Machine Learning Engineer Nanodegree
Capstone Proposal

Charlotte Dieter-Ridder, May 10th, 2017

History

May 5th	version 1.0	initial version
May 10th	version 1.1	Intermediate version to synchronize with review again: lot of improvement done according to review, mainly in the solution chapter.

Implementing a learning agent for the "Mountain Car problem"

Domain Background

Reinforcement Learning (RL) is a fascinating area of machine learning:

Wikipedia (https://en.wikipedia.org/wiki/Reinforcement_learning): "*Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward*."

A problem is modelled as interaction between environment and agent. The agent is the learner. It gets feedback from the environment. Based on its status and on the feedback the agent chooses it next action. The feedback a reward, and the agent tries to maximize the rewards earned over the time.

Other than supervised learning, RL need no large data sets. Instead the agent uses the feedback from the environment to learn how to act in an optimal manner.

RL got a lot of publicity when Google's DeepMind created AlphaGo won a Go game against the world greatest Go player. For real live there are other areas more important. In [Sutton,Barto] as areas are mentioned:
"*Beside impressive results in games (Go, checkers, backgammon, atari) there are physical control tasks for robots, dynamic random access memory (DRAM) for computers, personalized web services such as the delivery of news articles or advertisements, modelling the modeled the thermal soaring problem (birds, gliders)*."

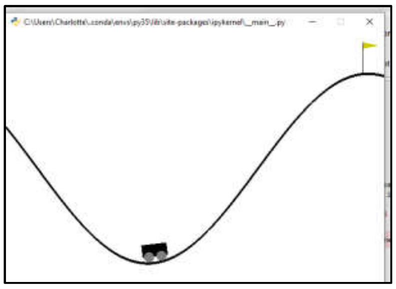
RL algorithms are able to approximate complex optimization problems which cannot be solved otherwise because of their complexity (e.g. Backgammon has around 10^{20} states).

For me this was the most fascinating part of the course but my feeling is that I'm still at the very beginning in this area. To gain some more experience I'll work on a "classic problem" in this area. I hope to get a better understanding of possible solutions and about RL in general.

Problem Statement

One of the classic RL optimization problems is the "mountain car" 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.*"


C:\Users\Charlotte\.conda\envs\py35\lib\site-packages\ipykernel\__main__.py

This problem has some very interesting aspects:

- The status of the car is continuous: velocity and position. They must be converted first to a discrete number of states before the known RL algorithms can be used.
- Before reaching the top of the hill the car has to drive up the neighbor to get enough momentum. The agent has increase the distance to the goal before reaching it.
- Only if the top of the hill is reached, the reward will be “1”. Otherwise it will be always “-1”

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) [SuttonBarto, 1998].

Datasets and Inputs

As this is a RL problem, no data sets will be necessary. The agent will learn from the environment. To base the project on a common environment I will use the implementation provided by [OpenAI].

Some words to **OpenAI** (find their mission statement here: https://openai.com/about/):
It is a non-profit research company, founded in Dec, 2015, working to “*build safe artificial general intelligence, and ensure AGI's benefits are as widely and evenly distributed as possible*”. They are “*building platforms for developing and measuring agents which take action in simulated worlds*”.

One of these platforms is **OpenAi Gym**. Intention according to the OpenAi Gym White Paper [Gym_White]:
“*OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms.*”
….

“OpenAI Gym focuses on the episodic setting of reinforcement learning, where the agent’s experience is broken down into a series of episodes. In each episode, the agent’s initial state is randomly sampled from a distribution, and the interaction proceeds until the environment reaches a terminal state. The goal in episodic reinforcement learning is to maximize the expectation of total reward per episode, and to achieve a high level of performance in as few episodes as possible.”
(In [wiki_OpenAi] you will find a good overview to the history of OpenAI.)

Technically this means that they provide an implementation of the environment via github: https://github.com/openai/gym. The environments have a standardized interface:

`env = gym.make('MountainCar-v0'):` initialization of the environment
`observation = env.reset():` reset of the environment
`observation, reward, done, info = env.step(action):` initiate next step

Both observation and action space differs between the environment. They are provided by: env.action_space and env.observation_space. For Mountain Car this is:

```
action space: Discrete(3)
observation space: Box(2,)
```

- States:
  Observation is the state of the agent in its environment. It consists out of `position` and `velocity: velocity = (-0.07, 0.07), position = (-1.2, 0.6)`
- Starting Condition:
  ```
  velocity =0.0,
  position: random between (-0.4, -0.6)
  ```

- Termination Condition:
  ```
  position >= 0.5
  ```

- Actions:
  ```
  Possible values: [0,1,2] - back, neutral, forward
  ```

- Update Function:
  ```
  velocity = velocity + (action-1)* 0.001 + cos(3*position)*(-0.0025)
  position = position + velocity
  ```

- Reward:
  in any case -1. The only indication that the work is successfully done, is "done". As the algorithm has no indication if it is on a good way, we need an algorithm which strong exploration skills.
  To enhance the complexity of the problem there's a limit of 200 timesteps after which it resets to the beginning. Successful agents must solve it in less than 200 timesteps. For this project it may be useful to change the limits, to be able to use not so successful algorithm first.

Solution Statement

Which algorithm should be used to implement the agent?

Which Algorithm?

Let's discuss this systematically (based on [SuttonBarto, 2016] and [Silver,2015]):

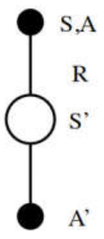
- **model-free control**
  For sure we need an algorithm out of the class "model-free control": no model available, and the agent should learn the optimal policy on its own out of the reaction of the environment.
- **Monte Carlo (MC) or Temporal difference (TD)?**
  Other than TD, MC needs completed episodes to approximate the action-value-function Q(s,a). I assume that we will have a lot of episodes at the beginning which will never end. This is a good reason to use an algorithm based on temporal difference. Other than MC, it will update the action-value-function Q(s,a) after each step, not only at the end of an episode.
- **On-Policy or Off-Policy?**
  On-Policy means "learning-on-the-job", Off-Policy means "learning-by-look-over-someone's shoulder". Q-Learning is the most prominent off-policy algorithm. In our case it must be compared with the TD(0) SARSA, which is "on-policy" and the approach proposed by [SuttonBarto, 2016]

- **TD(0) or TD(λ)?**
  TD(0) (as well as Q-Learning) updates only the actual Q(s,a). In an environment where we have a constant reward of "-1".

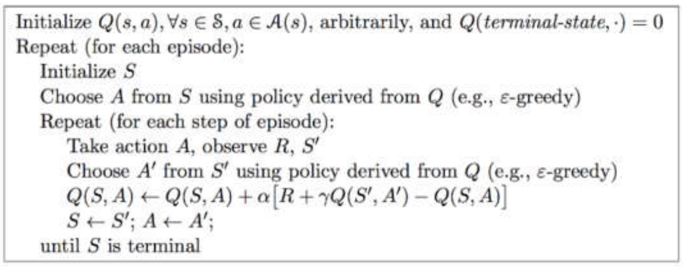
Comparision TD(0)-Sarsa – Q-Learning

(Both pseudo codes from [Silver,2015]:

TD(0)-Sarsa: Q(S,A) <- Q(S,A) + α (R + γ **Q(S', A')** – Q(S,A))

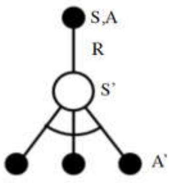


$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma Q(S',A') - Q(S,A)\right)$$

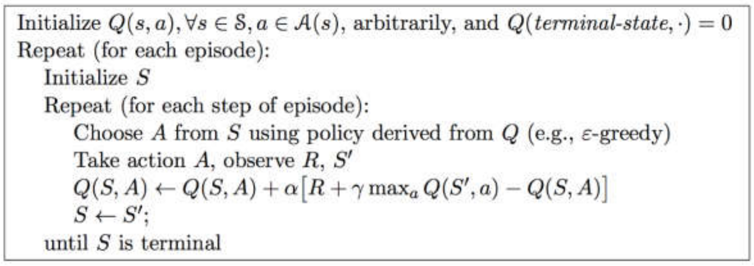


```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Repeat (for each step of episode):
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal
```

Q-Learning: Q(S,A) <- Q(S,A) + α (R + γ **max a' Q(S', a')** – Q(S,A))



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



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$;
  until $S$ is terminal
```

Both use ε-greedy policy to determine the next step to evaluate, but when updating Q(S,A) the strategy differs: Q-learning takes the action with the maximum Q(S',a), SARSA uses the same a' as already determined as next step by the ε-greedy policy. SARSA uses the same a' for the next step and for the update of Q(S,A) . In Q-Learning the a' used for the next step and the one used for update of Q(S,A) may differ. This explains the categories on-/off-policy:

SARSA uses the same a' determined by ε-greedy policy for the next step and for the update, and is in this sense "**on-policy**", Q-Learning determines the next step by ε-greedy policy, but for value update it uses the max Q(S',a). So it has two different strategies, and is in this sense "**off-policy**".

There are 2 reasons that I'll use TD(0) Sarsa:

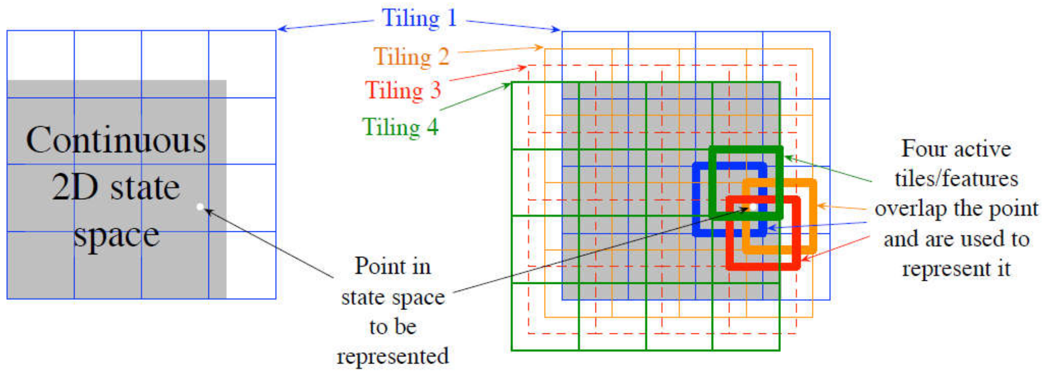
1) As we need an algorithm with strong exploration skills
2) As soon as I use linear value function for discretization, on-policy will still converge, for off-policy this is not sure. ([Silver,2015])

Handling continuous state variables

With position and velocity, the environment delivers a continuous state variables, which must be mapped to a discrete value of status. According to [wiki_MountainCar] there are two options: Discretization with Tile Coding or Value function approximation.

Tile coding is a form of coarse coding. You define some grids over a 2-dimensional layer, and according to the field which contains your status variable, the status is defined.

From [SuttonBarto, 2016]:


Continuous 2D state space
Tiling 1
Tiling 2
Tiling 3
Tiling 4
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Left side is a single tiling, so there is only one grid: the state (velocity, position) is mapped to one tile. On the left side, each pair (velocity, position) is mapped to 4 active tiles, so we have a state with 4 features. A detailed explication of tile coding can be found at [SuttonTile].

(Linear) function approximation: Instead of having a table of Q(s,a), the state-action-value is approximated by a linear function.

As I'm feeling as a bloody beginner I'll start with the simplest approach possible: this is TD(0), with single tile coding.

Benchmark Model/Evaluation 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 Design

For the project, I see as steps:

- **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, pandas, ..) this will be TD(0)-Sarsa with single tiling. 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**
  or by changing the algorithm, e.g. change the tiling to 4 active grids or use linear value approximation. Details for this step are still open because I need the experience of the basic implementation. Goal of this step is as well: deeper understanding of the theoretical knowledge.

- **Optional: Implement the solution based on Tensor flow:**
  Tensor flow is a quite different development environment promising to handle matrix, neural networks faster and promise an easy migration to a GPU. It would be extremely interesting to understand and to try for which of the possible algorithms the use of tensor flow makes sense.

Appendix

[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