Machine Learning Engineer Nanodegree

Capstone Project

Jürgen Mollen
Nov 3rd, 2019

„Space Shuttle World (Escape)“

Definition

Domain Background (Project Overview)

I would like to examine (and solve) a theme in deep leaning robotics. The idea is to navigate a space shuttle from a source position to a target position (‘docking bay’) This would be a continuous deep Q -Learning problem. I would like to adopt what I have learned in Q –Learning and neural networks to find a solution.

To simplify the domain space, the problem shall be in a 2 dimensional world. And the shuttle shall have a constant throttle, and be able to apply it to any direction by rotating along its axis. The angular speed of the rotation shall be limited.

In practice, this is a path optimization problem and there are a lot of akin problems in real life:

- Moon landing
- Air plane approach and landing
- Navigating a ship to its habour dock
- Flying a Zeppeline
- Finding the optimal trajectoy in a Formula1 cirquit …

So solving this, will show also a way to solve the related problems. Of course the physics have then to be adapted (slightly) to the specific situations. In moon landing, e..g. this is basically the additional constant g-force heading downwards. An airplane has special friction and elevation forces, etc.. Rotational inertia has to be taken into account, and also adhesing friction of tyres in case of Formula2, etc.

Here are some links to further resources dealing with some of the above topics:

https://digitally.cognizant.com/evolutionary-ai-the-next-giant-leap-for-artificial-intelligence-codex4871/

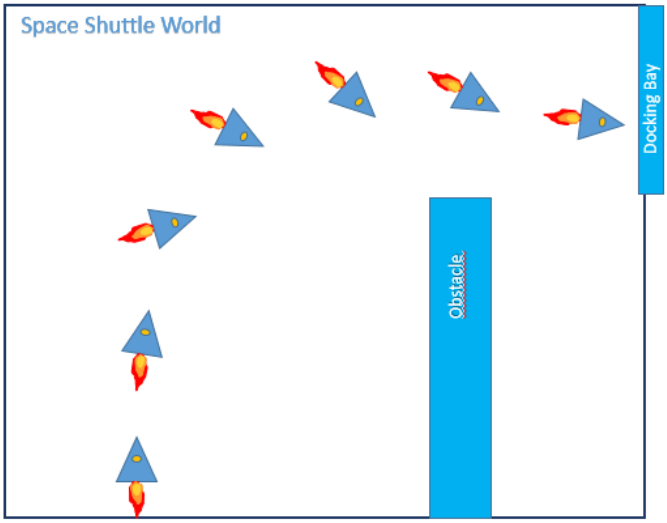
https://www.bbc.com/news/technology-48908346

https://gym.openai.com/envs/LunarLander-v2/

I have chosen this problem, because it - in its specific kind - is new and simple enough for me to write the physics and (hopefully find a good solution), all from scratch in the provided time frame.

Problem Statement and Metrics

Please see below a sketch of the problem:



The world consists of an rectangular area with an rectangular obstacle and a docking bay.

The space shuttle starts from the lower left corner heading upwards and with zero velocity. It accelerates with a variable throttle. The shuttle can rotate (with a limited angular velocity) and such steer its direction. Please note, that the velocity vector and the throttle vector usually don't have to be aligned as indicated.

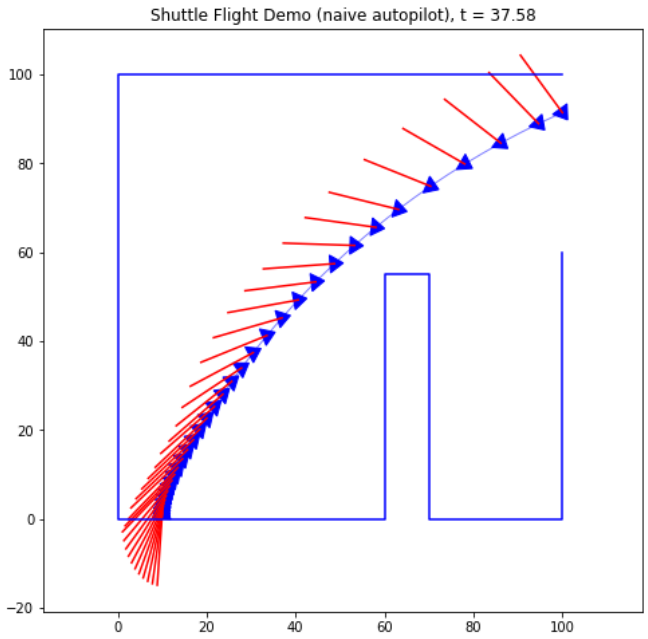
The goal is that the shuttle reaches the docking bay with whatever velocity (escapes). That is, its leaves the world to the right within the vertical limitation of the docking bay. If the shuttle leaves the world by any other means or hits the obstacle, the episode ends (too). As simplification, the spatial dimension of the shuttle is considered as a point, when verifying its allowed position. Also the thrust does not reduce the mass of the shuttle, so its acceleration is always constant for the same throttle.

- The state vector is continuous: positon vector, velocity vector, rotation.
- The action vector is discrete: rotate (left, right, don't rotate)

I have written a physical simulation of this world. Please see attached file 'SpaceShuttlePhysics.py'. It simulates the flight of the shuttle based on straight Newton mechanics. It contains also a naïve heuristically steering (autopilot). One can test a flight by starting the file 'SpaceShuttleTest.py' in a python3 console. It opens a matplot, showing the trajectory of a flight under heuristic autopilot.

Physics simulation

The following picture shows one simulated episode of the space shuttle simulator:



(It shows a flight steered by the built-in heuristic autopilot based on a naïve policy)

The physics step itself is done in the method ‘step’ that gets the actual action as parameter (rotation direction and throttle)

The autopilot is based on a simple linear policy:

- as long as the shuttle does not see the 'docking bay', it heads to the obstacles corner
- as soon as the 'docking bay is visible, it heads to its bottom corner

Problem Statement:

The flight takes 37.58 seconds - this is my benchmark:

- ***I want the system to learn itself an optimal flight. This flight shall be quicker than that of the naïve autopilot flight.***

Solution strategy:

I will try to address this by creating a Depp Q Network agent, that learns on experience and let it learn as long as to I get a stable result. This result then (hopefully) outperforms my naïve autopilot.

In order for the agent to learn, I need to define a way to reward. I will try to design the reward such that:

- Track length is penalized.
- Approach in direct line to mid of docking pay is rewarded
- Hit of a wall is penalized – the more, the further away from docking bay it has hit
- If the docking bay is reached, give extra bonus

This hopefully gives me a good enough reward with enough gradient for the agent to learn an optimal track.

Analysis

Visualization

Please note, that a visualization of the problem has been given above as a sketch and as a plot of the naive autopilots trajectory.

Benchmark

The benchmark is that the optimal learned trajectory takes less time than the 37.58 seconds the autopilot needs.

Algorithms and Techniques

Physical simulation:

I have written an environment of the domain in the class 'SpaceShuttlePhysics'. It considers the Newtonian mechanics of the space shuttle, and means to interact with this world and contains also helper methods for plotting and learning:

Relevant for the learning are the following methods:

- step: do the next time step and give reward and if the episode ends The action parameter are: 0='Steer right', 1='Dont steer', 2='Steer left'
- reset: start a new episode
- ... and several internal helper methods, like 'plot_trajectory'

The physics mechanics itself is done in this two code lines:

- velocity += acceleration * dt
- position += velocity * dt

whereas the acceleration (due to backfiring engine) heads to the rotation of the shuttle and dt is the time slice of the simulation. It is Hamiltons mechanics.

The internal representation of velocity and position is a numpy array of dimension 2 each, and the angle a float denoting rotation in 360 degree space counter clock wise from X-axis.

Using numpy has the advantage of efficient processing and straightforward algebraic description in code.

Using degree for angle has (although algorithmically little more expensive) the advantage, that it is intuitive, and that helped a lot while debugging.

Learning:

I have chosen a DQN agent (inspired by https://keon.io/deep-q-learning/) originally used for the cart pole simulation. (I have initially tried an actor critic model too, but was not able to bring it to performance.)

The algorithm works as follows :

```
DQN Algorithm:

   Initialize the memory D

   Initialize the action-value network Q with random weights

   For episode ←1 to M do

   Observe s0

   For t←0 to T−1 do

        With probability ϵ select a random action at,

        otherwise select at=argmaxaQ(st,a)

        Execute action at in simulator – this is done in SpaceShuttlePhysics
        and observe reward rt+1 and new state st+1

       Store transition <st,at,rt+1,st+1> in memory D

       Sample random mini-batch set from D: <sj,aj,rj,s'j>

       Set Q^j=rj if the episode ends at j+1,
       otherwise set Q^j=rj+ γ maxa'Q(s'j,a') - this is done in DQNAgent.replay

       Make a gradient descent step with loss (Q^j−Q(sj,aj))2

   Endfor
```

The DQN algorithm works as follows:

The agent remembers at each time step the action, that means it remembers: state (position & velocity vector & rotation angle) of this step, the action to take is decide and the resulting state found.

This is intuitively an ‘experience memory’.

After an episode ends, the agent learns from (a subset of) all of its experiences. The learning works as follows:

The agent has a separate ‘motoric intelligence memory’ (represented by neural network – I will explain a neural network further down).

The trick is that, at each state – as a result of each of the possible actions - another follow up state results, and this next state leads to an additional reward.

Now the agent combines the chain of states by summing its total actual reward plus the reward of the next state (and the next state gets its reward recursively from its over-next state and so on).

There is also a factor γ (usually smaller, but close to 1) which acts as a limitation for the look ahead of rewards. In other words, the agent is a little bit more greedy for immediate rewards than for rewards much further in the future (as we all are somehow ;-)).

This is a ‘trick’ that originates from very simple discrete state problems, but it (astonishingly) also works on a continuous state domain.

So finally, the agent learns how the total reward is for each possible action it can take in any situation, and it selects then (when you ask it) the action that would give the prospect of maximum reward.

In practice this is very similar as to train a pet to do tricks.

A neural network works as follows:

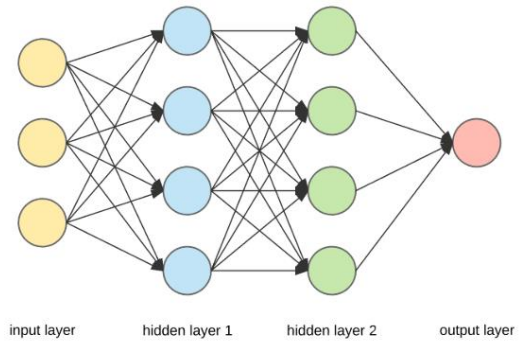
It can (also) serve to train and then perform a complex prediction task. In a broader sense, it can create an output based on a input stimulus. The idea is that you have a set of neurons, each has an activation function and a neuron can have several input signals and deliver its output signal to several other neurons. When the sum of input signals to a neuron exceeds a threshold, the neuron 'fires' itsef and sends its signal to all follow up neurons it is connected to.

This is biologically motivated by the neurons, dendrites, and axons of a brain.

Thereby a sophisticated enough neural network is able to 'learn' very complex output based on its input, and this is what we use here.

The input to our 'brain' is the state and the output cumulative rewards each of the three possible actions.

It schematically looks like that:



(from https://towardsdatascience.com)

Whereas in our case we have 5 input neurons and 3 output neurons and the intermediate layers consist of 124 neurons each.

There is a very good (from my point of view) introduction to neural network and its biological motivation for general audience:

https://becominghuman.ai/making-a-simple-neural-network-2ea1de81ec20

Methodology

Implementation

I have used a DQN Agent with continuous state and discrete action space. The solution consist of three files:

DQNAgent.py: This contains the agent.

The Q function is learned by a neural network with the following hidden layers:

- 128, activation = tanh, bias, bias_regularizer=l2(0.001), no dropout
- 128, activation = tanh, bias, bias_regularizer=l2(0.001), no dropout
- The final layer has a linear activation

It turned out, that this configuration is quick and quite stable.

SpaceShuttlePhysics.py: this contains everything for simulating (and plotting) the environment

The essential methods are:

- step: do the next step
- reset: start a new episode
- reward: calculate the reward

ShuttleEscape.py Steering and controller code – you have to start this file, to run the learning (and alternatively a Jupyter Notebook: ShuttleEscape.ipnb)

Refinement

Meta Parameters, Noise, Step Size, Look Ahead

The learning buffer was 2000 step, I tried this out by experiment. Considering, that an episode has approx. 60 steps, it seems to be sensible. I tried also up to 5000 but it was indifferent (if at all, it took more time to reach the maximum total reward). 1000 lead to instability - the reward curve seems to have more drops then.

I varied γ from 0.9 to 0.999 and stayed with 0.95. The effect of low gamma seem to be that there is not so good overall learning. Large γ (close to 1) took a longer time to learn until increase in reward was seen and also it was not 'smooth' - changes were nothing and then very quickly .

The minimum exploitation noise when selecting the gradient step for the Bellman equation, I have set to 1%. Start exploitation randomness was 100% with an episodic decay of ~ 0.997. I selected it such , that in the latter third of the run I have roughly reached the minimum.

I also added also some random noise in the starting angle and position and it seems to make learning quicker, I think that is because the 'phase space' is exploited better. And it is a kind of regularization for the first steps.

One important thing I noticed was, that the final situation of a run was very much dependent on the actions in the first ~10 seconds. That means, that a bad decision in the very beginning has a

big effect and later in an episode a (much) smaller. It means that the learning has to focus on the very first seconds, right after start or the episode.
Therefore the algorithm takes a smaller time step in this area. So to say, the agent looks in the beginning in 'slow motion' on the dynamics.

Another thing that helped on learning, was a 'look' ahead of some time slices (e.g. four steps) with a *constant* actual action.
This is what I also intuitively do when I drive a car: I look ahead considering the actual action and when I notice that I will crash (e.g. because the curve got more narrow), I know that a very bad reward will follow :-) and then I take an action to avoid it (i.e. get a better reward) and therefore I will steer more, isn't it?.

Process of improving: I did the refinement in the following order (and I describe qualitatively how they improved results):

- Tune neural network (layers, learning rate, activation function). This allowed me at all to get to stable results (especially it turned out that the last activation function *has* to be linear and that tanh-activation for the intermediate layers made the process more smooth)
- Create look ahead -> more general learning stability - quicker
- Tune buffer size -> more learning stability – less dropout and oscillations
- Tune decay rate -> more optimal tracks at simulations end
- Smearing of start position -> less dropouts
- Tune step size in the beginning of episode -> more general stability
- Tune gamma -> influence crashes did not propagate through the whole path backwards to start -> more stable

Reward Function

I have tweaked the reward function under the following assumption:

- Penalize the difference between used track length and gained approach to the mid of docking bay
- When the shuttle hits a wall, penalize with a gradient toward docking bay
- Give bonus if the shuttle leaves the world though the docking bay

In pseudo code, the reward function is as follows:

```
Reward function:
  Set reward to 0
  Remember start position
  Loop four times:
       Calculate next look ahead position
       if  <hit left bound>
            reward = reward - (200 - Y)
       else if <hit upper bound>
            reward = reward  - (100 - X)
       else if <hit right bound>
            reward =  reward + 40 - abs(Y - 85)
       else if <hit obstacle>
            reward =  reward – 100 - 10 * <distance hit to obstacle height>
       else if <hit bottom>
            reward = reward - 200

  start_dist = distance(<start_position of lookahead>, <mid of docking bay>)
  end_dist   = distance(<end position of lookahead>,   <mid of docking bay>)
```

```
reward += start_dist - end_dist – <lookahead_path_len>
```

It turned out that his is enough to find a good track.

The reward function in detail can be found in the file SpaceShuttlePhysics.py, method 'reward'.

Code

Worker code

- *SpaceShuttlePhysics.py* contains all code to the physical simulation and trajectories.
- *DQNAgent.py* contains all code that does the deep learning.

Steering code

- *MainDeepQ.py* contains the steering code to do the learning and plot the results-
- It can be run by starting the file in a Python3 console

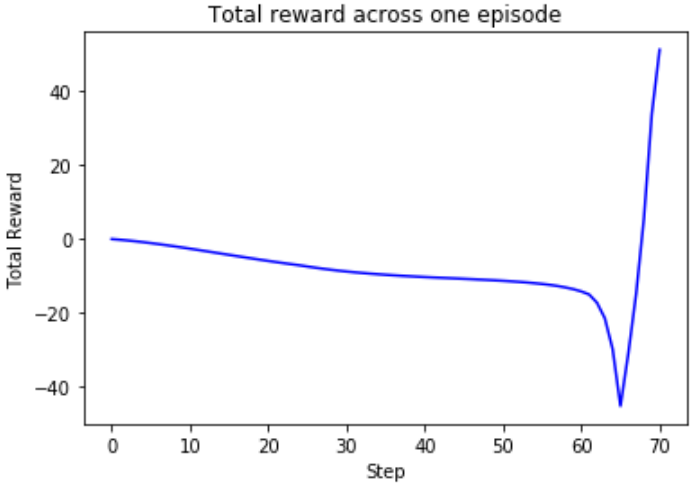
Results

Model Evaluation and Validation

Reward Curve across one episode

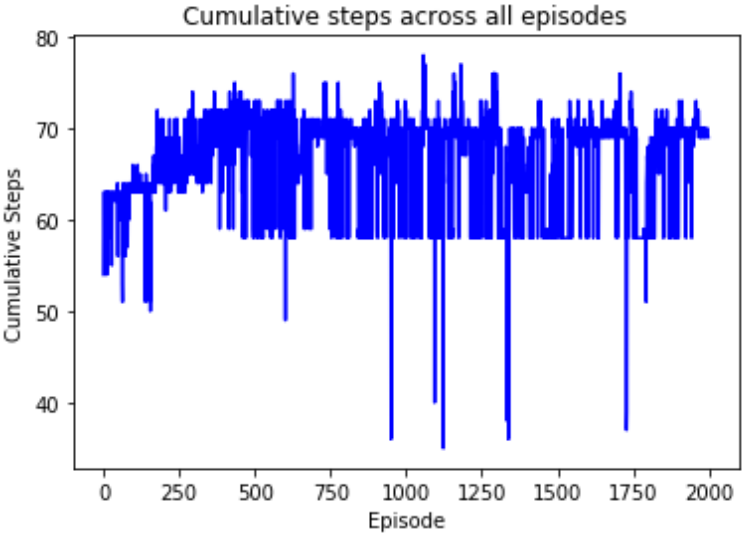
Please find below the reward curve along the steps of an episode at the last quarter of the learning:

The track obviously reached the goal. One can see that the reward decreases slowly (due to track penalty), drops a little more (that's when the linear look ahead thinks it will might hit the upper wall) and then gets a steep rise, as it became more and more clear that the docking bay will be reached.



Steps across Episodes

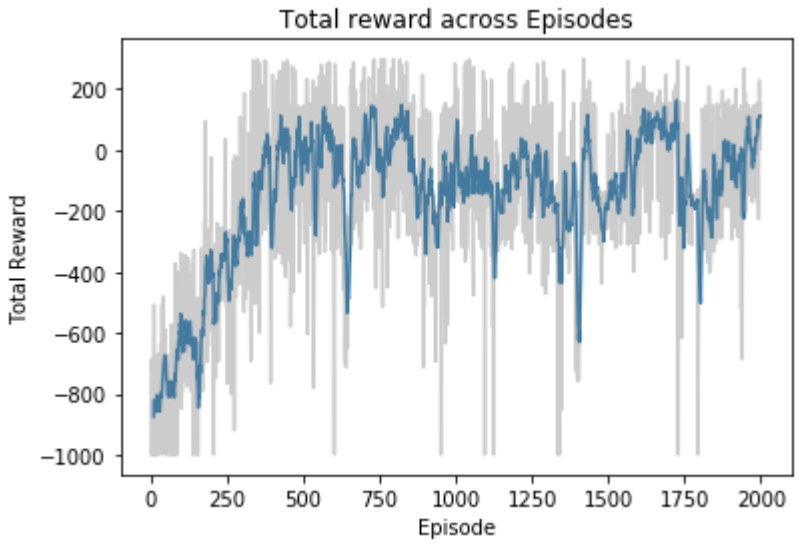
The following plot shows how much steps each episode took. One can see that it is about 60-70. It was important for me as an input for refinement (e.g. designing the buffer size or look ahead step count).



Total Reward Curve across all Episodes

Please find below the total reward curve around across all episodes:

One can see that it initially rises to a plateau and then stays there with some oscillations and drops. All in all one can see clearly that the DQN agent has learned to find its way to (or near) the target. Only then the total reward is near or above zero.



Trajectory Space

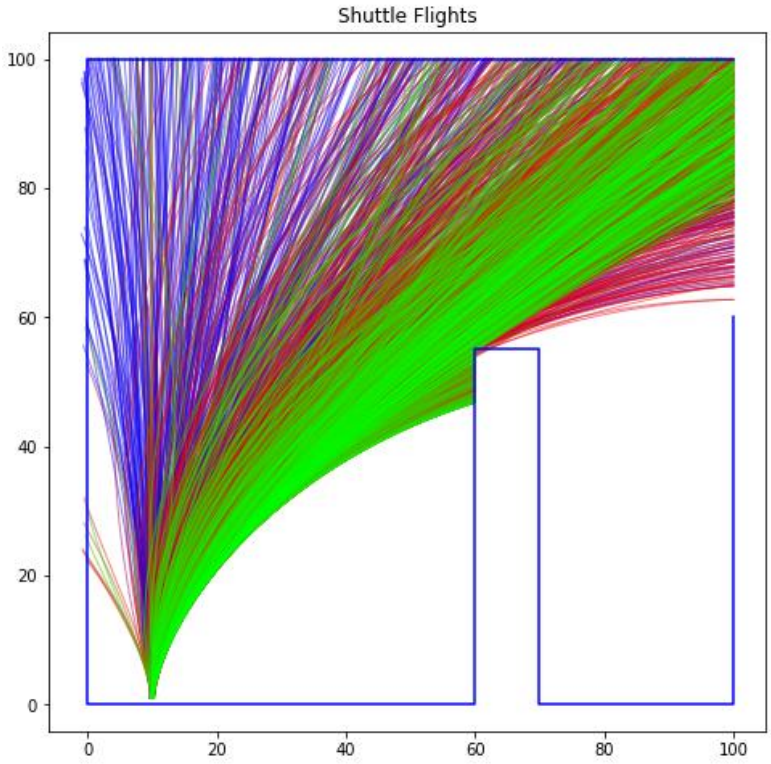
The following plot shows all trajectories during the leaning phase, from blue (start episodes) over red to green (end episodes). One can nicely see, how the AI learns better and better paths.

Darker blue lines going to read are those trajectories at the begin (and early middle) of the learning process. One can see that they are more curved and they 'exploit' the phase space.

The more read lines are already heading to the target and are still curved but consistently. They make also the error, to more frequently hit the obstacle (although one cannot see it so good because they are hidden by some of the latter green lines).

The green trajectories are more towards the end of the simulation, when the exploitation rate is at its minimum and the 'Bellman equations' straight them even more to minimize more the reward loss. Also the green lines are less spread.

This is roughly what I would expect.



Visualization of the Agents (Deep Q) learned Action Function

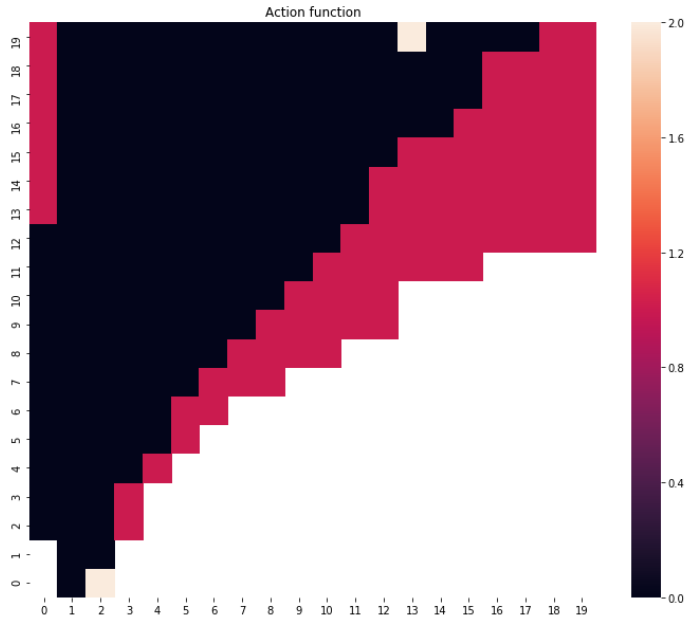
Now, I would like to see what the agents learning ‘looks like’. I want to get a picture of the action function!.

Since the state space is 5 dimensional and action space 3 dimensional, in order to be able to visualize, I need to do some simplification:

The idea is that - due to the continuous nature of the movement - each rectangle in physical space has an average velocity vector and an average angle with not too much deviation within, if the rectangle is not too big. In other words, the complete phase space is statistically steady with regard to the physical location space.

My hope is then, that I can take those average values and ask the (neural net of the) agent to predict an action for those 'averaged state space' for each tile. It allows me do to a 2 dimensional heat map that shows me the predictions of the agent.

The result is visible in the following plot:



First some explanations to the picture: The grid is 20x20, corresponding to the 100x100 world (i.e. the tile size is 5 in the 'real world')

Each cell has a color:

- Black -> turn shuttle to the right
- Red -> don't turn (stay with orientation at current angle)
- Wheat -> turn left
- White -> no trajectories, and nothing learned (phase space not exploited - this is consistent with the trajectories plot)

In order to interpret this, we have to be aware that:

The neural net of the agent forgets older experiences (e.g. due to limited buffer) The area was exploited by the shuttle over a long time with its trajectories bending more and more to the right. That means, when I ask the agent about its predicted for tiles in the top left third, I ask the *actual learning* of the agent using data of *very old* average exploitations, and therefore I do expect less accurate result.

What one has to do:

One should look only to the area covered roughly by the green curves in the 'Shuttle flights' plot. That is where the leaning stand corresponds to the exploitation trajectories, and what we see then is:

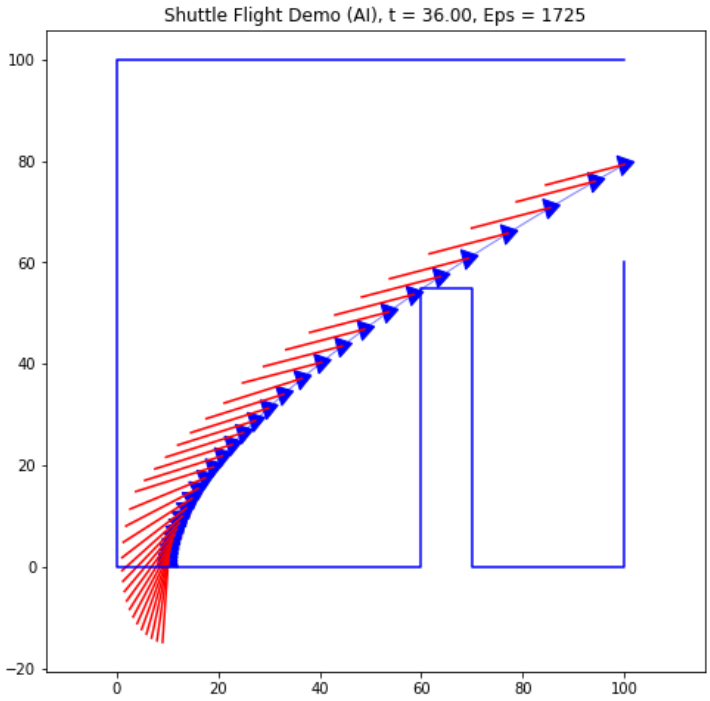
- In the beginning rotate the shuttle to the right, especially if you are late (black tiles). If you start a little too much to the right (remember, the start position is smeared), even turn a little to the left first.
- When you are moving already well to the goal, do not steer anymore (red tiles), except when you are a little of the track (to the top), then do some correction steering to the right.
-

I find this very interesting.

Justification

Please find below the best learned trajectory - that's one of the trajectories for the episodes (episode number = 1725) where the total reward curve at its plateau.

The AI - learned flight is from an episode on the plateau of the learning curve and (much) better than that of the naive auto pilot flight. It took 36.0 seconds:

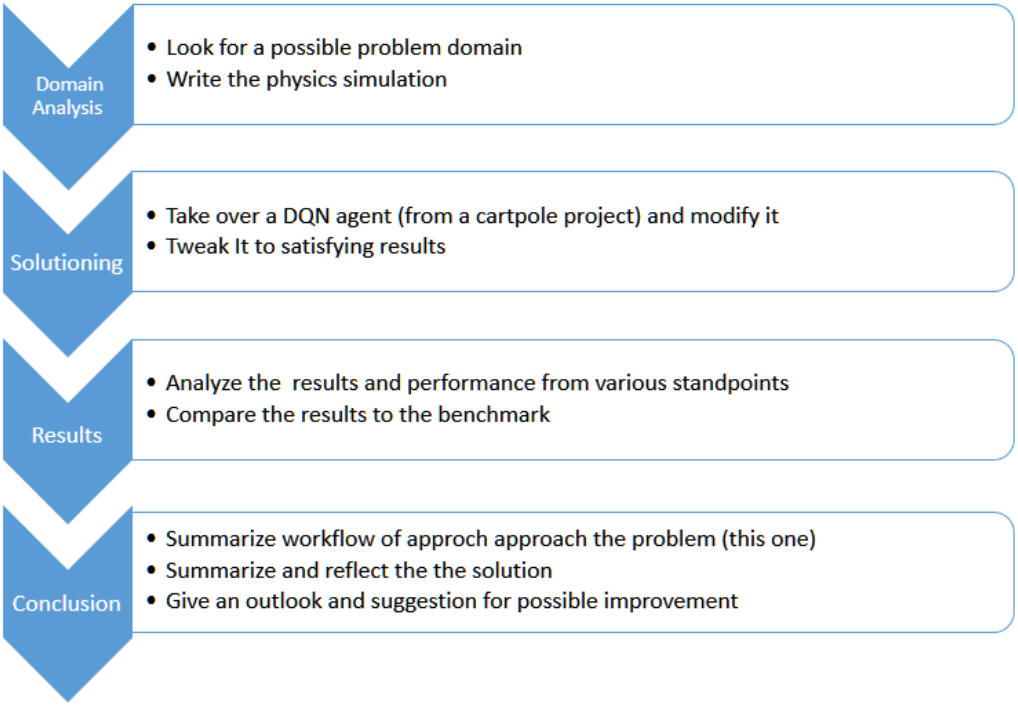


Therefore, I have reached my benchmark, that the AI flight - after the learning phase is stable - is better than the flight of my naive (linear steered) autopilot.

Conclusion

Approach to the Problem (Workflow Visualization)

I did follow this workflow to address the problem:



Reflection

To solve to problem, I followed the above sketched workflow.

- First, I tried to use an Actor Critic method to solve, but I wasn't able (due to lack of time) to get satisfying results.

- Then I tried a DQN network, that I have found as a solution to the cart pole problem (https://keon.io/deep-q-learning/) and adapted it to my problem domain.

The main difference is the following:

- the dimension of the state space is 5 (instead of 4)
- the dimension of the action space is 3 (instead of 2)

One further important difference is (I think), that the action space in my problem domain is not random at the start and the result of actions at the very beginning of an episode can have a dramatic effect on its end. This had made learning more challenging. I tried to address it by randomly smearing the start parameters a bit, looking (linear) ahead in the reward function and by reduction of step size of the simulation in the beginning, effectively allowing the agent to learn more detail in the start.

Tweaking the learning itself (i.e. neural networks layers, regularization, activation function, learning step size, exploration decay rate and memory buffer size) was also a challenge due the lot of parameter dimensions.

After I have achieved satisfying results, I analyzed the outcomes from various standpoints i.e.:

- Plot reward curves
- Plot single trajectories with the shuttles orientation
- Plot the set of all trajectories during learning phase (colored to see which trajectories are the latest)
- And I tried to look into the 'memory' of the agent

Suggestions for Improvement and Outlook

The first thing that I would try is to make the action space (more) continuous. This can be done by enlarging the discrete values. The next thing is, to really retry the actor critic model that I abandoned (due to lack of initial success). It was already running, but not really learning, maybe I did just something slightly wrong in my adaption that had a dramatic effect.

I think this problem could also be adapted to more complex (but related) problem domains. If I get the occasion or time, I would like to explore the following problems:

- Landing an airplane (to simplify in 2 dimensional world: height and X, with airflow simulation)
- Finding an optimal path in a Formular1 circuit (with drifting and territory properties)
- Sailing a boat from point A to B

What I have learned in this project, might give me a good starting point here.

All in all, I think I have learned a lot and now hopefully really understood (the basics of) Deep Q learning.

It was a lot of fun (although somewhat exhausting, because I did it in my free time).