sciendo

VOLUME 17, N° 2 2023
Journal of Automation, Mobile Robotics and Intelligent Systems

A Compact DQN Model for Mobile Agents with Collision Avoidance

Submitted: 11th April 2023; accepted: 31st May 2023

Mariusz Kamola

DOI: 10.14313/JAMRIS/2-2023/13

Abstract:
This paper presents a complete simulation and reinforcement learning solution to train mobile agents' strategy of route tracking and avoiding mutual collisions. The aim was to achieve such functionality with limited resources, w.r.t. model input and model size itself. The designed models prove to keep agents safely on the track. Collision avoidance agent's skills developed in the course of model training are primitive but rational. Small size of the model allows fast training with limited computational resources.

Keywords: *Q-learning, DQN, Reinforcement learning.*

1. Introduction

In order to accomplish a task, an unmanned agent must operate with adequate situational awareness and execute an adequate decision support algorithm. The complexity of both of the above components should be matched – be it a case of an autonomous car or a mere line-following toy robot.

The aim of work presented in this paper was to develop rich representation of road topology, yet create a representation suitable for processing by a relatively simple decision model of an autonomous and mobile agent. The agent's main goal is to reach its destination, by riding on a road and taking turns, simultaneously observing other moving agents in order to avoid collisions.

The range of stimuli processed by the autonomous vehicle is determined by their physical observability, the cost and power consumption of measurement and communication equipment, and the cost and power consumption of hardware the decision model is running on. Autonomous cars are nowadays by far the most sophisticated civil agents, equipped with LiDAR, radar, cameras, and GPS as well as a huge number of sensors collecting the state of the car itself. These real-time data must be completed with accurate, up-to-date and rich maps of the neighborhood in order to navigate efficiently, which poses problems either with storage or bandwidth demand, depending on where the maps actually reside.

In order to consume such rich input data streams in timely fashion, adequate processing power is needed. Tesla's Autopilot processor, NVIDIA DRIVE PX2, consumes 250 watts, which is much more than the power of the car headlights. This figure does not include powering all remaining sensors and communication devices. Moreover, as for energy used for hardware manufacturing and operation, contemporary decision models carry substantial training carbon footprint. For example, the training of a natural language modern neural model consumes over 650 MWh of direct and associated energy (mainly cooling) [1].

The case of an autonomous car can be considered an extreme one – yet striving to achieve planned goals but with smaller resources is more and more pronounced. Various neural model reduction techniques have been proposed [2], and the hardware itself becomes more energy efficient, including the autonomous car computers as well. But still it is a common approach to throw all training data into a complex neural network model, use a great deal of resources to train it, and perform model compression as the last stage before the deployment.

Here we propose an environment and decision model for an autonomous vehicle that uses less resources, due to careful encoding of the agent's input. With such lean processing infrastructure, model training is simple and ready for educational purposes as well as for further development for industry use.

1.1. Related Work

Our work contributes to the wide and active area of research for autonomous driving, whose main and most challenging topic is control algorithms for self-driving cars. These can be developed in two contrary methodologies: the modular one, encompassing perception, planning, and control blocks, or the end-to-end one, mingling the above functionalities into a single decision model. The state of the technology achieved through the two approaches is presented in detail in Grigorescu et al. [3], and we will point out key advances therein which are relevant to our research.

Modules for perception, high- and low-level planning, and motion control can be accomplished diversely in each class, adequately to available resources (hardware, data, power, money). For example, there are efforts to replace state-of-the-art LiDAR sensors with cheaper 2D stereoscopic cameras and 3D reconstruction models (PointNet, AVOD), trained on real LiDAR measurements. Detected spatial objects can be labelled semantically afterwards with a bunch of models: SegNet, IC-Net, ENet, and the like, derived from general computer vision architectures such as AlexNet, ResNet, and more.

2023 © Kamola. This is an open access article licensed under the Creative Commons Attribution-Attribution 4.0 International (CC BY 4.0)
(http://creativecommons.org/licenses/by-nc-nd/4.0/)

Planning involves a range of tasks, starting from high-level path planning and going down to low-level behavior arbitration (lane control, collision avoidance, etc.) While path planning usually employs a sort of white-box model working on map data and a local road occupancy grid, behavior arbitration models are neural ones and come in two flavors. Deep reinforcement learning is built on simulated environment, developing an optimal policy based on the reward function – yet the policy contains non-learnable parts that are essential for maintaining vehicle safety. Interestingly, the opposite approach – imitation learning – aims actually to learn correct reward function from real scenarios by human drivers. Both approaches have drawbacks, suffering from data quality: simulation output inadequate to reality or scarcity of corner cases, respectively.

Motion control is usually effected by state-of-the-art model predictive control (MPC) which, after having learned vehicle dynamics properties, takes advantage of optimal control theory in order to provide control signals on a considerable control horizon. Applied iteratively, it can adapt to disturbances and modify control trajectory computed so far.

End-to-end methodology transfers knowledge from existing models, which makes it possible to consume raw and granular data, such as object locations, including the agent itself, and transform it by a multilayer network into decisions believed to be optimal. PilotNet by Nvidia and AutoPilot by Tesla are examples of such complex, monolithic models.

Deep Q-learning (DQN) is now the key planning algorithm, and also central to our interest here. It is presented in detail in Section 2.2. A neural model is taught to estimate the action-value function Q of total rewards (immediate and discounted future ones) for a control decision, given the current system state. The framework has been enriched so far by a number of extensions, for instance [4]:

- Double Q-learning: two network models are trained using different batch data, in order to reduce bias in Q estimation;
- Prioritized Replay: samples for training are drawn with probability relative to their temporal difference errors, for example, errors of predicted Q values, in order to improve the model where it performed worst;
- Dueling Networks: a model of Q gets accompanied with a model of V, a value of state regardless of control taken there; both models are merged in order to focus learning on states where the control is really crucial;
- Multi-step Learning: control action takes place only every n-th step, letting the object evolve; it leads to faster learning in specific tasks;
- Distributional RL: reward value is modeled as distribution over predefined discrete space, allowing for more insight and resulting in faster training; the distribution gets updated by small corrections (*Categorical DQN*) or gets incorporated into the main model (*Quantile Regression DQN*; see [5]);
- Noisy Nets: a noisy input is added to the network model, allowing agents to learn states where the noise is relevant (high weights) for Q value.

DQN models are often successfully extended to provide end-to-end driving functionality, as in [6], where raw camera images as well as vehicle speed and orientation w.r.t. road waypoints are the network inputs. To account for vehicle dynamics, the latter inputs go through recurrent layers (LSTM, long short-term memory), while visual ones go through typical convolution layers. The authors report good results, also if the visual part is trained from scratch, without transfer learning of any sort.

2. The Solution

Let us assume that the goal is to provide an autonomous vehicle with situational data and equip it with a decision model so that it would be able to navigate a road system to reach a destination, while avoiding collisions with other agents. The basic task is therefore staying on the road and complying with traffic rules. The secondary task is to interact with other vehicles to avoid collisions. Below, we present how topology and traffic rules are prepared for the agent and describe the decision model used.

2.1. Road Topology Representation

In wide range of transport environments, there are two components present: the physical infrastructure and traffic rules imposed in it. We propose to merge them into a single map, where RGB colors encode both components. An example result of such process is provided in Figure 1. Channels in red and blue (in range of luminance 0 to 255) are used to encode recommended speed in east-west and north-south directions, respectively. Specifically, to make the map more human-friendly in appearance, we use only a part of this range. Thus, the horizontal component can vary from 150, which means "fast westward" to 250, meaning "fast eastward". Luminance value of 200 is therefore the new zero in such a coordinate system. Movement gradients encoded on arbitrary R, G, or B channels are widely used elsewhere – such as in computer graphics, to visualize so-called graphical flows [7].

The green channel, apparently superfluous, has been used to encode indulgence in reward for driving according to the recommended speed and direction. This concept is strictly related to the class of decision model discussed later, but it also makes practical sense in the phase of traffic rules encoding. Consider the intersection that is located centrally in Figure 1. Without such a recommendation-cancelling signal it would be impossible to encode driving directions for the four possible maneuvers there: two possible turns (north-to-west and west-to-north) as well as for just riding straight. Therefore, setting G luminance to zero there means no punishment, whatever particular recommendations encoded in channels R and B are.

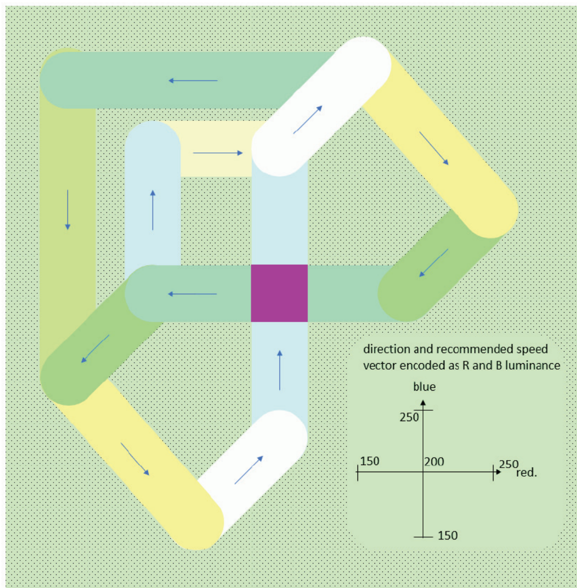

Figure 1. Example of road system with color-coded driving directions

And on the opposite, the value of 255 denotes that fine or reward is applied in full.

Considering the above rules, the colors of the plane in Figure 1 encode driving recommendations shown with arrows. In particular, the purple intersection RGB color is (200,0,200), that is, allowing unpunished driving in any direction. The recommended speed vector has been set to zero there for clarity and fairness w.r.t. the four maneuvers. The off-road parts color has been set to (200,255,200), which in essence results in maximum punishment, regardless of the driving direction. In Figure 1 they have been given a dotted pattern for better figure readability.

The proposed coloring approach carries quite a lot of information in just three channels. A part of the map in an agent's neighborhood can be fed into the model either physically (if a camera-equipped agent moves on such colored plane in a lab), or virtually (by retrieving it in real time from a database). It can constitute an extremely lean alternative to bulky, 3D urban maps, yet it can be created from information stored there. The sizes of current 3D maps for self-driving cars are measured in terabytes: it is 4 TB in case of San Francisco, many orders of magnitude bigger than in our approach [8].

2.2. Decision Model

The decision model class used was a DQN, being a variant of the Q-learning approach used widely when the state of an agent cannot be easily discretized [9]. DQN follows the original paradigm in the sense that, in a given state $\mathbf{x}$ of the agent, it provides the best known control a – w.r.t. the immediate as well as the following steps by the agent. The difference is that while classical Q-learning retrieves the value from a control-by-state table, DQN actually calculates the discrete control on the spot. This brings considerable implications to the learning procedure, described later.

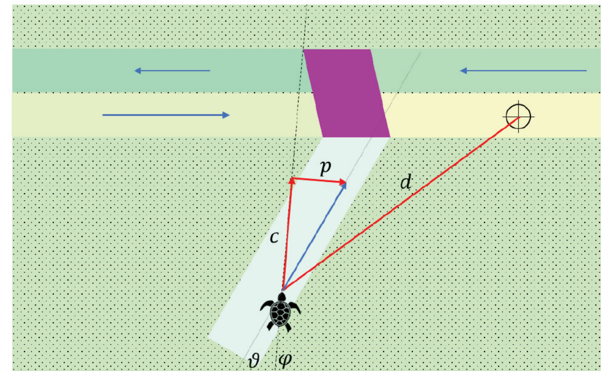

Figure 2. State components at agent location

We calculate the state information fed into DQN from the actual situation on the road and from the last step, so that it covers the following aspects:

1) agent's current driving direction vs. recommendations on the map,
2) indulgence to driving direction recommendations (channel G),
3) distance to the current goal,
4) presence of other agents or other moving obstacles.

Figure 2 presents how the above state components get calculated at the current agent location. The agent is marked with a turtle icon because the graphics comes from Turtlesim simulator [10] used further for experimentation. Component 1 is denoted by two values, colinear $c = v \cos \varphi - \vartheta$ and perpendicular $p = v \sin \varphi - \vartheta$, where v is the driving recommendation speed (blue arrow) and $\varphi - \vartheta$ is the angle between agent's actual and recommended driving azimuths, respectively. Together, c and p describe conformity of agent's movement to the recommendations. Component 2, denoted g, is just the value of G channel of the map; component 3, denoted d, is the Euclidean distance to the current goal (an arbitrary location on the map, which can be via a point of a longer route). Components with signs are presented with arrows in Figure 2, although they are in fact scalar values.

In reality, an agent, be it a robot, a driver, or a car, is able to collect the above data not only at its location point but also in the neighborhood – particularly in front of itself. To represent such situational awareness with the means provided so far, we decided to express components 1–4 not exactly at the current agent location, where some of them would be useless, but on a grid in front of it. Consequently, all components become matrices of size $N \times N$ elements, corresponding to cells of the grid, as shown in Figure 3. Therefore, component 1 values c and p become actually matrices $\mathbf{C}$ and $\mathbf{P}$, calculated for average plane colors in cells. Coarse grid is used here analogously to pixelized vision organs of insects, which apparently do not hinder them from performing high-precision tasks, such as dragonfly interception skills [11]. Component 2 is calculated with the same averaging process, resulting in matrix $\mathbf{G}$.

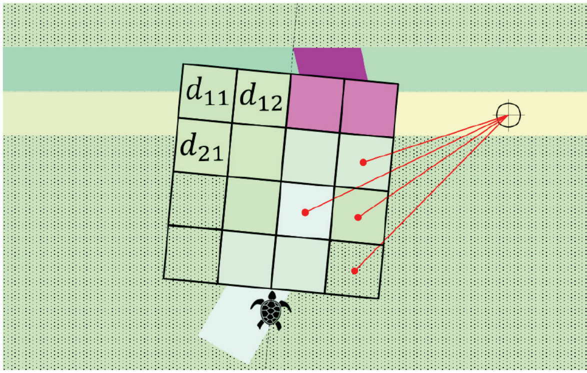

Figure 3. State components for a location grid

Component 4 distances are calculated for grid cells' centers, resulting in matrix $\mathbf{D}$. Component 4 is introduced here to complete situational awareness, as binary matrix $\mathbf{B}$, whose elements are set to 1 for cells that contain another agent.

The above matrices comprise the current situation, $\mathbf{S} = (\mathbf{C}, \mathbf{P}, \mathbf{G}, \mathbf{D}, \mathbf{B})$. Note that $\mathbf{S}$ is constructed so that it presents the neighborhood relative to current agent's location and position, rather than revealing the location coordinates themselves. This forces the agent's decision model to depend only on local situation and imposes the desired degree of generalization by design. The agent cannot learn the plane "by heart," remembering specific successful trajectories at certain locations. Rather than that, it learns desired behaviors for certain local situations, such as taking a turn on an intersection or passing a turning curve. Learning that behavior once, for one particular intersection or curve, will make it able to cope with any number of similar objects, wherever they are on its route.

DQN uses a neural network in order to evaluate Q value, that is, the quality of each of the possible actions, for a given state $\mathbf{x}$ of the agent. Then, in the greedy approach which we use, the action that gives best Q value is taken:

$$u = \max_{a} Q(\mathbf{x}, a). \tag{1}$$

The neural network behind Q is simultaneously used to direct an agent, and is trained with a history of recorded agent steps, so that it estimates as well as possible the intermediate consequences of an action:

$$Q(\mathbf{x}, a) = r(\mathbf{x}, a) + \gamma \max_{a'} Q(\mathbf{x}', a'), \tag{2}$$

where $r(\mathbf{x}, a)$ is the reward for the current step being taken, which leads to the next state $\mathbf{x}'$. The joint quality of the following actions is again estimated by the model, and the best action is assumed to be taken, as in (1) – but with the discount coefficient γ.

In order to make the reinforcement learning defined by (1) and (2) perform correctly, one has to take care with implementation details. We based our implementation on the framework reported in [12], which allows control of every implementation detail.

We found the approach of much educational value, and offering precise control on implementation details – and finally we preferred it over the popular OpenAI Gym framework [13]. One of the important implementation details is that two twin networks are used, the main network Q being subject to proper training, and the target network, denoted $\overline{Q}$, whose weights get updated from Q periodically in the process of learning. Consequently, $\overline{Q}$ evolves slowly and is used to predict quality of future actions. Therefore, (2) becomes

$$Q(\mathbf{x}, a) = r(\mathbf{x}, a) + \gamma \max_{a'} \overline{Q}(\mathbf{x}', a'). \tag{3}$$

The next detail lies in model input structure, that is, the agent state at step n. In our approach, such a state is composed not only of the current situation, but also the situation in the previous step, $\mathbf{x}(n) = (\mathbf{S}(n), \mathbf{S}(n-1))$. While such model input may seem superfluous, it describes well the first-order system dynamics. It also neatly conforms to model input structure, which must be a tensor. Describing dynamic phenomena implicitly in neural network models is quite an established practice. Confront, for instance, two action-recognition models, kinetics-i3d [7] and TSM [14]. While object movement directions in input video are calculated separately for the earlier one, and provided explicitly on the input, the latter model infers them by time-domain convolutions – which is exactly what we opt for.

The reward $r(\mathbf{x}, a)$ and the next state $\mathbf{x}'$ as the effect of action a taken in state $\mathbf{x}$, is provided in Q-learning by the environment. The environment can be the real word but, for the sake of model training efficiency, the most practical approach is to utilize some sort of a simulator or emulator. We used Turtlesim [10], with custom extensions made in order to calculate our situation components. The middleware between DQN routine and the simulator was an Environment class, implemented in Python. The class takes care of proper initialization and actual control of multiple agents, as well as of composing the situation $\mathbf{S}$. The reward r gets calculated as a sum of the following components:

- ϱ_f reward for driving according to recommendations
- ζ_r penalty for driving reverse to recommendations
- ζ_v penalty for speeding
- ϱ_t reward for approaching the current target (measured by velocity in straight line to target)
- ζ_f terminal penalty for falling off the track or crashing with another agent.

All parameters but the last one are based on continuous measurements of agent position or movement on the plane.

DQN training procedure is, in essence, a standard one. An agent is let to play an episode, that is, perform a number of steps, being controlled by the current model $\overline{Q}$, while its steps get recorded as training samples. Once there are enough samples memorized, one epoch of Q model training is performed every couple of episodes. After a number of such training sessions,

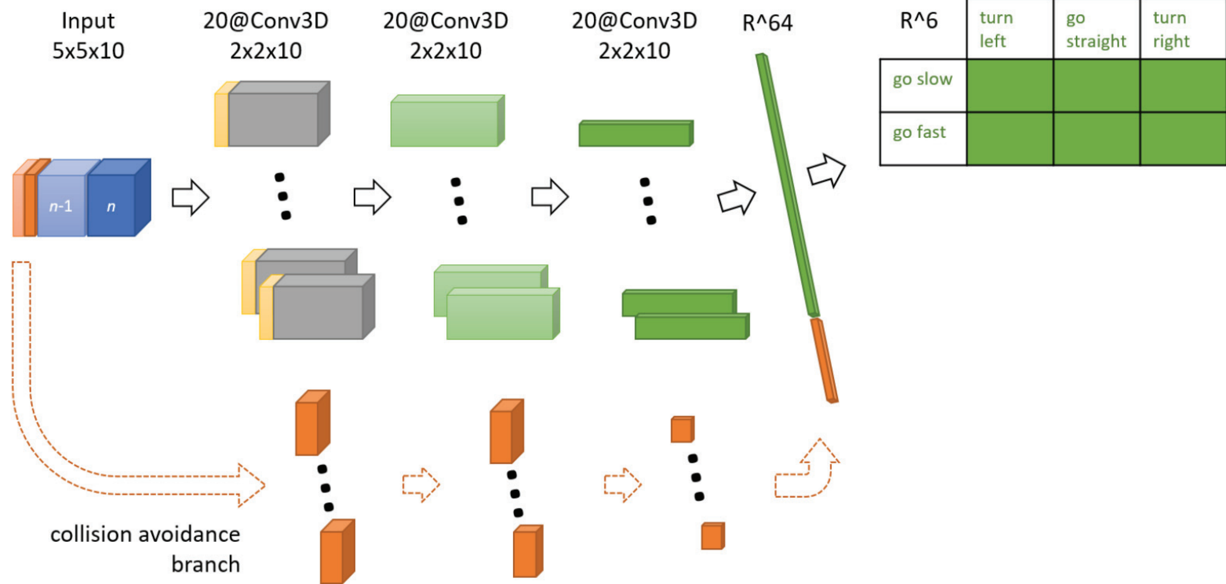

Figure 4. Neural network structure

Q model is transferred onto $\overline{Q}$ and used for further decisions.

Such a plain scheme can be, however, implemented with subtle differences regarding things such as initial behavior or extensions to a multi-agent scenario, which is particularly our case and our contribution. All in all, DQN procedure has quite a large number of metaparameters, and full grid search could be prohibitive.

3. Experiments and Results

Our structure of neural network for Q function modeling is shown with all variants in Figure 4. In its basic form it is similar to widely used classification models used in, for instance, optical character recognition. The input contains road situation (blue) and collision structures (orange). The first three layers of the main network perform 3D convolutions in 20 filters, with convolution window of size $2 \times 2 \times 10$, while the input size is $N \times N \times 10$ (the third dimension corresponds to 5-tuple (**C,P,G,D,B**), taken twice, at the current step n and the previous step, $n-1$. The last two layers are dense ones, and the output size is 2×3, that is, two values of speed (slow and fast) by three values of direction (turn left, drive straight, or turn right).

Model training and evaluation has been performed on a track shown in Figure 3. The agent scenario is to approach targets 0 to 4, in a loop. Once an agent gets close to a target, it is given another one. The plane is colored accordingly, giving an agent hints about the general driving direction – save for the intersection where no driving preference is provided lest it divert any of traffic streams that should pass the intersection straight ahead.

The training is performed on four training segments, corresponding to the targets 0–3. An agent is spawned on a track fragment within the rectangular areas, and directed towards the goal. Both the initial position and orientation are subject to random disturbances.

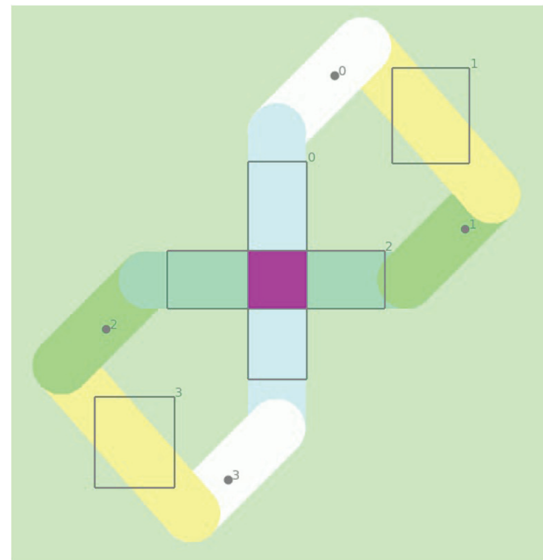

Figure 5. Location of training segments

The training segments are chosen so that agents can learn skills typical to this track, such as driving straight and taking slight and perpendicular turns in both directions. Note that learning each type of turn is done only in one part of the track, but is expected to be utilized by the agent elsewhere on its route.

Among many parameters, we decided to keep reward components constant, with values $\varrho_f = 0.5$ [sec/m], $\zeta_r = \zeta_v = -10$ [sec/m], $\varrho_t = 2$ [sec/m], $\zeta_f = -10$. Importantly, rewards ϱ_f and ϱ_t are small w.r.t. all penalties, therefore leaving much freedom of strategy to agents, unless they really violate rules imposed by the ζ's. After extensive preliminary tests, we have found that solution quality is not much sensitive to

Table 1. Solution quality in various training and testing settings

Training variant	grid 5x5				grid 7x7				grid 11x11			
	6 agents		10 agents		6 agents		10 agents		6 agents		10 agents	
	max	mean	max	mean	max	mean	max	mean	max	mean	max	mean
I	**0.64**	*0.34*	**0.53**	*0.38*	**3.52**	*0.37*	**1.05**	*0.33*	**3.05**	*1.53*	**1.07**	*0.65*
II	**3.27**	*0.70*	**0.63**	*0.27*	**3.34**	*0.98*	**0.84**	*0.44*	**1.71**	*0.71*	**0.55**	*0.28*
III	**5.27**	*1.07*	**0.61**	*0.28*	**0.60**	*0.36*	**0.35**	*0.23*	**4.66**	*0.78*	**0.62**	*0.47*

DQN-related parameters. Consequently, we decided to keep them close to the values from the original project [12]. In particular, reward discount was set to 0.9, the memory buffer contained 20,000 recent steps, model $\overline{Q}$ was trained every 4 steps, and transferred onto Q after every twentieth training session.

The experiments were designed to verify importance of the training length and the size of agent's location grid. The verification was measured by the mean number of loops done by an agent, until failure. It was calculated in variants with 1, 6, and 10 agents run simultaneously. The training for multiagent variants was done with 6 agents simultaneously, all learning the same uniform strategy.

The learning in multiagent setting can be perceived as an extension of a simpler one-agent scenario, where a part of the model handles interaction between agents. We trained multiagent models in three variants:

I) Uniformly, by training from scratch. In such a case, the agents learn at the same time, traversing their routes and avoiding collision.

II) By transfer of some part of a one-agent model onto multiagent model. Selected weights in the first convolution layer get initialized from the best single-agent model, and frozen, while other weights undergo training as in approach I. Part of the model with the frozen weights has a grey color in Figure 4.

III) By sequentially training a branched model, whose one branch processes only track-related data and is similar to the uniform model (I). The other branch processes only collision-related data, and its training starts later, once the first branch is trained well. The two branches are merged canonically in the second last dense layer. The other branch is drawn in the lower part of Figure 4.

Comparison of model performance, in terms of average number of loops by an agent before failure, is given comprehensively in Table 1. Bold numbers represent best results in a range of model training epochs (1500, 1750, and so on until 3000), while italics stand for their mean value. Models were trained for the same field of view of 200 by 200 pixels – divided into grids of 5x5, 7x7, and 11x11 cells, respectively. Additionally, we ran performance checks on a more crowded track, for example, with 10 agents.

The results consistently show that increasing the number of agents impairs overall performance. Other results are not consistent across training variants – especially the hypothetical improvement for finer grid resolutions. Interestingly, the solution quality increase is observed clearly only for variant I, and in the mean sense (figures in green in Table 1.)

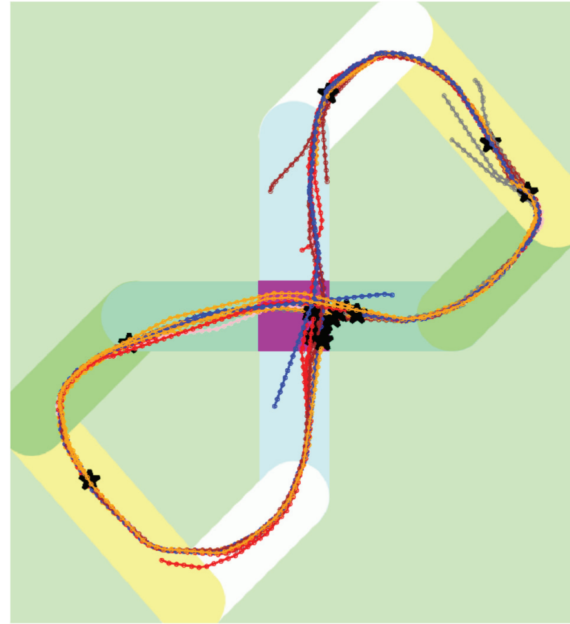

Figure 6. Sample agent trajectories, variant IIIwith 6 agents and grid resolution 5x5

Otherwise, we may clearly point out that results considered best for models II and III definitely outperform the plain and uniform one (I) – compare figures in red, columnwise. Let us examine the best result ever, obtained by variant III in 5x5 grid, and best mean result obtained by variant II in 7x7 grid. Sample agent traces for the earlier case are shown in Figure 6.

We can observe that agents, started at various locations, quickly converge to optimal trajectory. In a track section on the left to the intersection, where there is no apparent obstacle in view, agents tend to fan out and take alternative, equally optimal routes.

No departures from the track are observed and the only scenario terminations are due to agent collisions. Most of them happen at the intersection or immediately in front of it (which is a crash zone size artefact). In this setting, agents plainly did not develop any crash avoidance strategy.

Occasional crashes in other parts of the track are result of respawns of agents that happen to take place in too close proximity of another agent and can be easily eliminated by improving agent respawn procedure in the testing phase.

Sample agent traces in the latter selected case are shown in detail in intersection area in Figure 7. Agents' goal is to pass the intersection straight ahead. Here

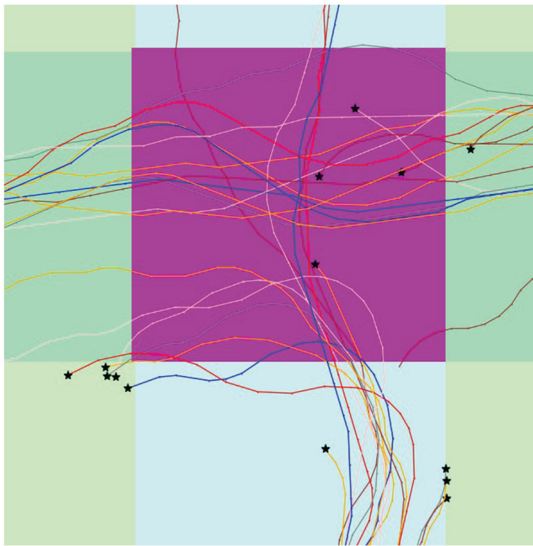

Figure 7. Collision avoidance, variant II with 6 agents and grid resolution 7x7

we can observe that developed strategy is to avoid collision by slowing down and diverging from own course – to either right or left. Skipping right results in almost immediate track fallout and episode termination. The decision of skipping left is taken if agent has spotted another agent crossing the intersection slightly later. It turns left sharply, taking course parallel to it. Consequently, the agent is pushed off the track or strays left, departing from its goal and eventually being terminated.

4. Conclusion

Here we presented a complete simulation and reinforcement learning environment, capable of training autonomous mobile agents to reach their goals. Our main contributions are twofold. First, a plane coloring scheme is proposed that efficiently encodes both track shape and basic traffic rules. It allows feeding the neural network model with very compact input, without need of transfer learning from big general pretrained computer-vision models.

Second, three variants of network architectures and their training procedures are proposed and examined, with the aim of somehow decoupling track control and collision avoidance agent skills. Experiments show that model structure matters in this regard: for instance, model variant II clearly develops a primitive but rational strategy of avoiding collision by falling off the track. Further improvements, including some sort of self-developed road code, would probably require reformulation of agent reward, and are a valuable topic for further research.

The inherent compact size of the model was a design guideline in our work. Being not a result of quantization or any other compression of a much larger model, it can be used directly by agents not only in the execution phase but also in training or updating the model itself. Practical verification of this claim is also an important research direction.

AUTHOR
Mariusz Kamola* – Institute of Control and Computation Engineering, Warsaw University of Technology, Warsaw, 00-665, Poland, e-mail: Mariusz.Kamola@pw.edu.pl.
NASK – National Research Institute, Warsaw, 01-045, Poland, e-mail: Mariusz.Kamola@nask.pl.
*Corresponding author

ACKNOWLEDGEMENTS

The author thanks Mr. Wojciech Dudek, PhD, Warsaw University of Technology, for careful development and maintenance of the Turtlesim simulator, and his advice and help in the course of research reported here.

References

[1] E. Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP,” *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. doi: 10.48550/arXiv.1906.02243.

[2] Y. Cheng, et al. “Model compression and acceleration for deep neural networks: The principles, progress, and challenges.” *IEEE Signal Processing Magazine* vol. 35, no. 1, 126–136, 2018. doi: 10.48550/arXiv.1710.09282.

[3] S. Grigorescu, et al. “A survey of deep learning techniques for autonomous driving,” *Journal of Field Robotics*, vol. 37, no. 3, 362–386, 2020.

[4] M. Hessel, et al. “Rainbow: Combining improvements in deep reinforcement learning,” *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[5] W. Dabney, et al. “Distributional reinforcement learning with quantile regression,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[6] M. Ahmed, C. P. Lim, and S. Nahavandi. “A Deep Q-Network Reinforcement Learning-Based Model for Autonomous Driving,” *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2021.

[7] J. Carreira, and A. Zisserman. “Quo vadis, action recognition? a new model and the kinetics dataset,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017. doi: 10.48550/arXiv.1705.07750.

[8] “How do self-driving cars know their way around without a map?”, https://bigthink.com/technology-innovation/how-do-self-driving-cars-know-their-way-around-without-a-map/ (accessed 2023.03.31).

[9] M. Sewak. “Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence,” *Deep Reinforcement Learning:*

Frontiers of Artificial Intelligence 2019, 95–108. doi: 10.1007/978-981-13-8285-7_8.

[10] W. Dudek, N. Miguel, and T. Winiarski. "SPSysML: A meta-model for quantitative evaluation of Simulation-Physical Systems," arXiv preprint arXiv:2303.09565 (2023). doi: 10.48550/arXiv.2303.09565.

[11] F. S. Chance. "Interception from a Dragonfly Neural Network Model," *International Conference on Neuromorphic Systems*, 2020.

[12] "Self-driving cars with Carla and Python," https://pythonprogramming.net/introduction-self-driving-autonomous-cars-carla-python (accessed 2023.03.31).

[13] OpenAI Gym homepage, https://openai.com/research/openai-gym-beta (accessed 2023.03.31).

[14] J. Lin, C. Gan, and S. Han. "TSM: Temporal shift module for efficient video understanding." *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.