Q (S, A) is chosen, so the already earned knowledge is used, but with a certain percentage (epsilon) a random action is taken for further exploration. (0<= ε<1). So **ε** is the parameter to tuning the "exploitation/exploration-Dilemma" of each RL-problem: If ε is large, car is very curious and explores new options – if it is small the car sticks to the knowledge already earned.

Collect reward and next state:
The environment executes the next step and returns as result:
- R(eward): -1 in any case
- next state S'
- done: either because the car reached the top of the hill, or because 200 timesteps are reached.

"Learn":
Now the car has to update its "Knowledge" Q(S,A) accordingly. Parameters to be tunes are:

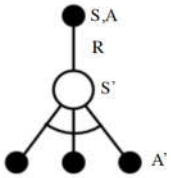
- **γ-gamma**: discount factor for next state/action - pair:
  It is straightforward to update Q (S, A) by difference Reward-Q (S, A), because this is the error between the car's knowledge (Q(S,A)) and the behavior of the environment (R). But as the goal is to optimize the final reward, not the immediate one, it may make sense to take into account future rewards as well. Q(S',A') is all we know about the future steps, and gamma (0<= gamma <1) defines its weight.

- **α-Alpha**: learning rate:
  alpha defines the pace the car learns from new steps:
  alpha=1: each new result overwrites the knowledge already earned
  alpha=0: car ignores new results

The alternative to SARSA is QLearning.

Q-Learning according [Silver, 2015]:

Q-Learning: $Q(S,A) \leftarrow Q(S,A) + \alpha (R + \gamma \mathbf{\max a' Q(S', a')} - Q(S,A))$



$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

Both algorithms choose the next action executed by "ε-greedy", but they differ when updating Q(S,A). SARSA takes the same policy as for the execution ("ε-greedy"), but Q-Learning takes argmax (Q(S',A')).

For this problem, I have chosen to use SARSA as main approach and compare with Q-Learning from time to time.

TD(lambda)

TD(0) takes into account only the recent step, even if optimizing the final reward. As online-algorithm it updates its knowledge after each time step. This is the important difference to offline-Monte Carlo-algorithm, which wait until the end of an episode to update the all states visited based on the knowledge about the whole sequence. Considering the future is not an option for an online-learner. Instead it is an option to consider the past, the steps the car has already passed. As essential idea of TD(λ), the steps already passed are stored as eligibility trace and considered during update of the "knowledge" Q. During update, not only Q(S,A) is updated, but the whole table, according to the factors in the eligibility trace: recent state has factor 1, state before has λ^1 , 2 states before has λ^2 , and so on. So the knowledge about the rewards are faster propagates along the steps the car has used up to now.

As SARSA-TD(λ) it is described by [Silver,2015]:

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S',A') - Q(S,A)$
    $E(S,A) \leftarrow E(S,A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

There is an additional table E() to store the eligibility traces, with the same dimensions as Q(). (number of states *number of actions)). E is initialized to "0" at the beginning of each period. During each step:

E(S,A)+=1, if car is at S,A.

Update not only Q(S,A), but the whole table Q by Q()+=α (same delta as TD(0)) * E().

E()*=λ*γ*E().

What happens for lambda=0 (and gamma=1)
Exactly the same as for TD(0): only E(S,A)==1, Q(S,A) is updated as before, and E(S,A) is reset to zero afterwards.

What happens for lambda=0.5?
After 1st step: Q(S1,A1): as TD(0), E(S1,A1)=0.5
in 2nd step:
Q(S2,A2): as TD(0),
Q(S1,A2): updated again, but with a factor 0.5
E(S2,A2)=0.5, E(S1,A1)=0.25

Goal is to learn faster. It will be tried during refinement.

Benchmark

OpenAI gym [OpenAI] has defined when the problem is seen as solved:
"MountainCar-v0 is considered "solved" when the agent obtains an average reward of at least -110.0 over 100 consecutive episodes." This is the benchmark which will be used when uploading the implementation to them, and so this is the "final goal". What does it mean? It means that the system learns to climb the hill in around 110 steps, so that a sequence of 100 episodes can be measured, where the mean number of time steps to reach the top of the hill is less than 110.

III. Methodology

(approx. 3-5 pages)

Data Preprocessing

Not applicable

Implementation

Overview

Implementation is done in Python 3.5 to be compatible with tensor flow (to have it as option, even if finally not used), using Jupyter notebooks, to be able to add texts and to store code and results together. Parts of the implementation are:

- Tile coding and eligibility traces
- Evaluation
- Strategies to decay hyperparameters depending on the episode
- Implementation of the reinforcement learning itself

Tile coding and Eligibility Traces

Class "tileModel" is a straightforward implementation of tile coding. Central function is "code". It takes the input status, shifts it to a range starting with zero and scales to the grid size. If the grid is 8*8, both velocity and position are shifted and scaled to the interval [0,8). Let's name them coordinatesScaled[]. Each tiling level needs now a pair of integers as coordinates. The grids of the tiling levels are shifted against each other by an equal distance, 1/numberOfTilings. If there are 8 tiling levels, the distance between two neighbored tiling levels is 0.125.

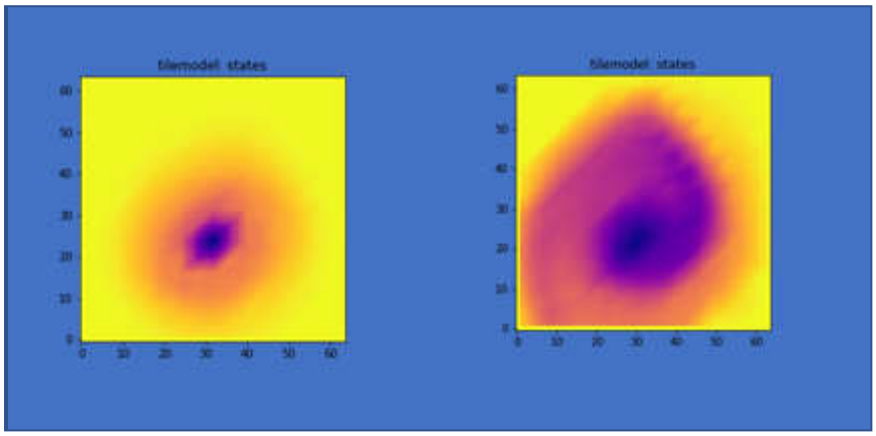
So, the coordinates for each tiling level are computed by:
coordinates[tiling][0:1]=np.floor(coordinatesScaled[0:1]+tiling*1/numberOfTilings)

As result we have numberOfTilings coordinate pairs, which are now mapped to a 1-dimensional array. As result there is a vector, length=nbOfTilings, of integers, each is the index for Q(S,A) for the tiling levels.

Besides the coding the status class tileModel contains the table Q(S,A), as well as the functions getQ() and updateQ(), which are used by the central algorithm.

To visualize the Q(S,A) there is an inverse function. It converts 8 tiling levels back to one 2-dimensional array, and plots the sum (Q) as heat map.

Find two examples here:



The left one is a Q(S,A) table where the car was trapped in the valley, the right one was very successful.

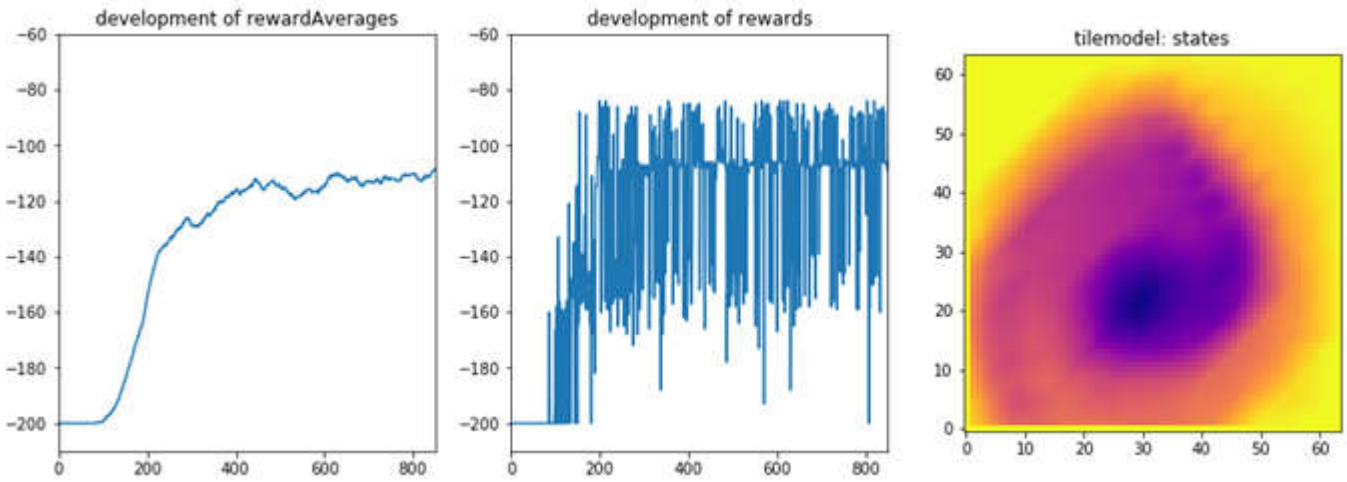
Eligibility Traces
This data structure is used for TD(lambda) and similar to TileModel. It contains the data structure, as well as the lookup and update functions, needed by the algorithm.

Evaluation

Class evaluation collects the evaluations of the experiments. One episode of the mountain car model has max. 200 steps. Lots of episodes are executed one after the other to reach the benchmark. I named a sequence of episodes a "cycle". So, one cycle is defined by n episodes, which were executed and the result afterwards documented. Interesting numbers for one cycle are:

```
number episodes: 850
first time arrived in episode: 87
car arrived: 733 (86.2%)
best reward: -84.0 in episode 200
best average: -108.6 in episode 849
problem solved:True
```

and as plots:

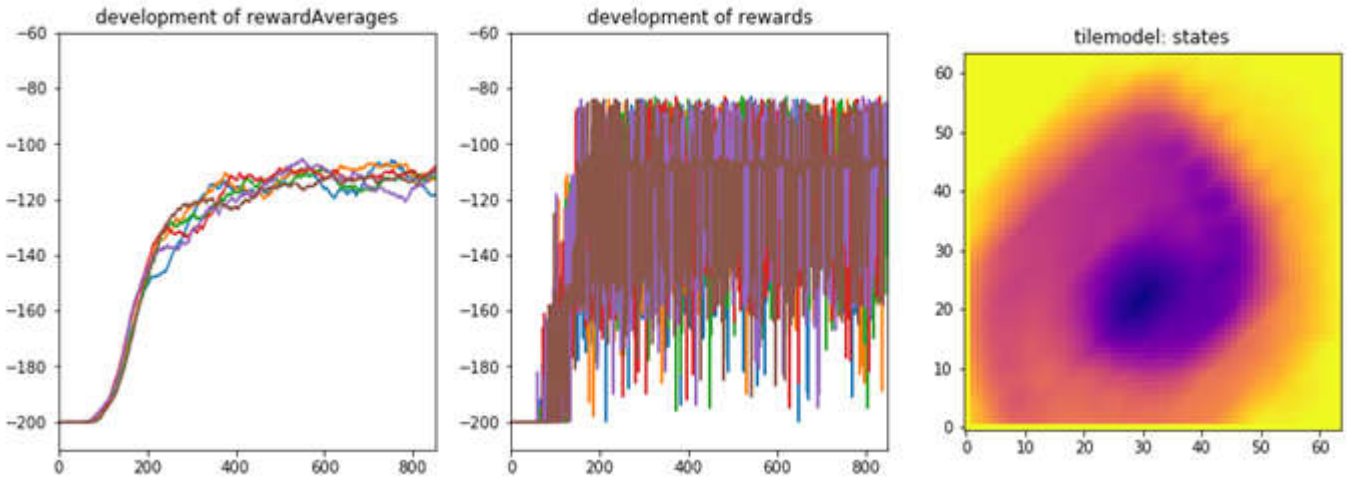


Both the development of the accumulated rewards and of the rewardAverage are interesting. rewardAverage is interesting because this is the benchmark. The development of the accumulated reward explains the development of the average and helps to decide which parameter should be tuned.

In some parametrizations, the results between cycles differ remarkably. As it is misleading to judge about a parametrization only based on one cycle, I defined an additional level "run" which consists out of several cycles, which are evaluated together at the end of the run:

```
test finished after 6 runs
input: policy: <function policy at 0x000002421D01ED90>, lambda: 0.6
first arrival: min: 63, max: 71, mean: 66, std: 3.0
nb of arrival: min: 751, max: 762, mean: 756, std: 3.6
nb of arrival: min: 88%, max: 90%, mean: 89%, std: 0.4%
best reward  : min: -106, max: -83, mean: -88, std: 8.2
in episode   : min: 184, max: 722, mean: 415, std: 164.3
best average : min: -122, max: -105, mean: -112, std: 5.4
in episode   : min: 248, max: 754, mean: 567, std: 181.2
prblem solved: min: 0, max: 1, mean: 0, std: 0.5
```

Mean and standard deviation tell us much more about the quality and of the stability of a result. According plots are:

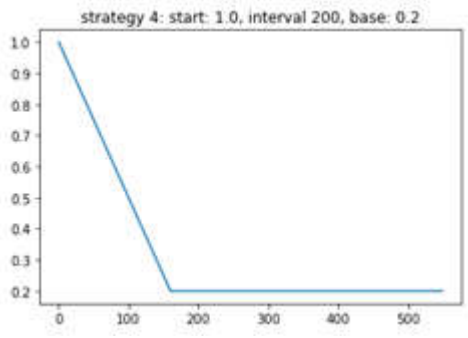


Where we see the development of rewards, and rewardAverages in all cycles together. Tilemodel is from the last cycle.

Strategies to decay hyperparameters depending on the episode

During refinement, the model is tuned by figuring out the best hyperparameters. One option are static values, other is to decay a parameter depending on the episode. In decay function, there are several strategies collected. In the final model only strategy -1 (= constant value, episode independent) and strategy=4 are relevant.

Strategy 4 decays the parameter linear until a base value is reached:



Example is curiosity parameter ε: at the beginning we want the car to try new options with high probability, later we prefer that is stays to the model already learned.

Implementation of the reinforcement learning itself

Last part of the implementation is the implementation of the algorithm itself. It follows the pseudocode, so only one remark: the state S is a vector of nbTiling indices, so Q(S,A) is more exactly Q(S[0:nbTilings],A). As we need one value for Q (S,A), getQ delivers np.sum (Q(S[0:nbTilings],A), but updates each index with the full delta.

Example:
delta for update of $Q(S_x[0:8],A_x)$ is 0.5: each index is updated.
next getQ ($Q(S_x[0:8],A_x)$) delivers 4 instead of 0.5.
to avoid this effect, alpha is scaled by number of tilings.

Refinement

In the paragraph the steps are presented which brought the car up the hill for the first time.

Basic understanding of the hyper parameters without tiling

As **base parameters**, I used:
Learning rate: alpha = 0.5
exploration rate for ε-greedy policy: 0.1
discount parameter for Q(S', A'): 1.0
Initialization of Q(S,A): np.random.uniform(low=0.0, high=0.0001): so near zero, but without biasing the action '0'.

On this base I varied:

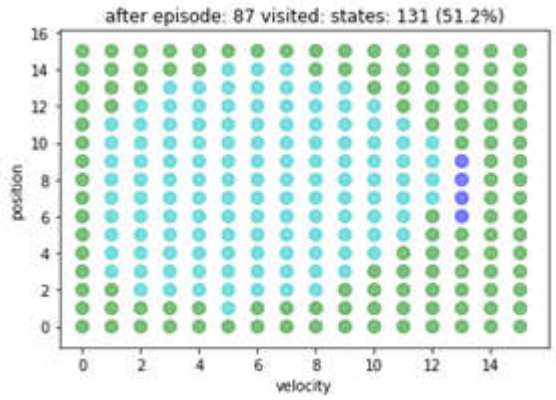
Grid granularity

Coverage of the state space is a good indicator about the exploration. After 10 episodes how many states of the state space are visited?

Granularity	10 episodes	50 episodes	100 episodes
4*4	12.5%		
8*8	6.2%		
16*16	24.2%	47.4%	51,2%
32*32	13%	27.8%	37,8%

(SARSA, TD(0), α=0.5, ε=0.1, γ=1.0)

If the field in table is empty, the exploration has stopped beforehand. This plot shows the episode-by-episode progress:



Green dots are not yet explored, dark blue have been explored for the first time in last episode. Exploration of the borders is very slow, and without reaching the extreme positions the car will not arrive at top of the hills.

Even if w/o tiling the 16*16 grid performs best, it has reached a coverage of 51,2% of the state space after episode 87, and afterwards there is nothing new. In the 32*32 grid, we just have explored 38% of the space, but it is still exploring: In any case: both results are not very helpful: 16*16 stops exploring at around 50%, 90 Episodes and 32*32 explores to slow. For further tests, we will use 16*16.

Gamma:
Gamma defines how much Q(S',A') are taken into account. We started with 1.0:
Learning rate: alpha = 0.5
exploration rate for ε-greedy policy: 0.1
discount parameter for Q(S', A'): 1.0
Initialization of Q(S,A): np.random.uniform(low=0.0, high=0.0001): so near zero, but without biasing the action '0'.
Grid: 16*16

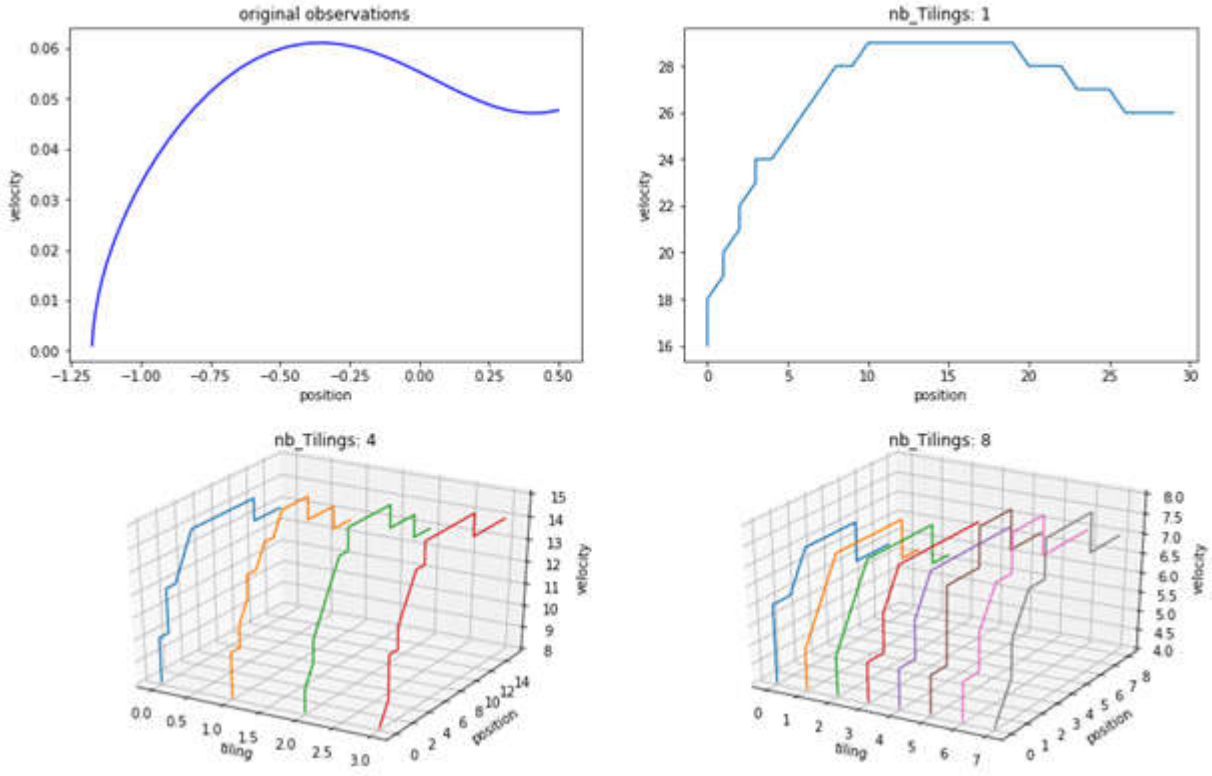
gamma	Coverage after 50 Episodes	After 100 Episodes	Exploration stopped at
0.1	11.3%		Episode 17
0.9	44,5%	49,2%	ca Epi 80
1.0		51,2%	ca Episode 87

(SARSA, TD(0), α=0.5, ε=0.1, grid 16*16)

With variations of alpha and epsilon no better coverage was reached. Conclusion is: grid 16*16 is the best with tiling =1, but will not reach the top of the hill.

Tiling – grid size and tiling levels

Why do we make the additional effort of tile coding? For better and intuitive understanding, find a plot, where the same sequence of states is discretized with different numbers of tilings, in an episode, where the car moves fast:



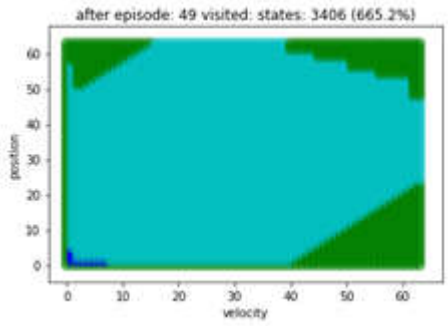
At top right, there is the original observation, with continuous position and velocity. At top left, there is a discretization to a 32*32 grid (=1024 points). 39 observation steps were mapped 19 states, so we can imagine that we have lost some information, even in a scenario were the car moves fast.

At bottom left, there is a 16*16 grid with 4 tilings (=1024). Each level has 11 points, but there are 4 levels =>44 points. At bottom right, we have a 8*8 grid with 8 tilings (=512 points), 9 points/Level, 54 overall.

Compare different tilings

Tiling:	50 Episodes
1*16*16	ca 50% still exploring
2*16*16	447 states/1024: 23%
4*16*16	2043 states/4096: ca. 50%
8*8*8	3046 states/4096, ca 75% hill **reached 4 times**

(SARSA, TD(0), α=0.5, ε=0.1, γ=1.0)



The car reached the hill for the first time with 8*8*8, and explored the state space better than before: Other tiling configurations do not work as well as 8*8*8.

IV. Results:

Model Evaluation and Validation

After having implemented and administrated an agent which can reach the hill at all, now the hyper parameters (epsilon, alpha, gamma, lambda) must be tuned to reached benchmark. It is necessary to have a reliable solution – reliable in the sense that results can be reproduced. As important as reliability is stability of the accumulated rewards during the episodes. It must be ensured that the model does not forget its learnings.

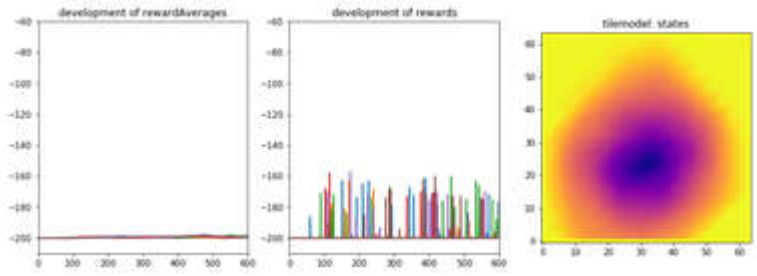
As consequence, this means that you may never trust one cycle alone – all the documented experiments are bases on 4-6 cycles/parametrization, and a cycle must contain enough episodes. The number of episodes/cycle differ between 500 and 750. Comparable short experiments (4*500) run during the day, longer experiments were started during the night.

With parallel experiments changing only one parameter I tried to figure out the effect of one parameter, and tried to do the experiments with clear results first:

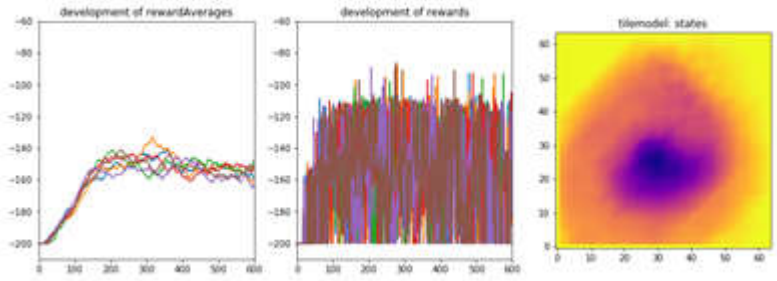
Epsilon – Decay or keep constant?

Epsilon	Best Average	Accu Reward	%arrival
0.5	min: -199, max: -198, mean: -198, std: 0.4	min: -163, max: -157, mean: -160, std: 2.0	min: 2%, max: 3%, mean: 3%, std: 0.6%
Max (0.5-episode/200, 0.1)	min: -146, max: -133, mean: -141, std: 3.8	min: -93, max: -86, mean: -90, std: 2.6	min: 83%, max: 88%, mean: 85%, std: 1.6%
0.1	min: -156, max: -149, mean: -152, std: 2.7	min: -108, max: -91, mean: -97, std: 6.0	min: 77%, max: 82%, mean: 79%, std: 1.7%

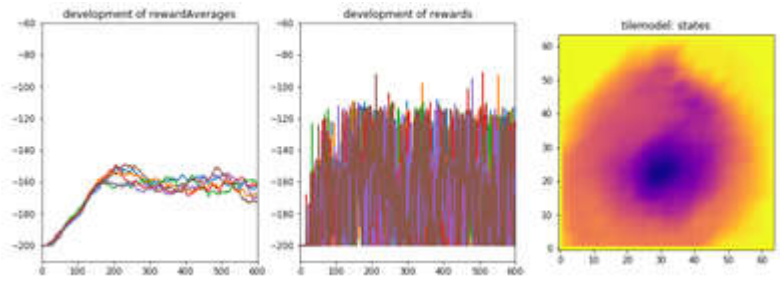
Epsilon = 0.5



Epsilon=max (0.5-episode/200, 0.1)



Epsilon = 0.1



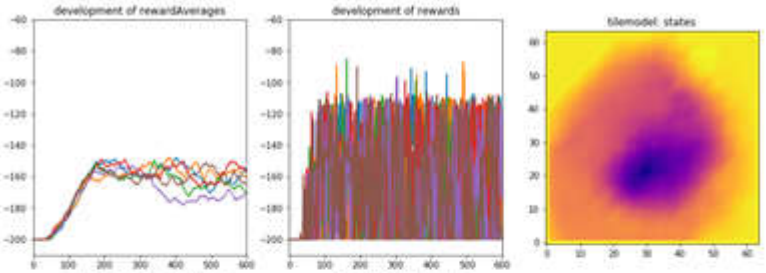
Conclusion:
Decay Epsilon! Staying at high curiosity does not lead to any useful result. Exploring little already at the beginning leads to worse results than decaying.

Alpha - Stable results over different cycles:

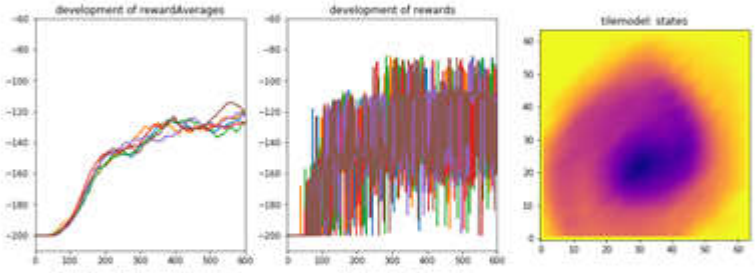
Learning rate alpha is crucial to arrive stability over several cycles. Example:

	α	Averaged Rewards	Accumulated Reward	Percentage arrival
fast	0.5	min: -153, max: -148, mean: -150, std: 1.9	min: -99, max: -85, mean: -91, std: 4.9	min: 70%, max: 76%, mean: 74%, std: 2.3%
slow	0.25	min: -125, max: -114, mean: -120, std: 3.6	min: -86, max: -84, mean: -85, std: 0.7	min: 86%, max: 88%, mean: 87%, std: 0.6%
extraSlow	0.125	min: -122, max: -118, mean: -120, std: 1.4	min: -85, max: -84, mean: -84, std: 0.5	min: 85%, max: 86%, mean: 85%, std: 0.3%

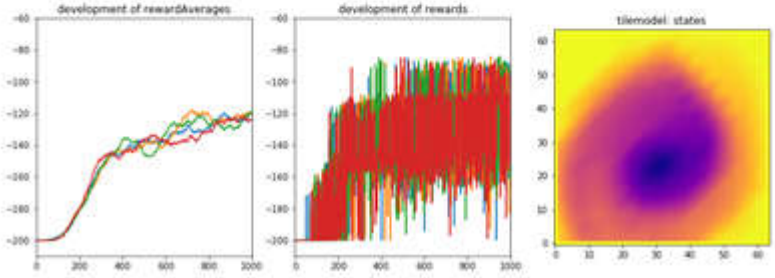
Fast learner: α=0.5:



Slow learner: α=0.25:



Extra Slow: α=0.125:



Both table and plot show that a slow learning rate increases reliability. It is visible both at the standard deviation, as well as in the reward plot: even if α=0.5 delivers from time to time excellent result, it is not reliable: In the reward plot we see that there stays a high percentage of extremely bad episodes, up until to the end.

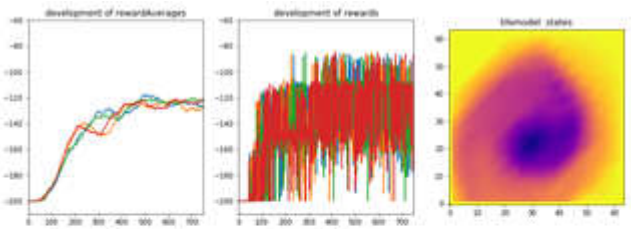
⇨ Alpha = 0.25 to keep the learning pace up to now.

Gamma – Discount factor for Q(S' ,A')

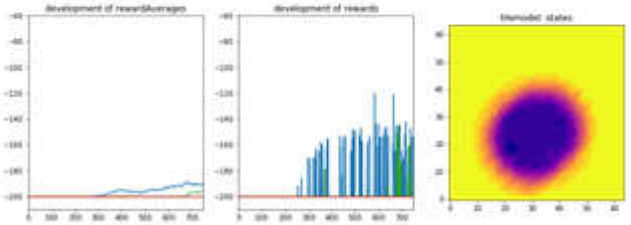
(lambda=0.2, alpha=0.25, epsilon=max(0.5-episode/200,01):

Gamma	Averaged Rewards	Accumulated Reward	Percentage arrival
1.0	mean: -120, std: 1.5	max: -84, mean: -85, std: 0.7	min: 89%, max: 90%, mean: 89%, std: 0.5%
0.75	mean: -196, std: 4.6	max: -120, mean: -166, std: 34.7	min: 0%, max: 9%, mean: 3%, std: 3.8%
0.5	mean: -200, std: 0.0	max: -200, mean: -200, std: 0.0	min: 0%, max: 0%, mean: 0%, std: 0.0%

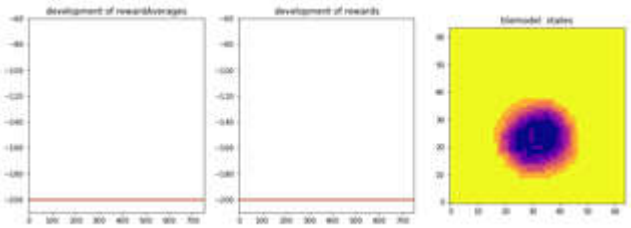
Gamma=1.0:



Gamma=0.75:



Gamma=0.5:



Conclusion:

With gamma < 1.0 the car does not even recognize that there is a top of the hill. It never leaves the value. For more tries: gamma = 1.0

Epsilon: Start/Base/slope of decay

Start value:

Epsilon start	Best average	Best reward
0.25	mean: -119, std: 2.5	max: -84, mean: -84, std: 0.4
0.5	mean: -120, std: 1.4	max: -84, mean: -84, std: 0.5
0.75	mean: -120, std: 1.2	max: -85, mean: -86, std: 0.7

(alpha=0.25, lambda=0.2, gamma=1.0 – epsilon: interval=200, base=0.05)

The difference is not big, but start value was changed to 0.75 because of best standard deviation.

Slope of decay

(alpha=0.25, lambda=0.2, gamma=1.0 – epsilon: start=0.75, base=0.05)

Interval	Best average	Best reward
200	max: -83, mean: -83, std: 0.4	mean: -112, std: 2.1
400	max: -84, mean: -85, std: 1.3	mean: -120, std: 2.4
600	max: -84, mean: -85, std: 1.2	mean: -121, std: 1.5

Based on the reward curve – which has less noise in the lower area in the reward figure I stayed with interval =200.

Base value:

(alpha=0.25, lambda=0.2, gamma=1.0 – epsilon: start=0.75, interval=200)

Base value	Best average	Best reward	Solved?
0.05	mean: -120, std: 1.2	max: -85, mean: -86, std: 0.7	0/4
0.01	mean: -112, std: 2.1	max: -83, mean: -83, std: 0.4	1/4 solved
0.005	mean: -110, std: 1.0	max: -83, mean: -84, std: 0.4	3/4 solved
0.001	mean: -110, std: 1.7	max: -83, mean: -84, std: 0.5	3/4 solved
0.00	mean: -111, std: 2.1	max: -83, mean: -83, std: 0.4	2/0 solved

Conclusion:
This is the first configuration where the problem is solved. Because of best standard deviation 0.005 is chosen for further optimization.

Lambda – propagation factor to the eligibility trace

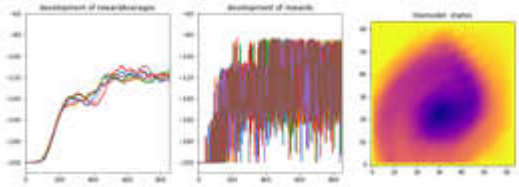
Some runs with different lambdas at the beginning of the process did lead to a clear picture, so there are not documented further up. All test until here have been done with Lambda=0.2.

With the first successful parametrization, there was a new test suite varying lambda:

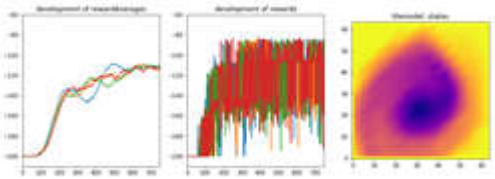
(alpha=0.25, gamma=1.0 – epsilon: start=0.75, interval=200, base=0.005)

Lambda	Best average	Best reward	Solved?
0.0	max: -109, mean: -113, std: 2.4	max: -83, mean: -83, std: 0.0	2/6 solved
0.2	max: -108, mean: -110, std: 1.0	max: -83, mean: -84, std: 0.4	3/4 solved
0.4	max: -105, mean: -107, std: 1.6	max: -83, mean: -83, std: 0.5	6/6 solved
0.6	max: -105, mean: -112, std: 5.4	max: -83, mean: -88, std: 8.2	3/6 solved

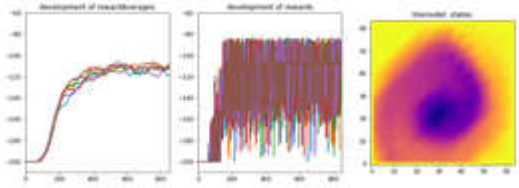
Lambda=0.0:



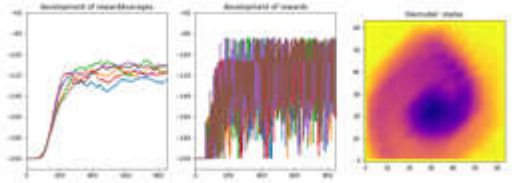
Lambda=0.2



Lambda=0.4:



Lambda=0.6:



The figures explain the good results of Lambda=0.4: The development of the rewards show for all 4 configurations a stable upper line of accumulated rewards. The interesting difference is the lower line. Lambda=0.4 has the least number of bad runs, which influence negatively the rewardAverage. Additionally, the configuration with lambda 0.4 arrives early at stable upper line.

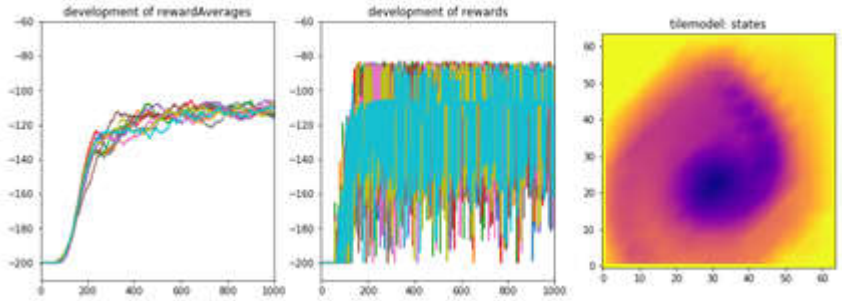
Sarsa <> QLearning

In between the refinement, I tried in parallel with Q-Learning but without clear results. That is why I run the "winner configuration" as well with Q-Learning:

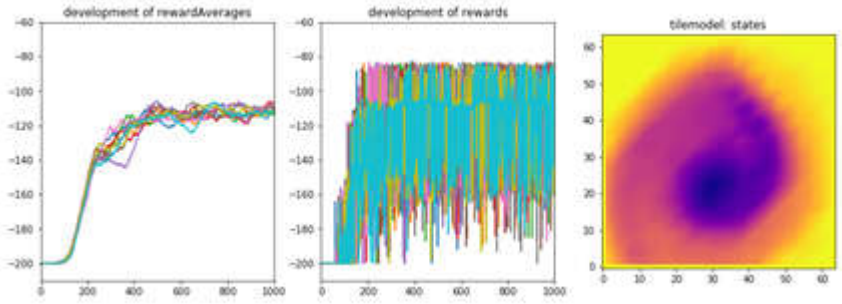
(alpha=0.25, gamma=1.0 – epsilon: start=0.75, interval=200, base=0.005)

Lambda	Best average	Best reward	Solved?
SARSA/0.4	min: -109, max: -105, mean: -107, std: 1.1	min: -84, max: -83, mean: -83, std: 0.4	10/10 solved
Q-Learning/0.0	max: -109, mean: -110, std: 1.0	max: -83, mean: -83, std: 0.4	3/10 solved
Q-Learning /0.4	min: -108, max: -106, mean: -107, std: 0.6	min: -84, max: -83, mean: -83, std: 0.5	10/10 solved

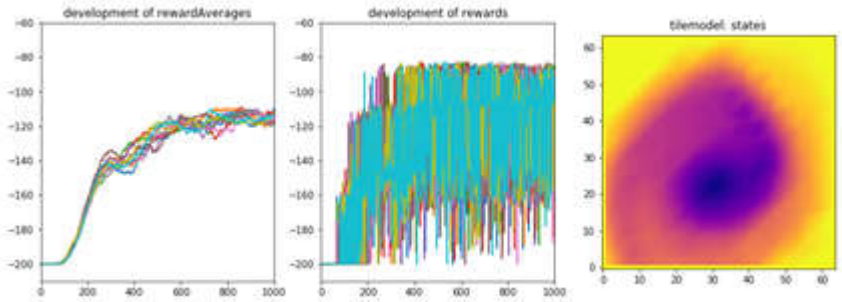
SARSA/Lambda=0.4:



Q-Learning/Lambda=0.4:



Q-Learning/Lambda=0.0:



After 10 cycles with 1000 episodes each I do not see any difference between Q-L and SARSA for λ=0.4. Both deliver a very good result very stable. Q-L/λ=0.0 is clearly worse than both of TD(γ=0.4), so the additionally effort for the eligibility traces can be accepted, because it adds the final bit of performance and stability.

Justification

Finally, I achieved a configuration matching the benchmark, and gained on my way some insight about the effect of the parametrization. For alpha, gamma, even for epsilon there are clear indications.

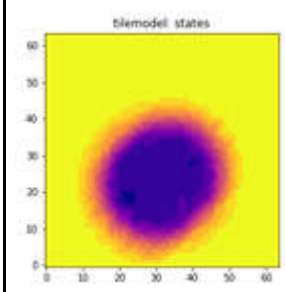
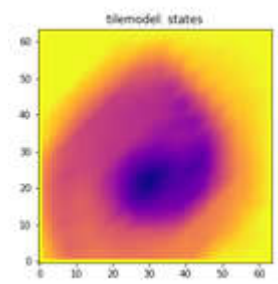
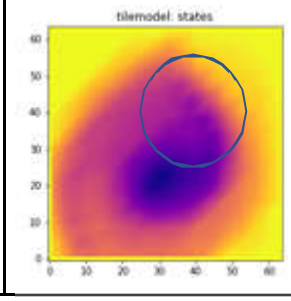
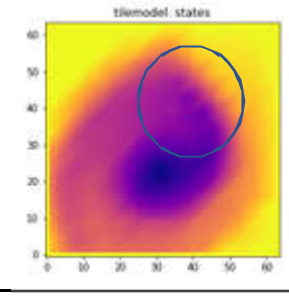
More difficult is the decision between SARSA and Q-Learning. There I haven't found experiments showing a clear preference.

The impact of Lambda/Eligibilty trace is trickier. According to the literature I expected faster learning, If the development of the rewards over the episodes are compared for Q-L(λ=0) and Q-L(λ=0.4) between episode 200 and episode 400 the runs with λ=0.4 achieve better rewards. For arrive at the benchmark it is more important that the figures λ=0.4 show less bad rewards in the later phase.

V. Conclusion

Free-Form Visualization

The heat maps used during the evaluation give a good indication about the achieved knowledge:

Car trapped in the valley: γ=0.75 <> γ=1.0 characterized by a blue ball with clear limits.		
Epsilon: 0.75->0.05 --- 0.75=> 0.005 Only visible if the picture are zoomed: there is a "nose" in the knowledge, and when the contours of the nose are clear the car performs better.		

Reflection

It is quite impressive that it is possible that a machine can figure out a stable solution for an optimization problem which is not trivial, only by try-and-error and some tricky book keeping of the rewards it has got. It is still more expressive because the reward is always the same (-1) and does not contain any information about the position, what you would expect.

This is the upside of the picture – the downside is, that manually tuning the parameters is extremely time consuming and complex even for 4 parameters. I struggled completely until I decided to execute the experiments very systematic, and run each combination of values several times.

But:
Mountain Car is a very simple environment. The "only" problems for the learner are:

- discretization of the states
- detecting that it must climb first the left hill to get enough velocity to arrive at the top of the right hill.

It is completely unclear for me how to tune an agent/environment model, which models a real-life problem and is only a little more complex. I tend to see tuning of an RL-System as an RL-challenge on its own.

For real life, we have additionally the problem that we have deliver stable and reliable behavior. What we see even in the winner configurations: they arrive very stable at the hill, but all of them have episodes in between with very poor performance.

So, quality management gets complete new challenges when it comes to deliver a machine learner in a defined quality.

Improvement

Next big step would be to change the underlying technology to deep neural networks to see if the tuning gets easier and the results get more stable.

Appendix/Sources

[OpenAI]: https://gym.openai.com

[Moore, 1990]: A. Moore, Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, November 1990

[SuttonBarto, 1998]: Reinforcement Learning: An Introduction. Richard S. Sutton and Andrew G. Barto. A Bradford Book. The MIT Press Cambridge, Massachusetts London, England, 1998

[SuttonBarto, 2016]: Reinforcement Learning: An Introduction, Second edition, in progress, Draft, Richard S. Sutton and Andrew G. Barto, 2016

[wiki_MountainCar]: https://en.wikipedia.org/wiki/Mountain_Car

[wiki_OpenAi]: https://en.wikipedia.org/wiki/OpenAI

[Gym_Launch]: https://blog.openai.com/openai-gym-beta/

[Gym_White]: https://arxiv.org/abs/1606.01540

[Silver,2015]: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html - UCL Course on RL, Advanced Topics 2015 (COMPM050/COMPGI13), Reinforcement Learning, slides and videos

[SuttonTile]: http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tiles.html

[SuttonTiles]: http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tiles.html#Python_Versions_

[SherstovStone]: http://web.cs.ucla.edu/~sherstov/pdf/sara05-tiling.pdf