Radboud University

Bachelor Thesis in Artificial Intelligence

SLAR: A Framework for Walking Real World Reinforcement Learning

Authors:
Johannes-Lucas Löwe s1013635
Supervisor:
Luc Selen, Radboud University
Second Reader:
Umut Güçlü, Radboud University
January 31, 2020

Introduction

The idea of using a hexapod to traverse rough terrain has been around for decades (Belter & Skrzypczynski, 2010). One of the first autonomous balancing hexapods was developed in 1996 (Celaya & Porta, 1998). In the early days research has focused on making the robot more complex to extend the terrain it can walk on (Jianhua, 2006). Later the focus shifted to the rather inefficient distance to power ratio, preventing usage in long-distance tasks, an issue that is still unsolved (Cafarelli, December, 2017).

In early robotics, analytical approaches were used to find appropriate control solutions to make the robot walk and find an efficient gait (Celaya & Porta, 1998). However, many problems in robotics do not have an (computationally feasible) analytical solution. Therefore fuzzy logic gained traction in solving such problems. In the case of hexapod-gaits the most prominent approach was genetic algorithms. Work published focuses on the use of genetic algorithms, finding useful constraints and an appropriate fitness function (Belter & Skrzypczynski, 2010; Cafarelli, December, 2017).

Another way to find control solutions is the use of Reinforcement Learning (RL). Early attempts of this in the real world were made in the ERS-110 Sony robot (ROBOTS, Gu, & Hu, 2002). In recent years the RL-approach has boomed inside and outside the robotics domain (Jang, Sun, & Mizutani, 1997). For example, it is now possible to teach a robotic arm to flip a pancake based on RL. However, RL is heavily slowed down, or even impossible, due to the many Degrees of Freedom (DoF) that most robots have. Therefore, many RL-algorithms are primed by a teacher-model, often a human (Kormushev, Calinon, & Caldwell, 2013). For Hexapods this is very hard given that, in contrast to pancake flipping, they are not necessarily intuitive for humans. Nonetheless, RL has been applied to teaching a hexapod to walk, often on a simulated robot platform.

Thanks to advances in computation speed on personal computers, nowadays any desktop machine with reasonable speeds can be used to run modern AI solutions.

The new main bottleneck experienced for developing real world RL algorithm, as a student is access to a real robot. Since robots are expensive, usually only owned by big institutions, and well-contained. Here, I present a platform to tackle these issues of affordability, availability, ability to extend and access to a simulation.

The design of a six-legged robot was chosen for several reasons. First, it resembles an insect, which walks have been studied extensively (Cafarelli, December, 2017; Graham, 1977). Secondly, although it does make producing the robot more expensive than a four-legged design, its walking can be far more stable (Cafarelli, December, 2017). This should reduce the training time that is needed on the robot significantly, which is essential for real-world RL. Furthermore, is it still possible to transform it into a four-legged robot by dismounting two legs. (Although this might introduce stability issues)

Here I will present a simulated version of the physical six legged robot provided by the SLAR (Six Legged Autonomous Robot) Framework (http://realworldml.com/DESIGN-FILES/). The 3D printable design is be provided as a open-source framework. The specifications for Unity and the 3D printer used in this paper can be found in the appendix, as well as links to the files.

In this paper, the aim is to present a simulated version of the SLAR physical existing robot to compare different algorithms. More detail about the physical six-legged robot

which has been developed can be found on realworlml.com. On that simulated framework genetic algorithm performance is compared to reinforcement learning.

Methods

The aim of the project is to compare performance of genetic algorithms to that of reinforcement learning on learning a walking gait. Towards that end, there are three parts that need to be specified. First the simulation mechanics, second the algorithms and their implementations (RL and GA) and lastly the methods of comparison. The language of choice for the the algorithms is Python, since it easily integrates into the existing simulation eco-system of Unity (www.unity.com).

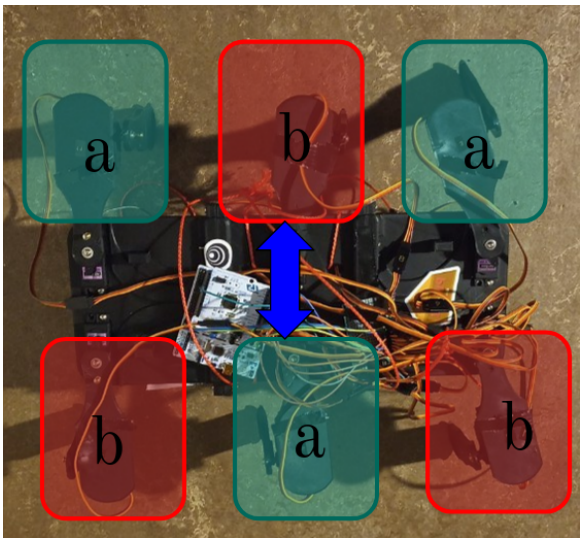


Figure 1. Grouping of the legs

Simulation of Mechanics

The simulation specifications are twofold. Firstly the general setup in the virtual environment and secondly a layer of abstraction imposed on it to reduce complexity.

The simulation environment is implemented in Unity. The robot base plate is represented as a cuboid and the legs as combinations of cylinders, connected by hinge joints. In order to decrease complexity and prevent errors from jitters in the physics engine, the environment has a two legged crawler with mirrored legs. The base-plate is constrained such that it does not rotate and has a fixed height above the ground.

The basic DOF in an unconstrained environment, is one per motor. Given six legs with two motors each results in a 12 DOF problem. On this agent, certain restrictions are imposed to reduce the state space that has to be explored. Constraints were imposed onto the freedom of the leg movement. For this, groups of three legs were connected together. Grouping can be seen in figure 1. The legs within each group (a,b) always go in parallel. Also, legs noted with b go opposite of their a counterparts. This hardware constraint is similar to the one observed by insects (Graham, 1977).

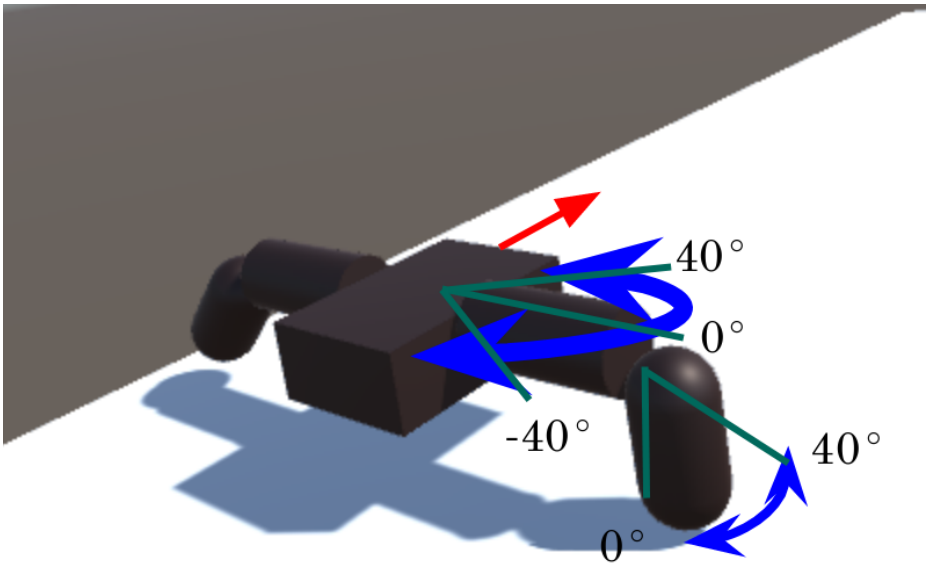


Figure 2. Simulation of SLAR based on two legs, indicated in blue: direction of joint movement, indicated in green: angles and limits, indicated in red: intended direction of movement

Furthermore the legs only have two joints instead of the three originally intended and the four usually observed in insects. Lastly, by imposing the groups to be mirrored, the problem is broken down to its simplest form with two Degrees of Freedom. This simulation setup is shown in figure 2.

To sum up, the robot has two angles which can be controlled through the algorithm. Firstly the shoulder-servos (motors on the base-plate), that control movement of the upper legs in parallel to the body plate. Secondly, knee-servos, that control the movements of the lower leg. Ergo there is an angle for the shoulders that can be observed on the servos mounted to the body and an angle for the knees, observed on the servos mounted to the upper segments of the legs.

In order to establish a communication between the hardware and the algorithmic implementations in python communication the Unity ML-Agents plugin is used (https://github.com/Unity-Technologies/ml-agents). Also by using this plugin, it allows Unity to optimize for physics performance over graphical rendering. This considerably speeds up the simulation.

Reinforcement Learning Agent

For Reinforcement learning, a custom implemented Q-table algorithm is used. The algorithm consists of four parts: input, discretization, output and reward.

As input to the algorithm the current positioning of the legs is given in angles. In order to perform any action, the input needs to be discretized.
The two angles are put into bins of 20 (shoulder) or 10 (knee) degrees. This resulting 'state'

#	DOF	Where
6	1	shoulders on base plate
6	1	servos on knies
Total		6+6=12

(a)

#	DOF	Where
1	1	shoulders on base plate
1	1	servos on knies
Total		1+1=2

(b)

Table 1
Degrees of Freedom for the SLAR framework, (a) without imposed constrained, (b) with insect walking inspired constraints

is represented as a table, where the first servo position (shoulder) is the column and the second one (knee) the row. In order to move the robot, i.e. the servos, at any decision step the four actions that can be taken are: move up a row, move down a row, move a column to the left, move a column to the right. These actions navigate through the table, creating a target state, which the robot will try to achieve until the next decision step. In a fixed time interval (every 20 frames), a decision for the next action can be made. The output is generated by using the target-state to determine the desired position of the servos. This target-position is then send to the environment.

Finally, a reward is provided by the environment. In order to stabilize the reward function, total movement along the desired axis (x) is integrated over the last 5 frames.

Performance of the RL algorithm is evaluated with test runs, where only exploitation is used ($\epsilon = 0$).

Genetic Agent

The second implementation is a genetic algorithm. It uses multiple training agents with multiple generations to determine a good strategy. The fitness of each agent is determined by the overall distance walked in the right direction. In order to stay close to the reinforcement learning problem, the same angle discretization as described in the RL algorithm is used. An individual in these generations consists of a certain chain of actions. These chains of actions are executed one after the other with equal amount of time (20 frames) between them. This means each action will generate a new target state to be achieved.

One agent, the one with the highest fitness is kept for the next generation. The rest is generated through pairing. For pairing, parents are drawn with a probability based upon their fitness (i.e. a parent with twice the fitness is twice as likely to be drawn) until all slots are filled with children.

For generating children, the two parent chains of action are combined via crossover and mutations may occur. Crossovers are the random insertions of actions from one chain into the other on the according position. Mutations are defined in a similar fashion and each mutation is the change of one action in that chain to a random new one.

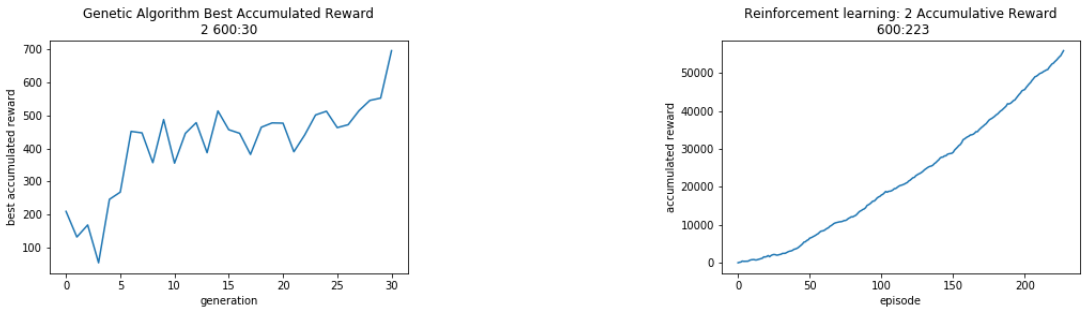
One big limitation is that this approach might not end up with circular solutions. This means that, though a action chain produces a good result, when repeating it multiple times the edge of the state-space might be reached, resulting in less beneficial behaviour. Through training in episodes longer than the time needed to execute the action chain once and therefore repeating the chain multiple times, individuals should emerge that have a circular or nearly circular action chain.

Comparison

In order to compare performance of reinforcement learning vs genetic algorithms, first a criterion has to be defined. This however needs a clear definition of the goal to be suitable. For this paper, the goal is to learn walking. However, walking is not a well defined property, therefore it will be abstracted to moving in the desired direction with reasonable performance. Here reasonable performance is defined as achieving an accumulated reward of 600 per episode (one episode is 400 steps). 600 was determined through manual experimentation.

Given this definition, comparison will be made on two levels, which are representative of two different interests. First, the wall-time needed to train on the same machine is used. This is relevant if trying to achieve a real-world application where the cost for the solution is mostly dependent on the time needed. It is used by other papers as a standard means of comparison (Zhang & Zaïane, 2017). However, wall-time factors in heavily the specifics of the environment used for simulation. This flaw becomes especially apparent since reinforcement learning and genetic algorithms have fundamentally different types of simulation needs. Because, whereas reinforcement learning is mostly dependent on one agent making many decisions consecutively, genetic algorithms try to evaluate many individuals at the same time to increase the gene pool. Therefore, a second measure is used, that factors in the steps that need to be taken by the agents, thereby also representing the effort if all training were to be done in the real world. This means every step of any agent involved in training will be counted and added towards the end cost. So nine agents taking one step in the genetic algorithm is considered of the same cost as one agent taking nine steps in the RL algorithm. In order to minimise the effect of randomness, multiple runs will be made.

Results



(a) (b)

Figure 3. Reward Graphs for RL and GA, (a) shows an example of the best reward history from a genetic algorithm, (b) shows the example accumulative reward plot for RL

Figure 3 (a) shows the best reward obtained in an generation of the genetic algorithm for all training episodes. As can be seen in the figure, the genetic algorithm finds better solutions in jumps. This behavior is to be expected, as it is only every so often that a good combination or mutation is found. However, what is unusual for a genetic algorithm is the little valleys in the phases between the peaks. In this case this is most likely to blame on the inherent randomness inside the unity physics engine. As is also visible, through continuous

optimization it is possible to achieve higher than the chosen benchmark of a reward of 600. For detailed information on the reward histories and exact numbers, please see the tables A1, A2 in the appendix A.
Figure 3(b) shows the accumulative reward for reinforcement learning plotted against the training episode. As expected the reward keeps increasing, signifying that the algorithm learns.

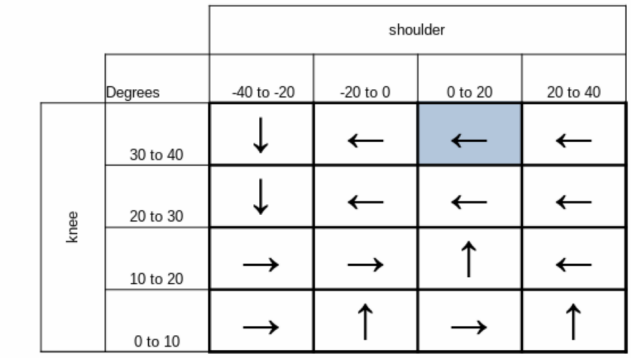
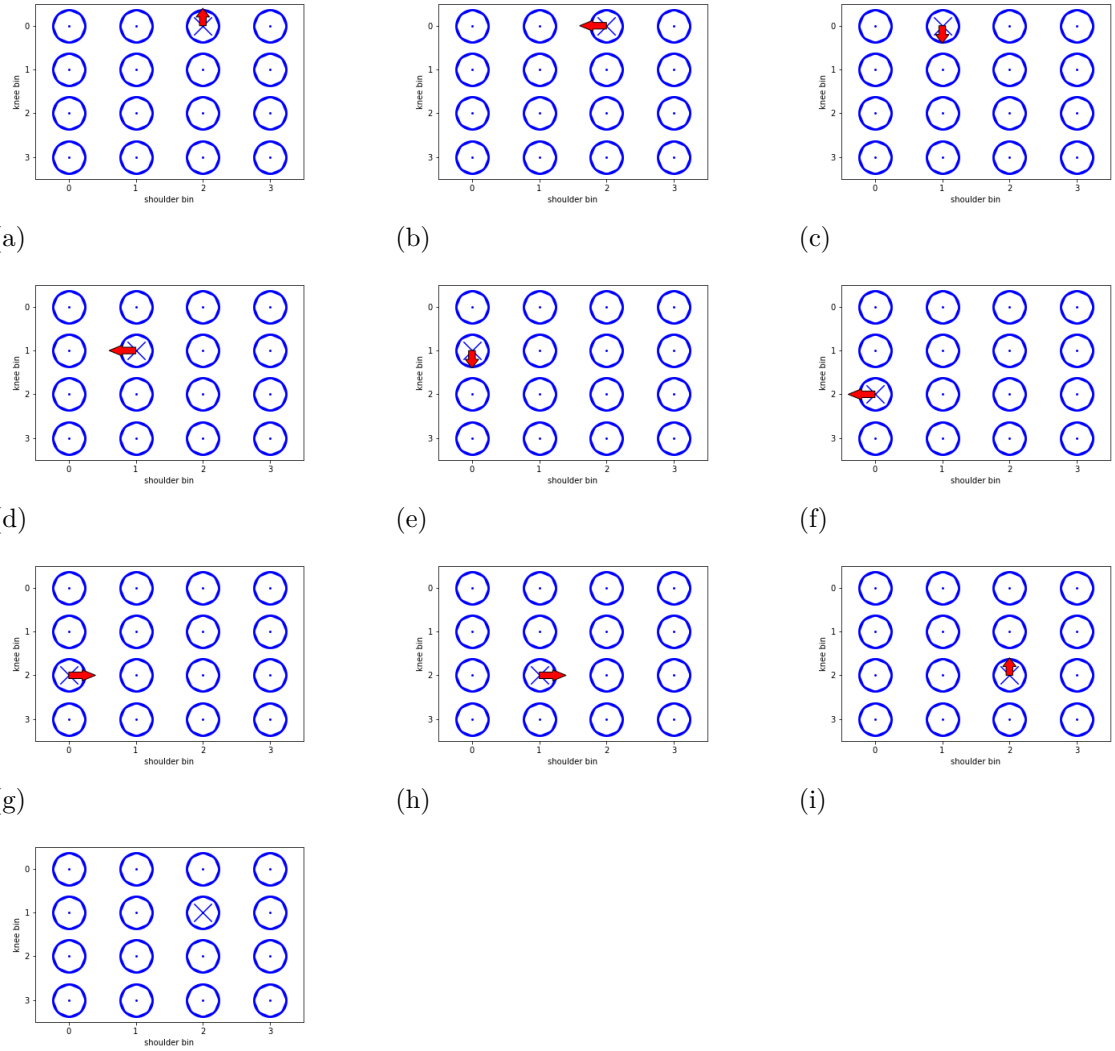


Figure 4. RL Q-table: best actions indicated by arrow pointing to the resulting state

Another interesting thing to look at, are the movements learned by the different algorithms. In order to visualize that, figure 4 shows the best action in each of the states of the Q-table for the RL algorithm. Cells represent states and the arrows point towards the state that will result after the action in that state is taken. The starting state is marked by a blue background. Here the x coordinate of the position represents the bin of the angle the shoulder is in. Left means the upper leg is inclined towards the front of the robot and right towards the rear. The y coordinate is the bin of the angle of the lower leg, where the top means the leg is lifted and the bottom that the leg is perpendicular to the ground. Given the start state at $x = 2$ and $y = 0$ the movement pattern can be traced. The upper leg is first moving towards the rear of the robot, then the lower leg lowers down until it is touching the ground. After that, the upper leg is moved towards the front until near the very front, at which point the lower leg is lifted and the upper leg moved back again. Here it crosses the point where it lowers the lower leg again and is now effectively in a loop. This movement is how one can generally describe walking in a hexapod, proving that the algorithm indeed learned walking. A reason for the lower leg to not go all the way down, is that it is not necessary. Going one state down appears to be enough to touch the ground. Going further would actually decrease the walking speed, since there are only a limited amount of decisions that can be made every episode. Why the upper leg does not go all the way to the front in this movement is an interesting question. There are two simple reasons for it not being the case. First when lifting the leg whilst in the most forward position, a backwards movement might be provoked, generating a negative reward and thereby an avoidance behaviour. Second it could simply not be necessary to achieve the desired goal and therefore might still be learned if trained longer. As expected, a circular behaviour pattern has emerged.

For the genetic algorithm the visualisation consists of a series of states that are the result from the action sequence learned. Figure 5 show the states resulting from the actions



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

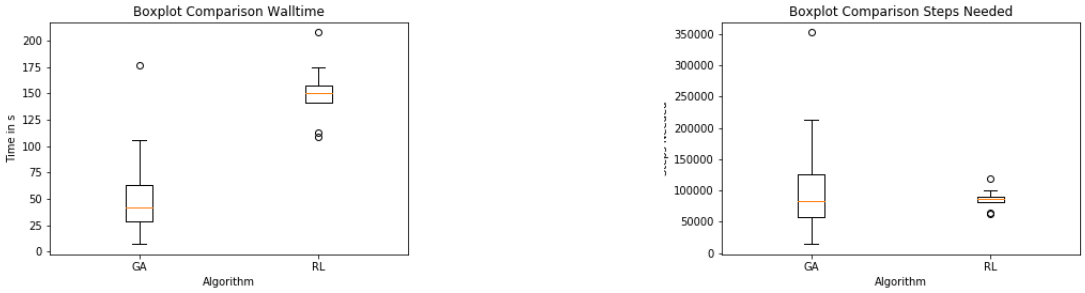
(j)

Figure 5. One example of states as learned by the genetic algorithm, (a)-(j) states in consecutive order, with next action indicated by a red arrow

of one example genetic algorithm solution. Each circle represents a possible state, and the 'x' the current state by the algorithm, with the red arrow indicating the next action. The y axis is inverted, i.e. 0 is at the top, to fit the layout used for RL state table. First off, it is possible to observe that the current state (cross) sometimes does not actually change. This is possible if the taken action causes a state outside of the state space (e.g. going up when already at 0). This can be beneficial to achieve a better circularity. In general it is visible, that the sequence is similar to the one from RL. Also the state at the beginning (2,0) is very similar to the state at the end (2,1), meaning that this is indeed a near circular movement. Since the first action is to move up (that's why the x does not change between state 0 and state 1), the movement will actually be repeated exactly. This means that indeed GA has

found a perfectly circular chain of actions.

Unfortunately it was time-wise impossible to do actual learning on the hardware of SLAR. However, the movements learned in the simulation were transferred onto SLAR. Though the result was not the most optimal way of walking, it appeared to be sufficient to gain decent forward motion. This might also be due to the lack of fine-calibration of the servos. For video material, please refer to the website *realworldml.com*.



(a) (b)

Figure 6. Box-plots comparing RL and GA performance, (a) based on wall-time, (b) based on steps taken

Looking at the solutions found, there appears to be no big difference in what the algorithms learned. So, in order to get a better understanding of their performance, the comparison metrics (wall-time and steps taken) can be used. Figure 6 (a) shows the box-plots of the GA and the RL for wall-time in seconds. It is clearly visible that the GA performed much better overall (mean GA:$\sim$ 53, mean RL: $\sim$ 153). However the standard deviation of the GA is much bigger than the one of the RL approach (GA: $\sim$ 45, RL: $\sim$ 26). Using an independent t-test without assumption of equal variances, the resulting p-value is $5.2 * 10^{-6}$, indicating a significant difference between the two algorithms. Figure 6 (b), shows the box-plots again for both approaches, but this time for the steps taken measure. The RL algorithm performance is now similar to that of the GA algorithm (mean GA: $10.6 * 10^4$, mean RL: $8.6 * 10^4$), now slightly favoring RL. Using an independent t-test without assumption of equal variances, the resulting p-value is 0.4, indicating a non-significant difference in means. However, this time there is a very clear indication that the RL algorithm performs much more consistently. The RL has a standard deviation of $\sim 15.5 * 10^3$. The GA based Agent has a lot of variation in the performance, marked by a big standard deviation of $\sim 90.3 * 10^3$, 6 times as big as the one of RL. So, overall GA has a clear advantage when measured in wall-time. However, RL has slightly better performance when considering the steps that agents have to take. In both cases, RL produces more consistent results, although the results show genetic algorithms to be favorable in terms of wall-time. Given infinite time and the right meta parameters both algorithms are expected to come up with the same/equally good action sequence.

Discussion

The aim of the project was to use the developed framework to compare RL and GA. Although a comparison was made, there is still ample room for improvement. Firstly the

RL approach used is rather simple and relies on a mostly consistent environment, which is not always the case in the simulations and certainly not in the real world. Consistent, in this case, refers to the fact that actions always have the intended outcome, and if performing the same state transition have the same reward. One issue can arise, for example, when moving to a state and immediately back to the previous state in the next decision. This can cause the maximum reward for the transition back to be taken from the action leading to that very state. If the environment is inconsistent (e.g random high rewards), then this might lead to a beneficial loop of back and forth that does not add any performance.

One other point of concern can be seen when looking at certain plots of the best reward per generation for the genetic algorithm. In graphs 1, 7, 9, 10 in appendix C it shows clearly a big jump in accumulated reward from a good generation (best accumulated reward: $\sim 400 - 500$) to one where the best accumulated reward is much worse (best accumulated reward: $\sim 200 - 300$). This appears to be more than just jitters in the physics engine. Further investigation is needed to determine the exact cause. One issue that might explain this is, that the resetting occasionally fails to actually regenerate the exact positions of the objects in the Unity Scene. Since the ML-Agents plugin for Unity is still in beta, this is not impossible. However there might also be other reasons, including inconsistencies in the reward accumulation.

Transfer to the real world seems in and of itself possible. More research towards stabilizing the reward in combination with more efficient algorithms is necessary before a founded conclusion about the success of SLAR can be drawn. Together with that, it should be mentioned that the current approach makes very big simplification of the state space. Overall, the results show that in principle it is possible to use the platform and the simulations to implement and compare algorithms.

All in all, it can be said that a proof of concept for the usability of the framework has been achieved, although a lot of steps still need to be taken to transform it into a stable and user friendly standard.

References

Belter, D., & Skrzypczynski, P. (2010, 03). A biologically inspired approach to feasible gait learning for a hexapod robot. *Applied Mathematics and Computer Science*, *20*, 69-84. doi: 10.2478/v10006-010-0005-7

Cafarelli, R. (December, 2017). *Energy-efficient gait control schema of a hexapod robot with dynamic leg lengths.*

Celaya, E., & Porta, J. (1998, 08). Control of a six-legged robot walking on abrupt terrain.

Graham, D. (1977, Dec 01). Simulation of a model for the coordination of leg movement in free walking insects. *Biological Cybernetics*, *26*(4), 187–198. Retrieved from `https://doi.org/10.1007/BF00366590` doi: 10.1007/BF00366590

Jang, J.-S., Sun, C.-T., & Mizutani, E. (1997, 11). Neuro-fuzzy and soft computing-a computational approach to learning and machine intelligence [book review]. *Automatic Control, IEEE Transactions on*, *42*, 1482 - 1484. doi: 10.1109/TAC.1997.633847

Jianhua, G. (2006, Dec). Design and kinematic simulation for six-dof leg mechanism of hexapod robot. In *2006 ieee international conference on robotics and biomimetics* (p. 625-629). doi: 10.1109/ROBIO.2006.340272

Kormushev, P., Calinon, S., & Caldwell, D. G. (2013). Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, *2*(3), 122–148. Retrieved from `https://www.mdpi.com/2218-6581/2/3/122` doi: 10.3390/robotics2030122

ROBOTS, W., Gu, D., & Hu, H. (2002). Reinforcement learning of fuzzy logic controllers for quadruped. *IFAC Proceedings Volumes*, *35*(1), 91 - 96. Retrieved from http://www.sciencedirect.com/science/article/pii/S147466701539248X (15th IFAC World Congress) doi: https://doi.org/10.3182/20020721-6-ES-1901.00827

Zhang, S., & Zaïane, O. R. (2017). Comparing deep reinforcement learning and evolutionary methods in continuous control. *CoRR*, *abs/1712.00006*. Retrieved from http://arxiv.org/abs/1712.00006

Appendix A
Data from Simulations

Run	Episodes
1	156
2	203
3	223
4	206
5	162
6	225
7	202
8	223
9	297
10	250

Table A1
Episodes needed by the Reinforcement Learning Agent to achieve a accumulated episode reward of 600

Run	Generations
1	98
2	31
3	4
4	35
5	16
6	30
7	19
8	5
9	16
10	4

Table A2
Generations needed by the Genetic Algorithm Agent to achieve a accumulated reward of 600

Appendix B
Reinforcement Learning Accumulative Rewards for Simulation Data

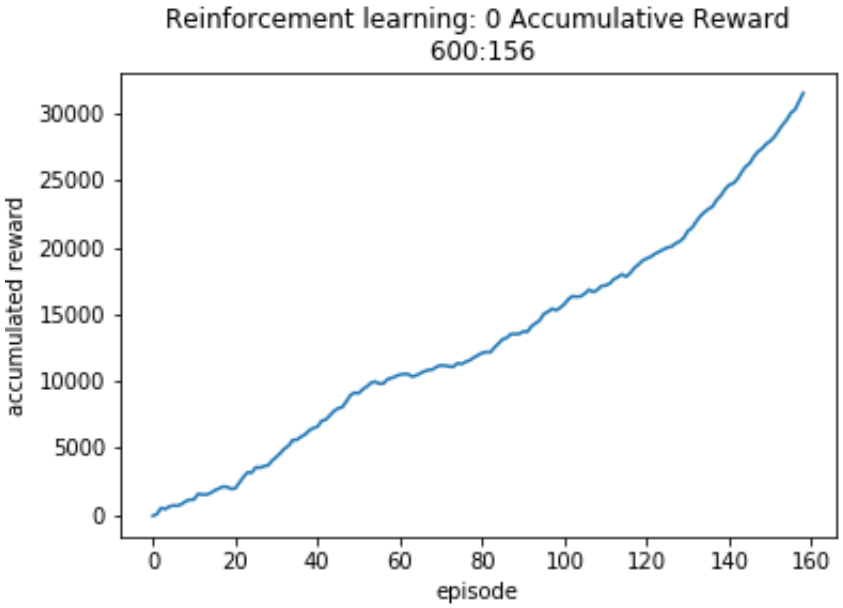


Figure B1. Accumulative Reward of run 0

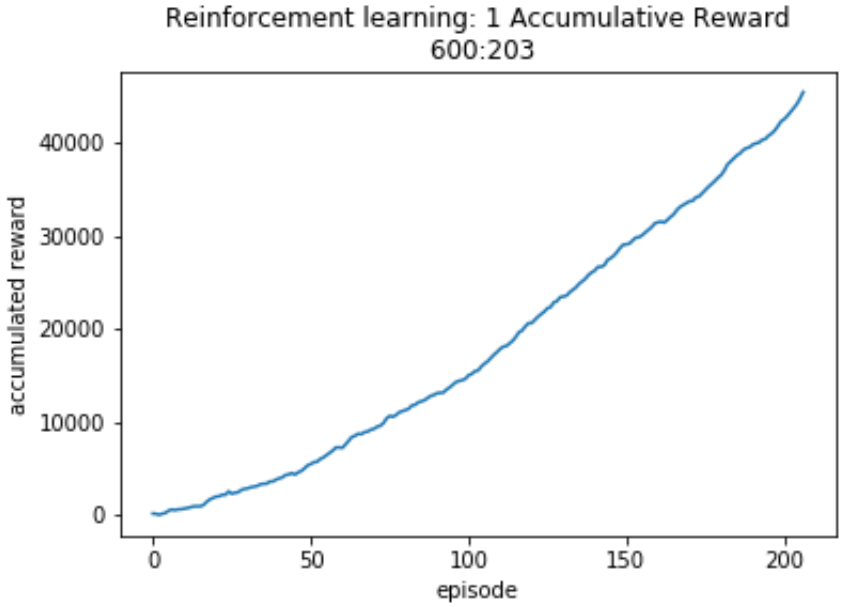


Figure B2. Accumulative Reward of run 1

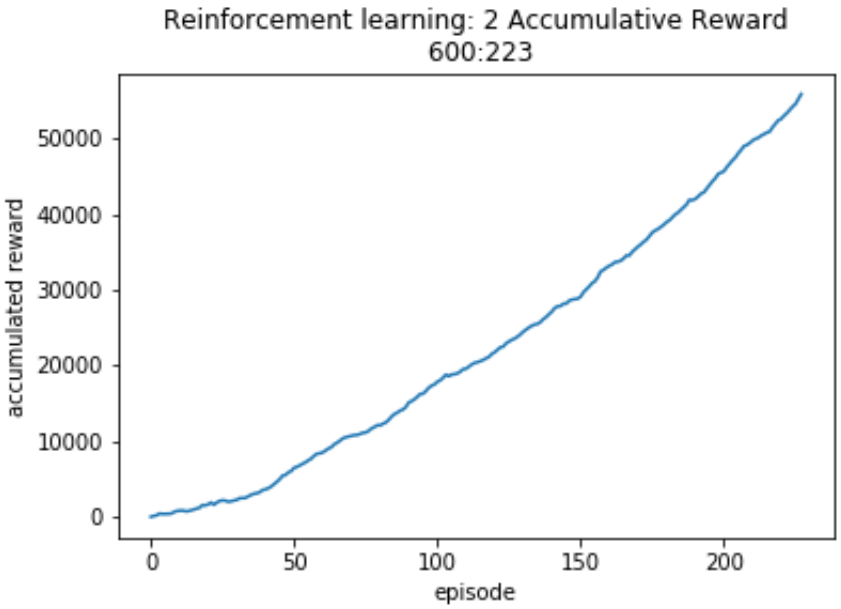


Figure B3. Accumulative Reward of run 2

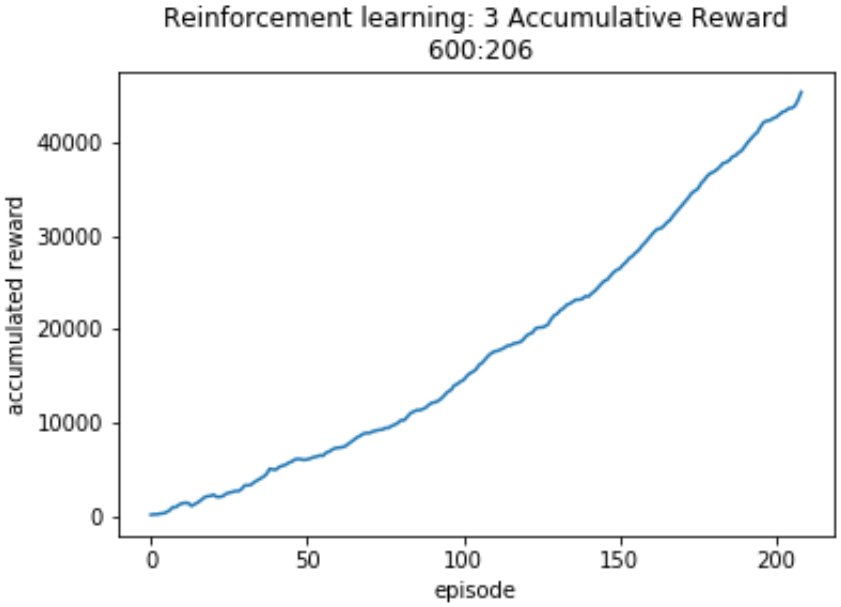


Figure B4. Accumulative Reward of run 3

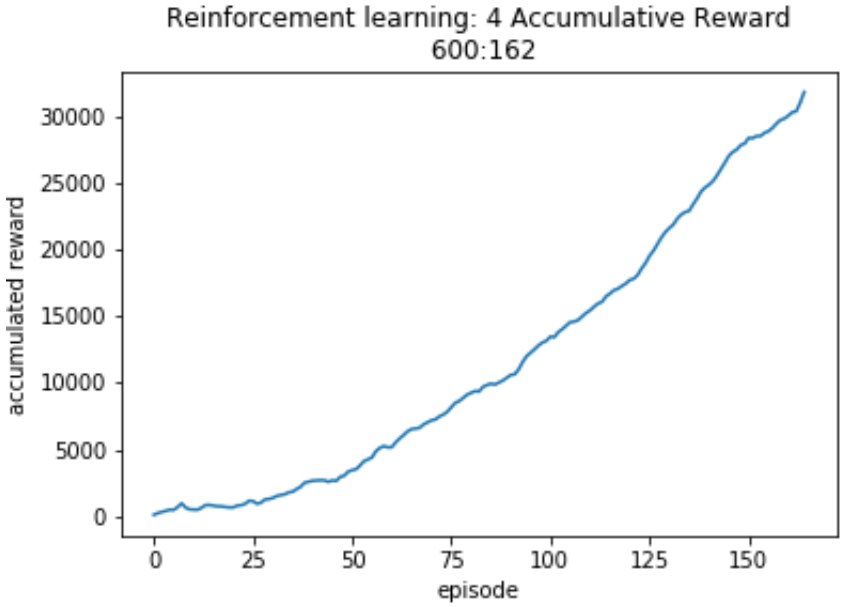


Figure B5. Accumulative Reward of run 4

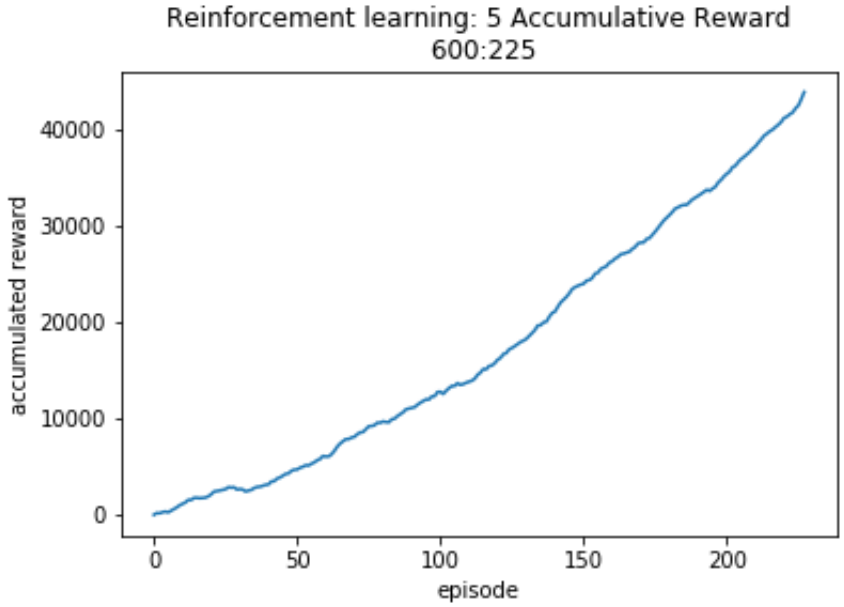


Figure B6. Accumulative Reward of run 5

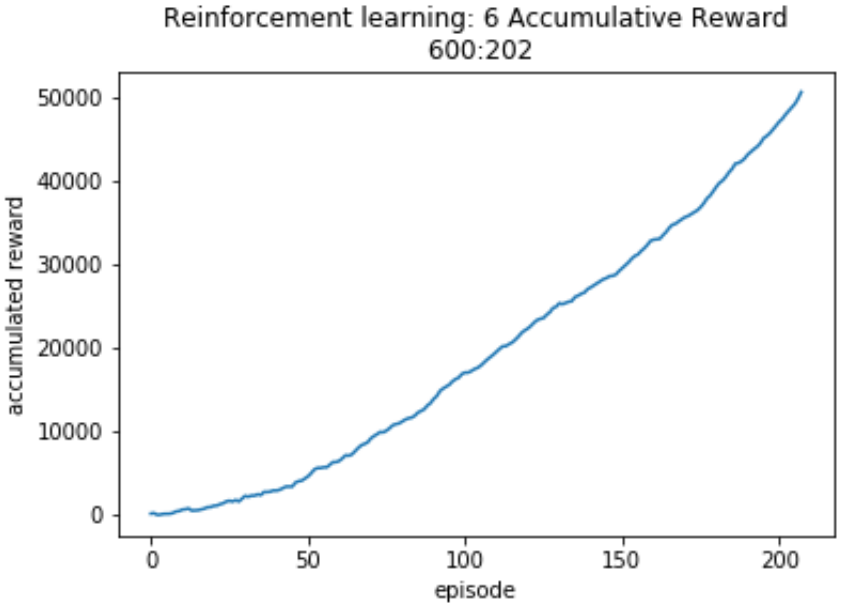


Figure B7. Accumulative Reward of run 6

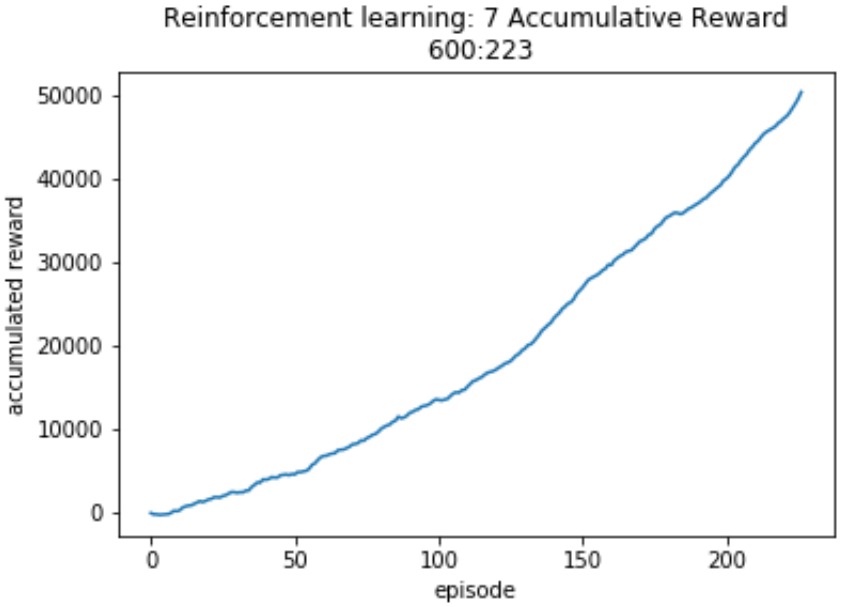


Figure B8. Accumulative Reward of run 7

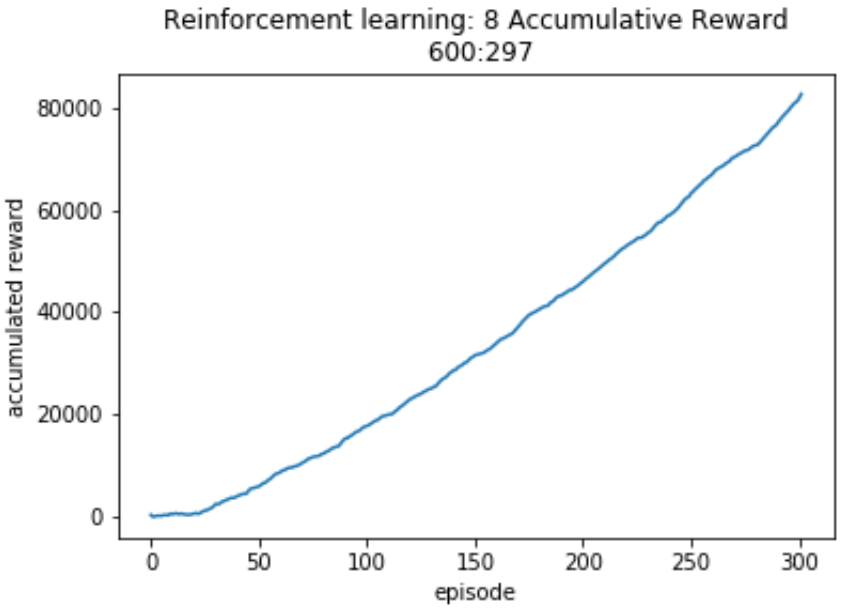


Figure B9. Accumulative Reward of run 8

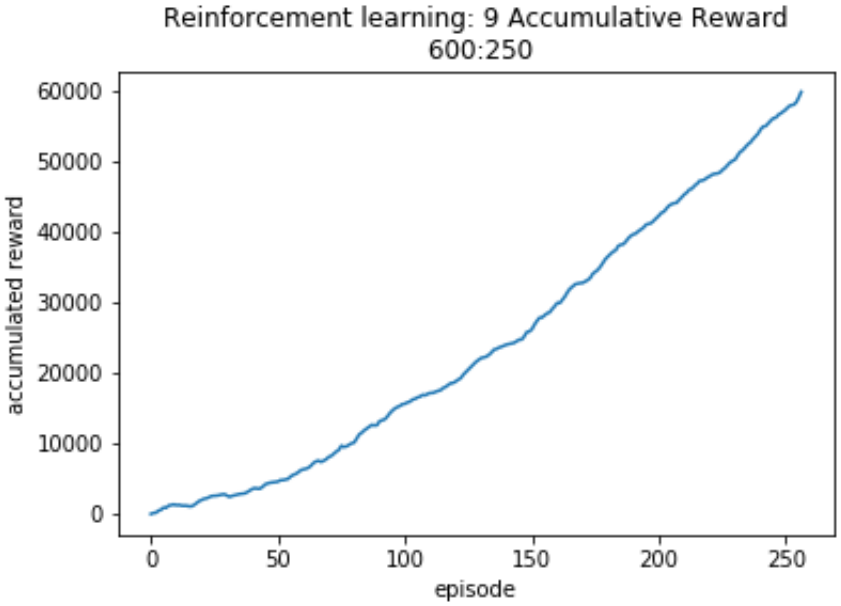


Figure B10. Accumulative Reward of run 9

Appendix C
Genetic Algorithm, best Accumulative Rewards for Simulation Data

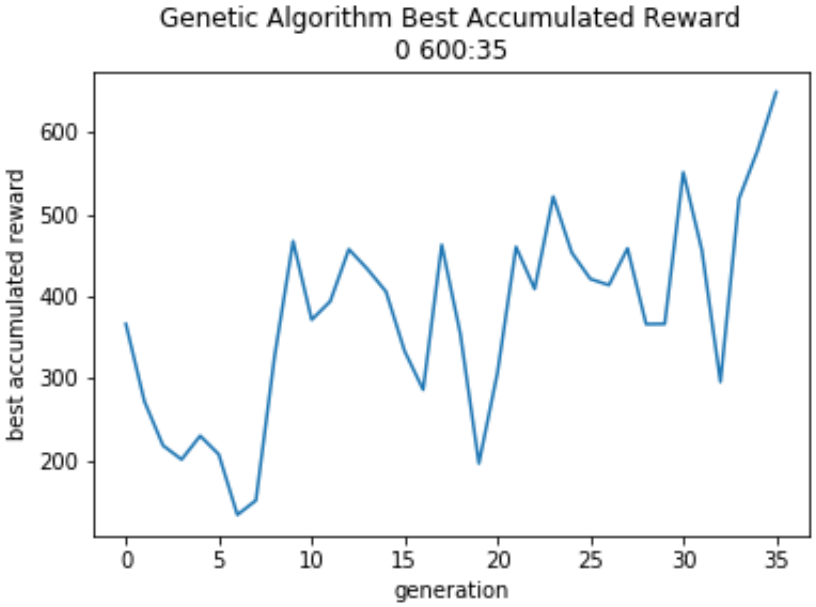


Figure C1. History of best rewards for genetic algorithm run 1

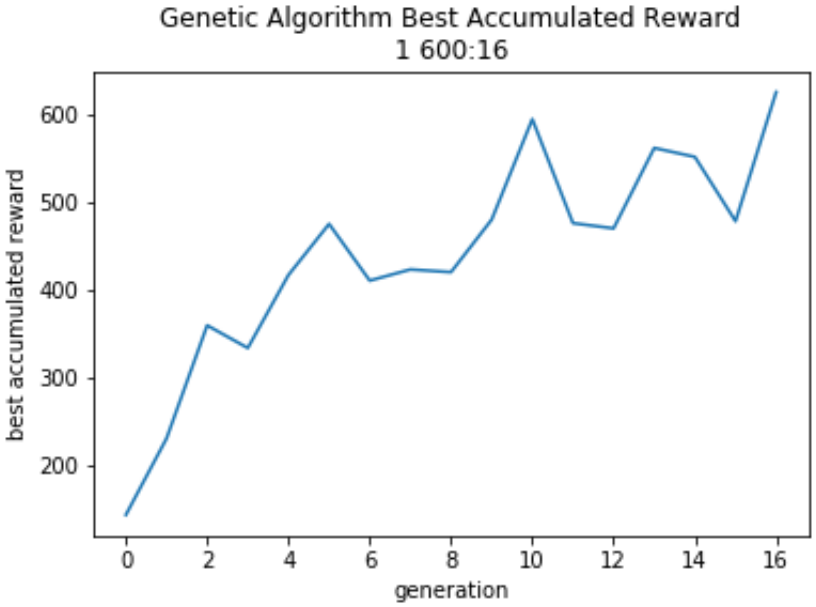


Figure C2. History of best rewards for genetic algorithm run 2

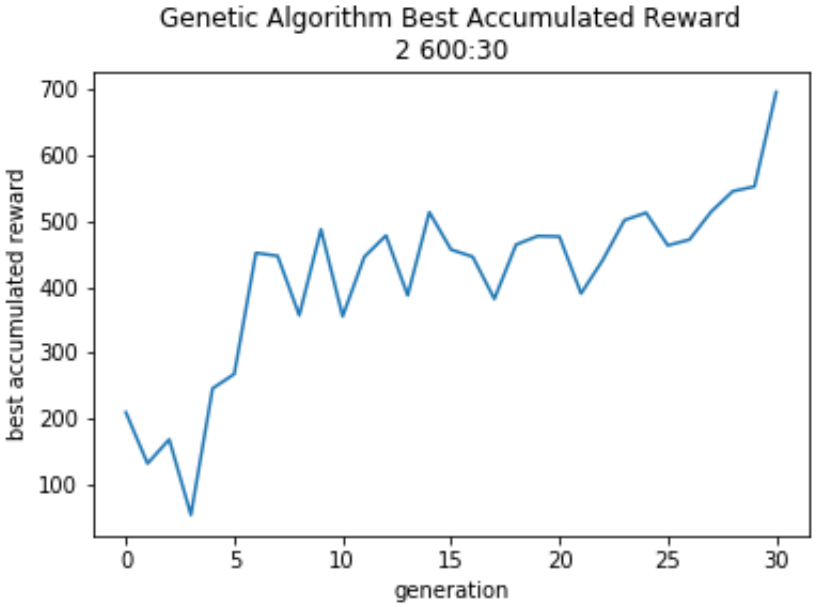


Figure C3. History of best rewards for genetic algorithm run 3

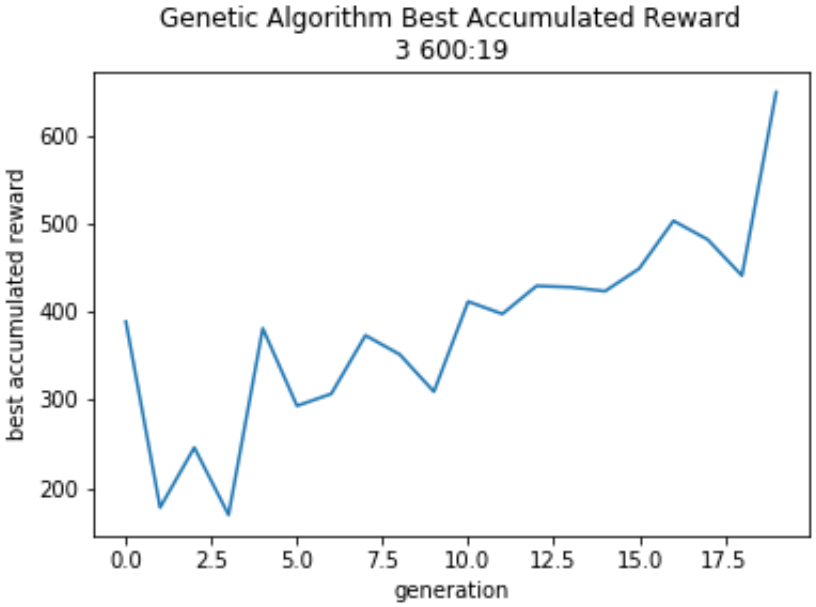


Figure C4. History of best rewards for genetic algorithm run 4

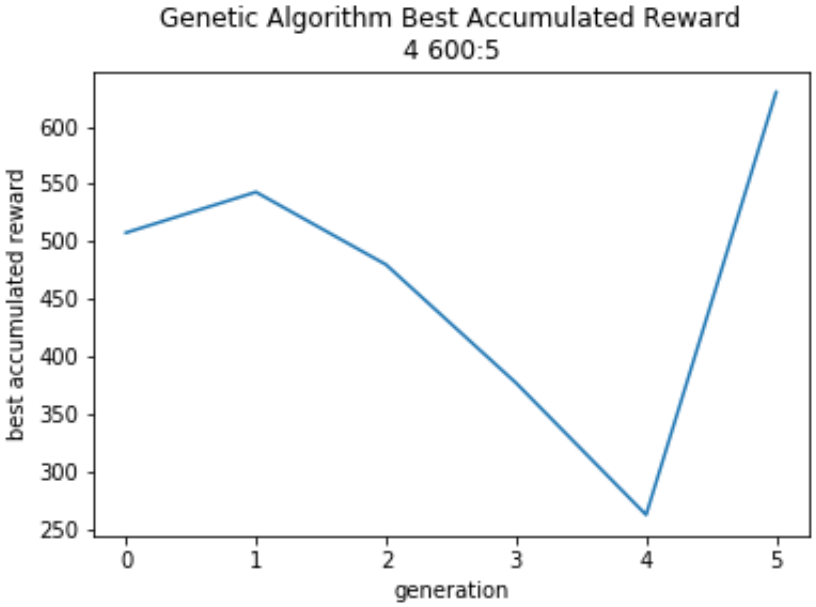


Figure C5. History of best rewards for genetic algorithm run 5

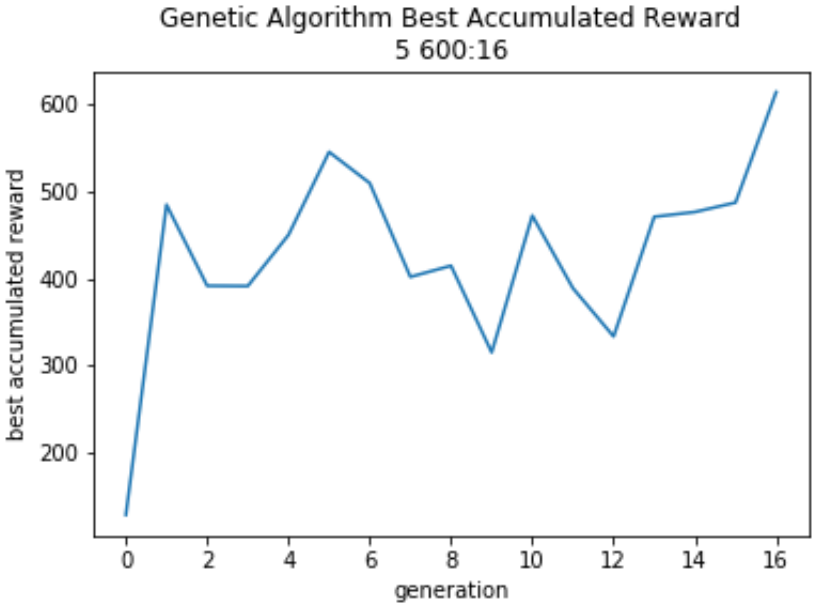


Figure C6. History of best rewards for genetic algorithm run 6

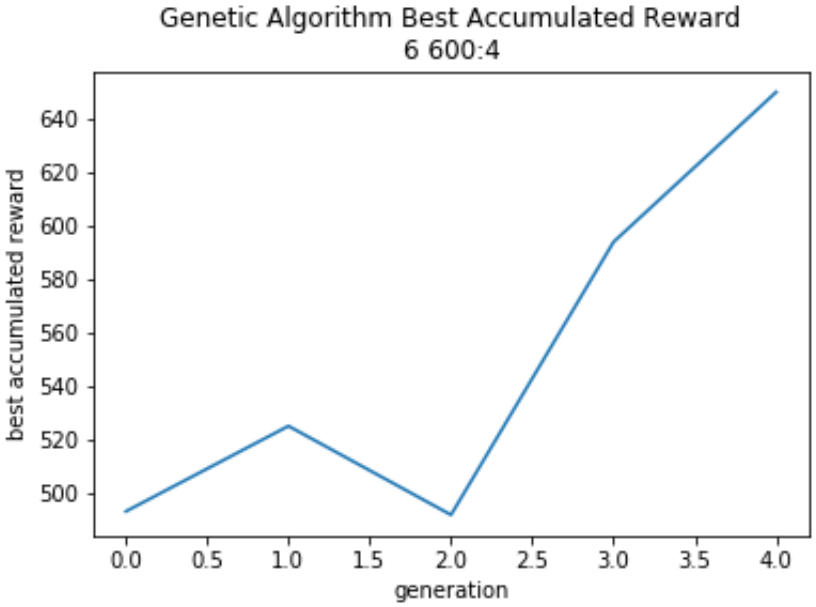


Figure C7. History of best rewards for genetic algorithm run 7

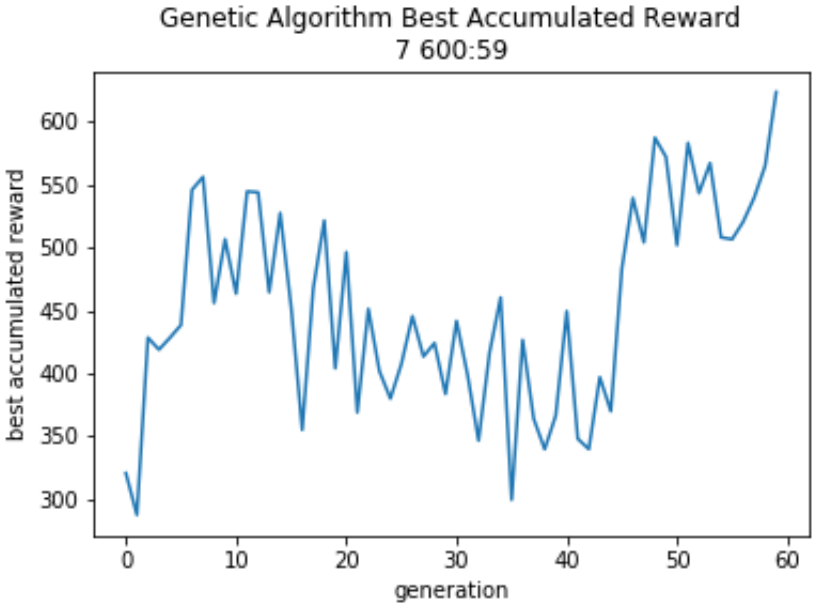


Figure C8. History of best rewards for genetic algorithm run 8

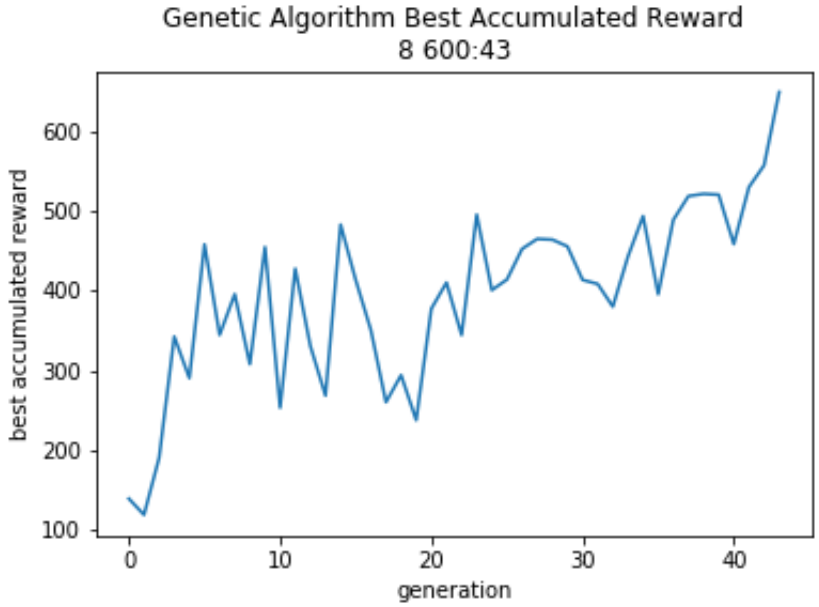


Figure C9. History of best rewards for genetic algorithm run 9

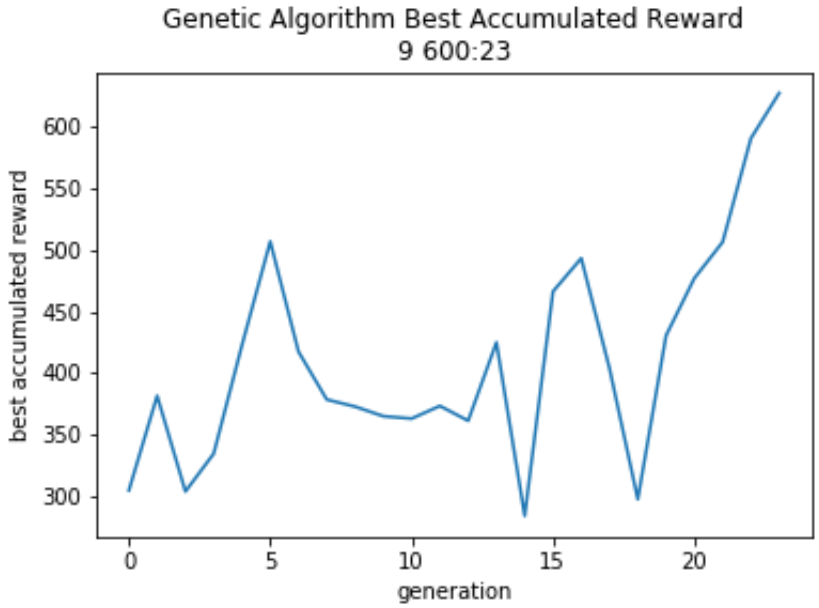


Figure C10. History of best rewards for genetic algorithm run 10

Appendix D
Specifications

Unity Version used: 2018.4.11
ml-agents version: beta, 0.12
3D-Printer Used: Qidi Tech One, model 2018
links to files: *realworldml.com*