Bio-inspired Intelligence and Learning for Aerospace Applications Report

Nico Voß - 4550749 - https://github.com/NicoEVA/BioInspired

August 31, 2021

1 Introduction

This report serves to reflect on the development and implementation of a bio-inspired algorithm for optimising control of a simplified path-finding exercise. To this extent the report will discuss the background and idea for the assignment, the theories and methods used and the creation of the resulting code.

It was decided to apply Q-Learning to a path-finding problem that represents a glider attempting to get to a target location. Firstly, a two dimensional model was created, in which the piloting agent could choose to go in either of the four cardinal directions. Later on, for the purpose of better representing the three dimensional nature of flight, it was extended to three dimensions by adding a third degree of freedom in the form of altitude.

One of the biggest issues with Q-Learning and self-learning algorithms in general is that the problem and environment need to be set up in such a way that the agent truly optimises for the desired behaviour, instead of simply cheating the system and finding valid but undesired solutions. In the light of the glider problem, this means motivating the agent to find an economic path to the target location, and thinking about the rewards and penalties to apply for e.g. violating restricted airspace or making smart use of thermals. To resemble a realistic problem it has been decided to make use of negative rewards, which will be alleviated when flying through thermals and heavily increase when violating restricted airspace. Termination statements are tied to attempted leaving of the environment and flying into restricted airspace, to prevent invalid behaviour and terminate moot runs. In the following section a short description of the Q-Learning method is given, followed by the mathematical background, the code implementation, the resulting output and the conclusion.

2 Q-Learning Background

Q-Learning is a type of reinforcement learning that consists of an agent operating in a well defined environment, being given a finite set of states, rewards and actions. By using a trial and error approach, the agent optimises its policy as it learns more about the environment. It is thus a model-free version of reinforcement learning[1].

The optimal decisions and their weights are stored in a Q-table containing the estimated sum of future rewards when choosing a specific action in the current state. This is also the source that is used for taking decisions on optimal actions, which is overruled by a random choice during training to improve especially the early exploration. Effectively, the Q-table is later the policy of the agent, which is initialised and modified during learning to create a means of extracting the best possible path from any point in the environment to the specified goal.

By having the agent complete a certain number of runs with specific settings for the learning rate (the speed with which entries in the Q-table change) and the likelihood of the random choice, the action space is explored by the agents with different behaviours, resulting in a final Q-table at the end of the learning run. An additional parameter is the discount factor, which influences the agents decision based on how much total reward can still be acquired when taking an action. All three values were kept constant in between the runs shown in Figure 1 to allow for a meaningful comparison.

In general these parameters can be adjusted to improve the agents learning process, by e.g. reducing the amount of random choices as the number of runs increases for better convergence. Similarly, the learning rate can be adjusted to either quickly or slowly react to changes, which has to be traded off against the possibility of reacting too fast to non-optimal solutions and delaying or preventing convergence. The discount factor is usually kept such that its influence is marginal, as the immediate action taken by the agent is usually of greater interest than what happens at a later point in time. For the exact values, please take a look at the supplied code.

3 Mathematical Foundation

The Q-Learning approach relies on a set of relatively simple but in combination potent formulas and constructs. The Q-table has previously been mentioned and logically takes the shape of the state space with each entry having another dimension equal to the number of possible actions at that point. Thus, assuming we have a state space with a dimension of 4x4x4 and we can go in either of the 4 cardinal directions plus up/down, the Q-table will have a shape of 4x4x4x6.

The variables that define the learning behaviour have previously been mentioned. The randomisation variable Epsilon controls how often a random direction is taken instead of following the current best choice. The learning rate determines how fast the values in the Q-table denoting the best choice are changed per run. Finally the discount factor puts a limiter on how much influence future rewards have on the decision process.

3.1 Temporal Difference

The Temporal Differences gives a method of calculating how much the Q-Value of a specific action taken in the previous state should be changed based on the learned knowledge of the agent from the current actions.

$$TD(s_t, a_t) = r_t + \gamma \cdot MaxQ(s_{t+1}, a) - Q(s_t, a_t) \tag{1}$$

In this TD is the Temporal Difference, r_t is the reward achieved by taking an action in the previous state, γ is the discount factor in the range of zero to one, a measure of how important the future rewards are. It is usually kept relatively close to one, causing a smaller effect of future rewards[3].

MaxQ denotes the largest of the Q-values available for any action available in the current state, thus the largest predicted sum of future rewards. Q denotes the value for the action taken in the previous state.

3.2 Updating Q-values with Bellman Equation

By relying on both the old Q-values and the learned information after moving to the next state, the Q-values in the Q-table are updated from the Temporal Difference making use of a learning rate to tune the effects of how fast the ideal direction can be changed[2].

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha \cdot TD(s_t, a_t) \tag{2}$$

Here, we update the Q-Value of the Q-table of the taken action from the previous state Q_{old} by adding the product of the previously calculated Temporal Difference TD and the learning rate α. The resulting Q_{new} replaces the previous value. This also points out the necessity to initialise the Q-table with a set of random, uniform or pre-tuned values.

4 Environment Setup

To begin the development first an environment had to be generated that with it's properties represents the situation of the glider. As such, it was decided to make use of negative rewards representing the sinking of the aircraft over time. Thermals were included to observer whether the agent makes use of areas that have lower penalties in favour of a direct path. Finally, restricted airspace was introduced, to add complexity and observer how the agent deals with limited decision space.

This space is defined in the code by three variables: the columns, rows and stacks. To explore the space, the initial position is randomly selected from the bounded box that is the state space. It is bounded by creating an enclosure of large negative rewards to prevent exceeding the limits of the space by tying a termination statement to said negative reward and it's location. The termination on running into a set negative reward will be used to prevent flight into restricted airspace and disregard invalid routes that would otherwise continue to be evaluated. The target point is set manually and attention has to be paid that it does not lay within any of the restricted areas to be reachable by the agent.

To create the environment we make use of the following code:

```
## Define Extent of Environment and create actions/q-values
rows =24
cols = 24
stacks = 10 #attempt 3D
q_vals = np.zeros((rows,cols,stacks,6))
choices = np.zeros((rows,cols,stacks))

# Define Modifier areas
alt_rest1 = 5
restricted_airspace1 = [5,15,0,22]
thermals1  = [12,20,15,20]
thermals2  = [7,13,7,18]

## Set up environment boundaries and varying negative rewards
actions = ['up','right','down','left","3ddown","3dup"]
rewards = np.full((rows,cols,stacks),-1)
for z in range(stacks-1):
    for x in range(cols-1):
        rewards[x,0,z]=-100
        rewards[x,rows-1,z]=-100
    for y in range(rows-1):
        rewards[0,y,z]=-100
        rewards[cols-1,y,z]=-100
    for x in range(thermals1[0],thermals1[1]):
        for y in range(thermals1[2],thermals1[3]):
            rewards[x,y,z] = -0.2
    for x in range(thermals2[0],thermals2[1]):
        for y in range(thermals2[2],thermals2[3]):
            rewards[x,y,z] = -0.2
```

```
for x in range(cols-1):
    for y in range(rows-1):
        for z in [0,stacks-1]:
            rewards[x,y,z] = -100

# establish restricted airspace
for z in range(alt_rest1,stacks):
    for x in range(int(restricted_airspace1[0]),
    restricted_airspace1[1]):
        for y in range(restricted_airspace1[2],restricted_airspace1
    [3]):
            rewards[x,y,z] = -100
#Defining where the goal/target point is
rewards[2,2,3]=100
```

In addtition the previously established methods have to be implemented in the code:

```
## Create required functions
def terminal(row,col,stack):
    if rewards[row,col,stack]==-100:
        return True
    elif rewards[row,col,stack]==100:
        return True
    else:
        return False

def startloc():
    row = np.random.randint(rows)
    col = np.random.randint(cols)
    stack = np.random.randint(stacks)
    while terminal(row,col,stack):
        row = np.random.randint(rows)
        col = np.random.randint(cols)
        stack = np.random.randint(stacks)
    return row,col,stack

def findact(row,col,stack,eps):
    if np.random.random() < eps:
        return np.argmax(q_vals[row,col,stack])
    else:
        return np.random.randint(6)

def newloc(row,col,stack,act):
    newrow = row
    newcol = col
    newstack = stack
    if actions[act]=="up" and row > 0:
        newrow -= 1
    elif actions[act] == 'right' and col < cols-1:
        newcol += 1
    elif actions[act] == 'down' and row < rows-1:
        newrow += 1
    elif actions[act] == 'left' and col > 0:
        newcol -= 1
    elif actions[act] == "3ddown" and stack > 0:
        newstack -=1
```

```
    elif actions[act] == "3dup" and stack < stacks -1:
        newstack += 1
    return newrow,newcol,newstack

def generatechoices(q_vals,choices):
    for i in range(np.shape(q_vals)[0]):
        for j in range(np.shape(q_vals)[1]):
            for k in range(np.shape(q_vals)[2]):
                choice = np.argmax(q_vals[i,j,k])
                choices[i,j,k] = choice
    return choices
```

In order to train the agent, a routine is set up to have it practice on the space for a limited number of episodes:

```
## Set up and perform training Run
epsilon = 0.35
discount = 0.8
lrate    = 0.8
ep = 0
for episode in range(15000):
    row,col,stack = startloc()
    tries = 0
    while not terminal(row,col,stack):
        aind = findact(row,col,stack,epsilon)
        orow, ocol, ostack = row,col,stack
        row, col, stack = newloc(row,col,stack,aind)

        rew = rewards[row,col,stack]
        oq = q_vals[row,col,stack,aind]
        tempdiff = rew + (discount*np.max(q_vals[row,col,stack]))-
    oq

        newq = oq +(lrate*tempdiff)
        q_vals[orow,ocol,ostack,aind] = newq
        if tries % 10 == 0:
            print(tries)
        tries +=1
    print("Reached target in EP " + str(ep+1))
    ep += 1
print("trained successfully")
```

The final important snippet is the function that makes the agent follow the learned actions from any chosen point in the environment:

```
def extrsp(srow,scol,sstack):
    if terminal(srow,scol,sstack):
        return []
    else:
        row, col, stack = srow, scol, sstack
        sp = []
        sp.append([row,col,stack])
        print(sp)
        while not terminal(row,col,stack):
            aind = findact(row,col,stack,1.)

```

```
            row, col, stack = newloc(row,col,stack,aind)
            sp.append([row,col,stack])

        return sp
```

5 Results of Runs

To visualise the behaviour of the agent, a three dimensional plot has been created, showing the ideal path from the selected start point. For this, a chosen starting point is used in a function that follows the optimal actions as dictated by the trained Q-table. In addition, a figure of the ideal directions for each state is created by plotting the vector field from the Q-values and the respective optimal actions. The generated track features additional marking for restricted airspace in red and thermals in green. Within the thermal columns the negative rewards have reduced penalties.

```
## Set desired Origin and create Graphs based on found Path
data = extrsp(20,20,7)

# create agent trace
x = [xi[0] for xi in data]
y = [yi[1] for yi in data]
z = [zi[2] for zi in data]

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')

ax.plot(x, y, z, label='Agent Trace')
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
# create modifier visualisations
ceilingx = np.arange(restricted_airspace1[0],restricted_airspace1
    [1],1)
ceilingy = np.arange(restricted_airspace1[2],restricted_airspace1
    [3],1)

wally1 = np.zeros_like(ceilingx)
wally1.fill(restricted_airspace1[2])
wally2 = np.zeros_like(ceilingx)
wally2.fill(restricted_airspace1[3]-1)
w1x, w1y = np.meshgrid(ceilingx,wally1)
w2x, w2y = np.meshgrid(ceilingx,wally2)


Zwallsx = np.zeros((np.shape(ceilingx)[0],np.shape(wally1)[0]))
for len in range(np.shape(Zwallsx)[1]):
    Zwallsx[len][:] = alt_rest1 + len

X,Y = np.meshgrid(ceilingx,ceilingy)
Z = np.ones((np.shape(ceilingy)[0],np.shape(ceilingx)[0]))
Z.fill(alt_rest1)
```

```
35
36 thermalsx = np.arange(thermals1[0],thermals1[1]+1,1)
37 thermalsy = np.arange(thermals1[2],thermals1[3]+1,1)
38 Xt, Yt = np.meshgrid(thermalsx,thermalsy)
39 Zt = np.zeros((np.shape(thermalsy)[0],np.shape(thermalsx)[0]))
40
41 thermalsx2 = np.arange(thermals2[0],thermals2[1]+1,1)
42 thermalsy2 = np.arange(thermals2[2],thermals2[3]+1,1)
43 Xt2, Yt2 = np.meshgrid(thermalsx2,thermalsy2)
44 Zt2 = np.zeros((np.shape(thermalsy2)[0],np.shape(thermalsx2)[0]))
45
46 # add modifier locations to plot
47 ax.plot_surface(Xt,Yt,Zt,color="Green",alpha=0.4)
48 ax.plot_surface(Xt2,Yt2,Zt2,color="Green",alpha=0.4)
49 ax.plot_surface(X,Y,Z,color="Red",alpha=0.4)
50 ax.plot_surface(w1x,w1y,Zwallsx,color="Red",alpha=0.4)
51 ax.plot_surface(w2x,w2y,Zwallsx,color="Red",alpha=0.4)
52
53 ax.legend()
54
55 # show compount figure
56 plt.show()
57
58 Note: github code includes code to plot best-action vector field
```

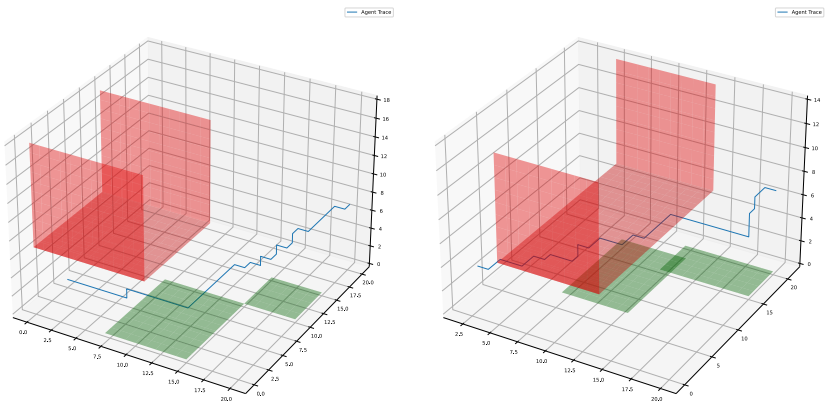
It was tested whether the agent adjusts the best learned path when the restricted airspace and the thermals are moved with the start and end points being kept constant. As can be seen from Figure 1, the agent indeed attempts to make efficient use of the available thermals without violating restricted airspace.

During the development it was also observed, that the frequency of random choices has to be kept at a low value to prevent the agent from taking primarily random actions. This issue has caused some problems in development, as the solution did not converge to a path at all with a too high set value. That being said, the epsilon part of epsilon greedy allows for more efficient exploration of the available space and is necessary to populate the Q-table. One fact that was quickly confirmed is that the agent requires significantly more episodes to reliably find a best route from a selected starting point, owed to the fact that the space to be explored has grown significantly compared to initial 2-dimensional test runs. This coupled with the random initialisation and occasional termination of the agent can occasionally lead to problems with finding an optimal route via *extrsp*, as the agent can not decide on a sole path decision if the decision becomes ambiguous at any point. A random step in any direction, followed by a return to the original functionality could solve this problem.

It is also noticeable that the limited actions of the agent restrict its motion significantly. For further exploration of the topic, one should either think about adding more available actions, such as Even.Up-Right, Up.Up-Right, Down.Up-Right or consider changing over to a continuous system, which would also allow for a more detailed description of the aircraft dynamics.

6 Conclusions

Q-Learning, albeit mostly used for two dimensional path optimisation problems has proven to be a feasible, if not optimal method to have an agent reach a goal within a 3D environment. The agent is capable of taking into account different types of environment modifiers, in this case: Restricted Airspace, Boundaries of the space to be explored and Thermals. These are defined by modifying the reward table to represent them via appropriate rewards. The results are shown in Figure 1, where the agent trace can be seen as a blue line, with the Thermals being shown as green ground-trace and the restricted airspace as simplified red box. During development of the code it was often noticeable that the agent would move in ways, that were not immediately obvious as an ideal path. However, this is to be expected when simplifying the dynamics of the aircraft to the extend that was decided upon. As such, the aircraft can be observed to climb without moving in either of the cardinal directions. A better idea might be to allow only slanted climbs, but it would likely be a better idea to develop a new continuous state system for the aircraft.



(a) (b)

Figure 1: (a) Initial Run (b) Modified(Green shows Thermals, Red Restricted Airspace)

*Note that the figures are available in higher resolution and better scale in the github repository***https://github.com/NicoEVA/BioInspired**!

References

[1] Andrew, A.: Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii + 322 pp, ISBN 0-262-19398-1. In: *Robotica* 17 (1999), S. 229–235

[2] Kampen, Erik-Jan van: *Reinforcement Learning for Flight Control.* – URL `https://bit.ly/3mIrDQe`

[3] Soper, Dr. D.: *Foundations of Q-Learning.* – URL `https://bit.ly/3mToL3b`