Student: IJsbrand Daleboudt, 6212573
Supervisor: Gerard Vreeswijk **Second reader:** Dragan Doder

Reinforcement learning in self-driving cars: the difference in learning time between multi-agent Q-learning and neuroevolution.

Abstract

In this research, the learning speed of the multi-agent Q-Learning and neuroevolution algorithms is compared in a simulation of multiple self-driving vehicles. Both algorithms are reinforcement algorithms that can learn to move through a course without supervision from outside the simulation. Q-Learning is a well-known reinforcement algorithm and works on the basis of a Q-table that is updated during learning. Neuroevolution is the branch of artificial intelligence that researches the combination of neural networks with Darwin's theory of evolution. In this way, a neural network is obtained that can best adapt to its environment. The research has shown that there is a significant difference between the learning speed between the two algorithms, where neuroevolution comes to a faster resolution than Q-Learning.

Index

1. Introduction

Artificial intelligence has made a massive development in making vehicles autonomous over recent years. A lot of research is currently being done on the implementation of neural networks in cars with the aim of making them autonomous. Now the question is what the best way would be to obtain a neural network that can move a vehicle autonomously through traffic using sensors. An important requirement for this network is that this manoeuvring is done safely.

Different types of techniques

There are many different techniques that are currently used to make a car autonomous. With the help of supervised learning, the self-driving cars receive labelled data on which they can learn to recognize components of a traffic situation. These are, for example, labelled images of traffic signs so that the self-driving vehicle can recognize them and act correctly in traffic. The networks that drive autonomously can also be obtained through supervised learning. These networks can be obtained from the input of a driver of the vehicle. The disadvantage of this technique is that it takes a long time to obtain such a network and that it entails considerable costs. An alternative is to use reinforcement learning and train a network in a simulation that mimics the world. The advantage of this is that a vehicle can learn to drive a car safely, without endangering its physical environment. Another advantage is that the simulation can take place on several computers at the same time so that it can be learned more quickly. In this way, a suitable network can be found faster than if an agent learns separately.

Different type of sensors

Opinions about the required sensors are currently divided among car manufacturers and AI researchers. Waymo focuses on autonomous driving of cars by using, among other sensors, a LIDAR system. This is a system that measures the distance to objects in its environment from the car with laser sensors. Tesla, on the other hand, is currently focusing on learning to drive autonomously based on camera images, partly because this is a cheaper alternative. They are convinced that this will work because people can drive a car based on what they see. However, in this research, only a simplified version of LIDAR will be considered. The main question of this research is: what is the difference between Q-Learning and neuroevolution in a multi-agent system using a simplified radar system based on the LIDAR system?

In this study, neuroevolution is chosen because, according to the literature, this approach provides the best performance compared to other reinforcement algorithms. Another advantage that is mentioned is that this technique is widely applicable. For example, neuroevolution can be used for supervised learning, but also for unsupervised learning and reinforcement learning. Another advantage is that neuroevolution is also a very useful algorithm for large search spaces (Risi, 2015). A major drawback is that the neural network does not provide direct insight into the behaviour that has been learned. The neuroevolution algorithm is compared in this study with Q-Learning. Q-Learning is an older reinforcement learning algorithm. However, Q-Learning is still relevant today and is used in many different variants. For example, Double Deep Q-Learning is often used in combination with a convolutional neural network to learn modes of action based on camera images. There have been many successes in the applications of variants of Q-Learning in recent years. Google DeepMind successfully implemented Double Deep Q-Learning on some of the games in the Atari 2600 domain (Van Hasselt, Guez & Silver 2015) and other deep learning variants of the Q-Learning algorithm can be used to recommend news (Zheng, Zhang, Zheng, Xiang, Yuan, Xie & Li, 2018). Q-Learning is, therefore, a good algorithm to compare with neuroevolution.

In this research, the simulation, the algorithms and their implementation will be discussed, there will be explained how the simulations were performed, what the results were and an interpretation of the data will be given.

2. **Simulation**

The simulations for this research were done in the game engine Unity. This software program is a suitable program to do simulations because Unity includes a built-in physics engine that can easily move objects and detect collisions. In the simulation, 100 cars have been placed to manoeuvre through the course using both reinforcement algorithms.

Cars

The car has the shape of a square cuboid. A radar is placed on top of the vehicle that collects the data from the five input radars (see figure 1). The number of five input radars was chosen because a balance had to be made between the effectiveness of the radars and the time it takes for the

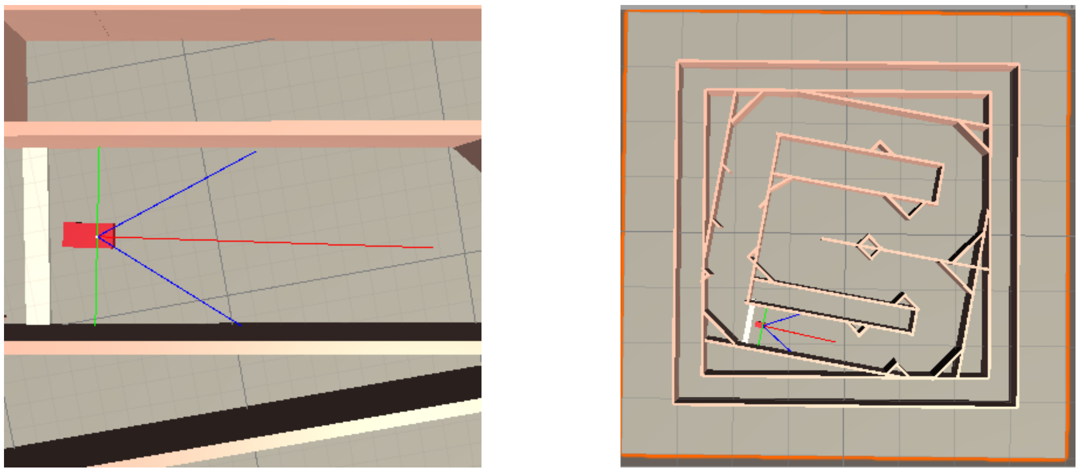


Figure 1: On the left is a top view of the car used for the simulations of the Q-Learning algorithm. This car has a discretized finite input space which gives the agent a limited view. The cars that use neuroevolution have no limited vision and can see straight ahead in all directions. The top view of the course is shown on the right.

algorithms to converge. Tests have shown that using five radars is enough to teach the car to drive through the course, but at the same time provides a working solution relatively quickly. Furthermore, the shape of the car has been left simple to put the full computer power into the simulation. The radar is based on the LIDAR system and measures the distance to the wall in five different directions, namely -90, -45, 0, 45 and 90 degrees in the direction in which the car is driving. The cars start in the first round at a speed of 0 units. When the simulation starts, the cars accelerate with 1 speed unit per frame. This is implemented because it prevents the simulation from errors. The car has a maximum speed of 10 units. Steering power is determined by multiplying the speed by 15 to ensure that the car cannot disproportionately steer at low speeds. The cars only have to steer. Changing the speed is not taken into account in this simulation. The purpose of the cars is to drive ten laps on the track without colliding. If the cars collide, it will differ per algorithm what happens to the car. If a car collides during the simulations with the Q-Learning algorithm, the car is immediately returned to its starting position and it can start a new lap. In the neuroevolution algorithm, the car first comes to a stop and can only start again when every car in the population has collided. The cars cannot see the other cars.

Walls

The walls contain a script that can detect collisions. The script of the wall can make adjustments to the scripts of the cars. In the Q-Learning algorithm, the script of the walls ensures that the car is returned to its original position after a collision. In the neuroevolution algorithm, the wall script sets the speed of the car to zero.

Course

The course contains two U-turns to the left, a U-turn to the right and two bends to the left (see figure 1). Additional objects have also been placed on the course. This is to test whether the algorithms can handle additional information on the road. It was decided not to test the algorithms on a second course, because this research focuses only on examining the differences in learning speed between the two algorithms.

3. Q-Learning

Q-Learning is a form of model-free reinforcement learning (Watkins, 1989). Q-Learning allows agents in Markov domains to learn how to interact with their environment. A Q-Learning agent determines his actions based on his current state and evaluates the action taken in terms of the direct reward or punishment he receives and the estimate of the value of the state he is moved to. By repeating this often, the agent learns an optimal method to move through the environment based on the reward received. This process is shown in Figure 3. The states and actions are stored in a Q-Table. This Q-Table is updated based on the actions that an agent takes. The updates will give the best actions the highest values and the agent will choose the corresponding action based on these values.

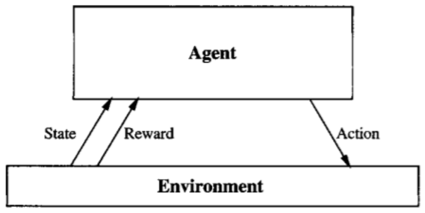


Figure 2: Model-free reinforcement learning. (Mitchell, 1997)

Markov Decision Processes

Markov Decision Processes (MDPs) represent the environment of an agent in a model (Littman, 2001). MDPs assume that the agent has full visibility of his environment. A Markov Decision Process can be represented in a tuple: $\langle S, A, T, R, \gamma \rangle$.

- S stands for the states of the agent's environment.
- A is the finite set of actions that the agent can perform.
- $T: S\ x\ A \rightarrow \Pi(S)$ is the transition function, and represents the probability that an agent will end up in the state s'.
- $R: S\ x\ A \rightarrow \mathbb{R}$ represents the reward or penalty awarded for an action.
- γ represents the discount factor between 0 and 1.

At time t, the agent chooses an action $a \in A$ based on $s \in S$ and executes it. The environment determines the quality of the action taken by the agent and rewards the agent based on this action. The agent then moves to the next state. Previous states or actions are not included in determining future actions (Mitchell, 1997). In this experiment, the states S are the discretized inputs from the radar. The states have been discretized to keep the size of the Q-Table within bounds. The states S and actions A are both discrete and finite. The policy $\pi: S \rightarrow A$ chooses from state $s \in S$ an action $a \in A$. The goal is to teach an agent behaviour that maximizes the total score V, and thus the future reward. This is summarized in the following formula:

$$V^{\pi}(s_t) \equiv r_t + \gamma^2 r_{t+1} + \cdots \equiv \sum_{i=0}^{\infty} \gamma^2 r_{t+i}$$

V represents the total score in the formula, r represents the reward that can be earned in a future move and γ represents the chosen discount factor between 0 and 1. With a discount factor close to 0, the agent will choose the actions that give the highest reward in the near future. With a discount factor of 1, the agent will choose actions that will yield a higher reward on the long term (Mitchell, 1997).

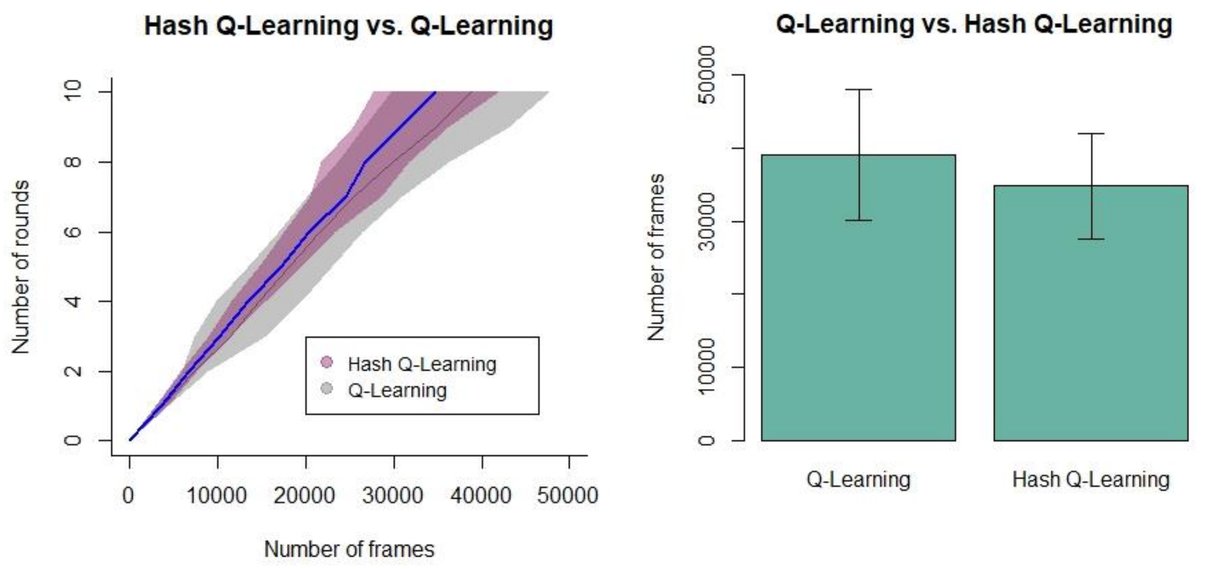


Figure 3: Q-Learning is compared to a multi-agent solution called Hash Q-Learning. The left graph shows that Hash Q-Learning comes to a faster result than the normal Q-Learning algorithm. In addition, the spread is lower with the Hash Q-Learning algorithm.

Multi-agent Q-Learning

In order to obtain a converged Q-Table faster, it is possible to use a multi-agent solution. By connecting multiple agents to the Q-Table, the learning process can be accelerated by sharing experiences. However, this creates new challenges, namely: the difficulty of specifying a common learning goal, the non-stationarity of the learning problem and the need for coordination (Buşoniu, Babuška & De Schutter, 2010). Non-stationarity is a problem in multi-agent reinforcement learning because the agents are connected to the same table and this can cause the same values in the table to be updated several times in the same time span (Buşoniu et al., 2010). This can cause a Q-Table to take longer to converge using multiple agents. Nevertheless, there are practical examples where Q-Learning is directly applied to multi-agent learning. In these examples, the agents learn what is the best action using the Q-learning update rule, without being aware of the other agents. However, the non-stationarity of the MARL problem invalidates most single-agent reinforcement learning theories (Buşoniu et al., 2010). Therefore, in this research a proposal is made to improve the Q-Learning algorithm in a multi-agent setting. This solution addresses the non-stationarity problem of the Q-Learning algorithm when used in a multi-agent system.

The proposed solution takes the frequency with which a place is visited in one update into account, and will not be given more than one update per frame per state. A hash set keeps track of which states have already been visited in the update round. This results in values in the Q-Table not being disproportionately updated when multiple agents are in the same place at the same time. This small adjustment yields a significant difference, based on 50 simulation rounds for both algorithms (see figure 3). The results showed that there is a significant difference between the Hash Q-Learning algorithm (M = 39067.73; SD = 9136.449) and the Hash Q-learning algorithm (M = 34849.80; SD = 7092.614) in the same simulation; t(90.44) = 2.5527, p = 0.01236. The customization is called Hash Q-Learning because of the use of a hash set.

Hash Q-Learning algorithm implementation

To implement Hash Q-Learning, a Q-table is initialized for each discretized combination of the input sensors and the number of actions an agent can perform. Each combination in the Q-table receives two values: the Q value and the frequency with which this combination was visited. Both values are set to zero. The agent is then placed on the track and considers the state he is in. Then an infinite loop is started that updates the Hash Q-Learning table. First, an action $a \in A$ is selected and performed based on the input sensors. Based on the action, the agent receives a reward or a penalty and considers the new state. Based on the action taken and the agent's reward or punishment, and whether the agent's state is in the hash set or not, the table is updated. The old state's Q value is updated using the following formula (Mitchell, 1997):

$$Q(s,a) \leftarrow (1 - a_n)Q_{n-1}(s,a) + \alpha_n[r + \gamma \max_{a'} Q_{n-1}(s',a')]$$

Due to the use of multiple agents on the same Q-Table, a learning rate is used based on the number of times an agent has visited the state. Using this formula ensures that the table is not unilaterally updated. The learning rate α_n is calculated using the following formula (Mitchell, 1997):

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

The formula is built into the simulation in the following way:

```
**Algorithm** Hash Q-Learning
1:  Initialize Q-Table
2:  resetPosition = position
3:  collided = false,
4:  newCar = true
5:  epsilon = 0.2
5:  while number of rounds is not 10:
6:      if newCar is false & state is not in Hash set
7:          reward = 0.1
8:          get state(s',a')
9:          Append state to Hash set
10:         $Q(s,a) <- (1-a_n)Q_{n-1}(s,a)+a_n[reward+y maxQ_{n-1}(s',a')]$
11:     find max Q(s,a)
12:     if random (0, 1) < epsilon
13:         a = random
14:     Execute action
15:     newCar = false
16:     if collided == true
17:         reward = -10
18:         $Q(s,a) <- (1-a_n)Q_{n-1}(s,a)+a_n[reward+y maxQ_{n-1}(s',a')]$
19:         position = resetPosition
20:         collided == false
21:         newCar = true
22:         epsilon =* 0.9998
```

When the car hits a wall, the Boolean that keeps track of whether the car has crashed will become true. A new Q-value will be calculated with the reward awarded when a vehicle is impacted. The reward awarded on impact is -10. In all other situations, the agent receives a reward of 0.1. If the car has crashed and the new Q-value has been calculated, the vehicle is returned to the starting position and the Boolean that indicated the collision is set to false. Another script ensures that the hash table is emptied after every round of updates. That script also contains the epsilon that allows the agents to take a random action. The epsilon ensures that the agents not only exploit but also explore. The epsilon represents the probability between 0 and 1 that a random action will be performed. Epsilon is set to 0.2 at the beginning of each simulation. With every car that collides, the value is gradually reduced so that the balance slowly shifts from exploration to exploitation.

In this experiment, a discount value of 0.99 is used. The Q-table is a 6-dimensional array. The indexes of the table represent the 5 discretized inputs of the sensors and the number of actions the vehicle can perform. The array for the radar inputs on the side of the car is 5 buckets in size. The array for the sensors that measure the distance at 45 degrees is 8 buckets in size. The middle radar has a bucket size of 15 buckets. Furthermore, there are three different actions that a vehicle can make: steer to the left, no steering or steer to the right. Due to the exponential growth resulting from adding additional values to the Q-table, it was decided not to implement a varying speed in this study. The table of actions would then be at least three times larger to also add brakes, release the throttle and accelerate. This would also mean that another table had to be created where the current speed is tracked. The following formula can be used to calculate the table size of a Q-table:

$$table\ size\ = \prod_{i=1}^{dimension\ of\ Q-Table} array\ size_i$$

That means the table used in this study has a table size of 72,000 values.

$$72000 = 5\ \cdot\ 8\ \cdot\ 15\ \cdot\ 8\ \cdot\ 5\ \cdot\ 3$$

Adding the extra actions would add another seventh bucket with three values in it for the agent to adjust the speed. This would bring the total number of values in the table to 216.000. If the agent also had to distinguish between all the different speeds, there would be an eighth bucket with ten values. The number of possible states of the agent would then stand at 2.160.000 values. This would take too long to train for the computing power available.

Discretizing the input

The Hash Q-Learning algorithm is implemented using buckets for the radar data, to reduce the number of possible states. The data from the radar input therefore has to be discretized. The bucket size is 0.9. Runs have determined that 0.9 buckets provide enough accuracy, but also provide sufficient visibility for the agent. The array sizes indicate the maximum size that the buckets can take, which is the number of times the size of the array in the Q-table is divisible by the size of the bucket.

4. Neuroevolution

The field of neuroevolution focuses on acquiring neural networks using evolution (Stanley, Clune, Lehman & Miikkulainen, 2019). Neuroevolution does not focus on incrementally improving weights in an artificial neural network, but is inspired by Darwinian evolution. The neural network that adapts best to the environment will survive and will have the opportunity to reproduce. Due to the increasing accessibility to computer power, neuroevolution is becoming increasingly popular. In fact, neuroevolution is a competitive alternative to other reinforcement algorithms that are often used today, such as Deep-Learning. Neuro-evolution is currently already being used to generate music, control robots or model natural phenomena. With neuroevolution, the algorithm can focus on finding both a good structure of the network and the weights that act on that neural network (Risi & Togelius, 2015). In this study, however, a fixed structure of the network is chosen and the algorithm will only the search for the right weights. Figure 4 shows the structure of the chosen network.

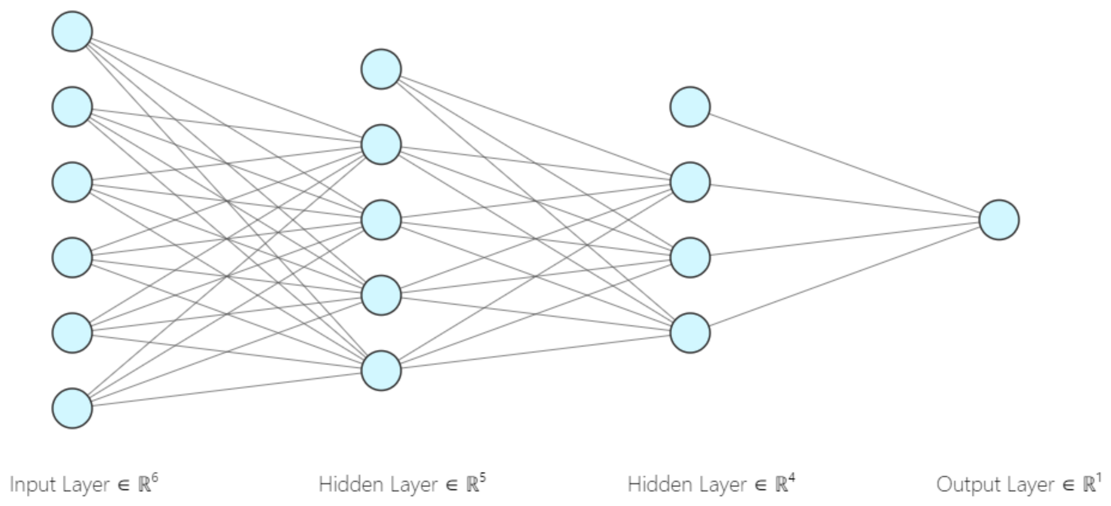


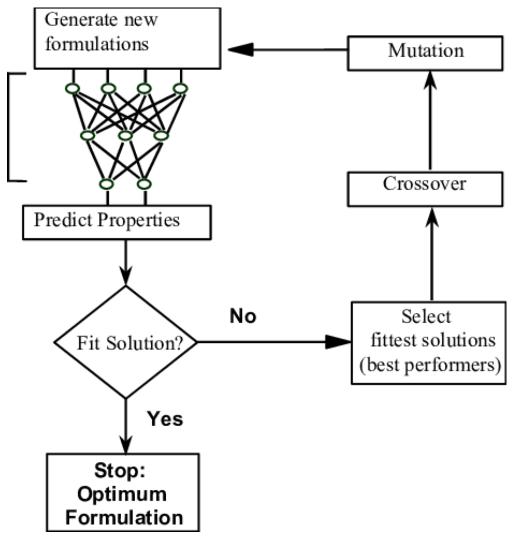
Figure 4: The structure of the neural network used in this research.

Structure

For the neural network, a feedforward network is chosen with 5 input layers and a bias, a hidden layer with 4 nodes and a bias, then a hidden layer with 3 nodes and a bias and an output layer of 1 node. The input layers receive their input from the input radars. This will propagate forward with the feedforward algorithm and give an output in the output layer. The hyperbolic tangent was chosen as the activation function. This activation function returns a number between -1 and 1. This can be used well in the output to estimate the control the vehicle must make. In this case, -1 is to the left, 0 is in the middle and 1 is to the right.

The neuroevolution algorithm

The neuroevolution algorithm is based on Darwin's theory of evolution (see Figure 5). Each individual in the population is assigned a neural network with random weights between minus one and plus one. A bias is added to the input layer and both the hidden layers. The value of the bias is a rational number between the minus two and the plus two and differs per layer. The individuals then move based on the output determined by a combination of the radar input and the calculation of the network by the environment. A fitness will be determined on the basis of these actions. In this experiment, fitness is the distance an agent travels. High fitness individuals are more likely to reproduce than low fitness individuals. After each round, the fitness of each individual is evaluated. The best ten individuals are allowed to reproduce. The individuals who do not belong to the top ten cars will get a new network that will be passed on by parents who are among the best ten individuals. Crossover of weights of the neural network occurs with a risk of mutation. The chance to inherit a chromosome from the father or mother is ten out of twenty-one. The chance of mutation is one in twenty-one. This process will be repeated until a network has been found that meets the conditions for driving 10 laps without colliding.



Generate new formulations
Mutation
Crossover
Predict Properties
Fit Solution?
No
Select fittest solutions (best performers)
Yes
Stop: Optimum Formulation

Figure 5: The selection process based on Darwin's theory of evolution. (Rowe & Colbour,2003)

5. Methods

A quantitative study was chosen to determine which algorithm converges faster. For Hash Q-Learning and neuroevolution, it was decided to drive a hundred cars through the maze at the same time. Both algorithms had to determine the best steering route. The simulation was repeated 50 times for the Hash Q-Learning algorithm and 50 times for the neuroevolution algorithm.

Hash Q-Learning

For the Hash Q-Learning algorithm, all 100 different cars have been placed on the track and a Q-Table has been initialized before the start of the simulation, with each value set to 0. The cars had to drive through the track until they collided. When they collided, the cars were returned to where they started. In addition, their reward was reset to 0. The decision was made to have the cars spawn immediately at their starting point if they crashed. This is to prevent the table from converging unilaterally.

Neuroevolution

When investigating the neuroevolution algorithm, it was decided to use populations. With this algorithm, all cars were first placed on the board and assigned a random network. Hereafter, the first round could be started. After this round, the best 20 were allowed to reproduce with each other and their descendants replaced the worst 80 agents.

Converging

For both algorithms, the simulations continued until the algorithms had a car that drove ten laps. Also, for both algorithms, it was recorded how many frames it took from starting the simulation to learning to drive each additional new lap. These values were stored. An independent t-test was then carried out on this value.

5. Results

An independent t-test was performed to measure the difference in learning speed between the Hash Q-learning algorithm and the neuroevolution algorithm. The results show that there is a significant difference between the neuroevolution algorithm (M = 17328.08; SD = 10707.5) and the Hash Q-learning algorithm (M = 34849.80; SD = 7092.614) in a 3D simulation of self-driving vehicles; $t(85.254) = 9.6168$, $p < 0.001$. This means that the neuroevolution algorithm in this simulation learns significantly faster than the Hash Q-learning algorithm.

The results fit in well with the theory that was consulted. Many researchers state in the literature that neuroevolution is a good opponent for contemporary reinforcement algorithms. However, Q-Learning is a bit older and since Q-Learning has been developed, in addition to adjustments to existing reinforcement algorithms, many new reinforcement algorithms have been added. With reinforcement algorithms that use Markov Decision Processes, the problem is that the search space increases exponentially when additional input sensors are added. This simulation required a 6-dimensional table, even if the car did not have to accelerate itself. With this accumulating exponential increase in the search space, neuroevolution can be expected to win over Q-Learning. Neuroevolution learns almost twice as fast. The results showed that neuroevolution requires an average of 17328.08 frames to complete ten laps. Hash Q-Learning, on the other hand, needs more than double the number of frames with an amount of 34849.80 frames.

Figure 6 shows that the Hash Q-Learning algorithm learns almost linearly with a slightly decreasing upward slope. During the execution of the simulations, it was clear that the Q-Table needed time

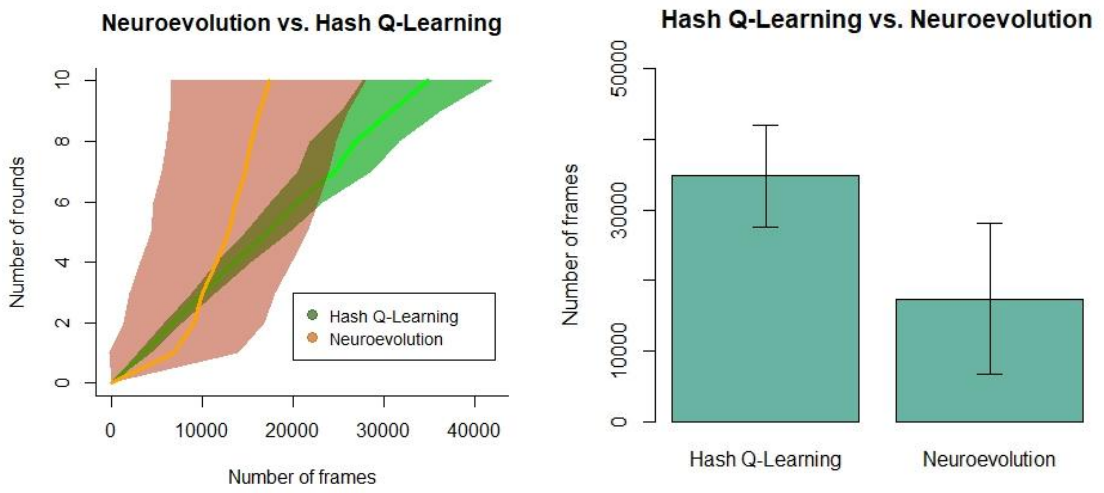


Figure 6: Neuroevolution is compared to Hash Q-Learning. There is a statistical difference in learning speed.

to adapt per corner of the track. Learning was gradual, step by step. This was not the case with the neuroevolution algorithm. In neuroevolution, the cars initially gradually progressed a little further, but compared to the cars that used Hash Q-Learning, this was slower in the earlier stages than with Hash Q-Learning. This can also be seen in figure 6 between 0 and 10.000 frames. This is due to the fact that the cars of the neuroevolution algorithm were brought to a halt when they collided with one of the walls, and could not drive again until they were spawned into the new generation. After an average of 10.000 frames, the cars continued to improve through inheritance and mutation. In this simulation, it quickly became clear which neural networks could adapt best. Neural networks that manage to do a full lap usually also manage to do two laps, and neural networks that can do two laps usually manage to do three laps as well, and so on. This is also reflected in the data. In many cases, the cars were able to drive straight to the end after one successful lap. This also explains the steep learning curve upwards from around the 10.000 frames. Once the agent with a well-adapted neural network is up and running, chances are it will continue.

6. Conclusion and discussion

This research has shown that there is a statistical difference between the learning speed of a neuroevolution algorithm and a multi-agent Q-Learning algorithm in a 3D simulation of self-driving vehicles. Those simulations where neuroevolution has been used come to a good result faster than simulations where Hash Q-Learning has been used. However, Q-learning is a relatively old reinforcement algorithm. Q-Learning in this form is not suitable for learning to drive a car. It takes a long time for the simulations to provide a satisfactory solution, and the algorithm also cannot safely handle situations that the algorithm has never seen. After all, the action values on which the agent bases his actions are set to 0 throughout the table upon initialization. When the car is in an as yet unknown situation, a random action will be chosen and this may cause problems. However, neuroevolution comes to a result very quickly and does not have the problem of not being able to deal with situations it has never encountered, because he can derive favourable behaviour from similar situations because of his neural network. However, the problem remains that the network found does not provide direct insight into the learned behaviour.

For the conclusion of this study, a number of points that have not been included in this study should be taken into account. The autonomous vehicles in the study only had to steer. It has not been investigated how the agents learn when they can also influence their own speed. Furthermore, the simulations in this study were conducted in a static environment where there are no moving objects other than the agents, and no interaction between the agents. The simulation also had no traffic rules, while traffic rules do apply in real life. Furthermore, this research only investigated a limited search space by testing on only one course. However, this research does provide insight into the difference between the learning speed of the Hash Q-Learning algorithm and a neuroevolution algorithm. However, for further follow-up, research it would be interesting to look at the difference in learning speed between more modern reinforcement algorithms and neuroevolution. In such a follow-up study, it is interesting to also include several factors to be able to paint a more realistic picture of the time an algorithm needs to learn such a complex task. In addition, it is also interesting to see which algorithm can guarantee the greatest security, and it is interesting to compare the effectiveness of LIDAR techniques with the effectiveness that cameras provide.

References:

Buşoniu, L., Babuška, R., & De Schutter, B. (2010). Multi-agent Reinforcement Learning: An Overview. *Innovations in Multi-Agent Systems and Applications - 1*, 183–221. https://doi.org/10.1007/978-3-642-14435-6_7

Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, *1*(1), 47–62. https://doi.org/10.1007/s12065-007-0002-4

Littman, M. L. (2001). Value-function reinforcement learning in Markov games. *Cognitive Systems Research*, *2*(1), 55–66. https://doi.org/10.1016/s1389-0417(01)00015-8

Mitchell, T. M. (1997). *Machine Learning*. New York, United States: McGraw-Hill Education.

Rowe, Raymond & Colbourn, Elizabeth. (2003). Neural computing in product formulation. 8. 1-81.

Risi, S., & Togelius, J. (2017). Neuroevolution in Games: State of the Art and Open Challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, *9*(1), 25–41. https://doi.org/10.1109/tciaig.2015.2494596

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*(1), 24–35. https://doi.org/10.1038/s42256-018-0006-z

Van Hasselt, Hado & Guez, Arthur & Silver, David. (2015). Deep Reinforcement Learning with Double Q-learning.

Watkins, C.J.C.H. (1989). Learning from delayed rewards. PhD Thesis, University of Cambridge, England.

Zheng, G., Zhang, F., Zheng, Z., Xiang, Y., Yuan, N. J., Xie, X., & Li, Z. (2018). DRN. Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18. https://doi.org/10.1145/3178876.3185994