I, Jordan Ambs, hereby certify that I have written this thesis independently and that all sources are listed in the Bibliography.

All passages taken verbatim or in spirit from published or as yet unpublished sources are identified as such.

The drawings or illustrations in this work have been created by myself or have been provided with an appropriate source reference.

This work has not been submitted in the same or similar form to any other examination authority.

Frisco, TX, 03.29.2023

. . . . . .
Jordan Ambs

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Optimization of Sensor Placement for Autonomous Driving Vehicles Using Neural Networks and Deep Learning

Bachelor's Thesis

for application of the degree

Bachelor of Engineering

presented

Jordan Ambs (32275)

March 29, 2023

Supervisor: Prof. Dr. Stefan Elser

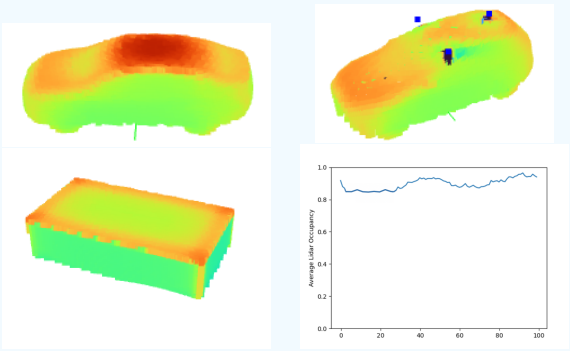
Advisor: Felix Berens

- *Abstract*

During the transition to fully autonomous driving vehicles, Light Detection and Ranging (LiDAR) sensors have proven indispensable in providing the autonomous control algorithm with accurate and up-to-date information about the vehicle's environment. The placement space for these sensors is non-convex, including the three dimensional position of each sensor, and significantly influences the ability of an autonomous driving vehicle to perceive its 3D environment. Algorithms have been created to solve this problem, notably Kim and Park, and Berens et al. This article introduces a new method to iterate upon the previous research; the existing evaluation and optimization algorithms are reclassified as machine learning problems, and the resultant methods applied. Testing has been performed on a number of vehicles as well as a simplified test-case of a rectangular prism of size $3m \star 2m \star 1.8m$. These experiments show how the proposed methods can be used to optimize the sensor configuration for any type of autonomous vehicle.

This article uses a Deep Deterministic Policy Gradient (DDPG) to automate the placement of LiDAR sensors. The DDPG is trained using reinforcement learning, within the CARLA autonomous driving simulator. As an input, the DDPG network receives a set of $n_{vehicle}$-points describing the surface of the vehicle. As an output, the DDPG returns a set of n_{sensor}-points in space representing the sensor placement(s) with the largest predicted visibility. The visibility is simplified to a scalar value $r \in [0, 1]$, with 1 corresponding to full coverage by every sensor, and 0 to no coverage by using the metrics from Kim and Park and expanded upon by Berens et al. This is the same value used to evaluate the DDPG during training.

While the training of the network is heavily resource and time intensive, the final trained network is able to propagate the input data for a new example in trivial time. The trained network is also able to evaluate novel example vehicles without re-optimization. This represents the most significant improvement upon previous research.



(a) Example Vehicles (b) Example Training

Contents

1 Introduction

In recent years machine learning has gained immense popularity in various technical fields. This is partly due to the myriad of applications in which the re-surging technique has been newly employed [13] [2] [16]. One such cutting-edge application of neural-network aided design is its use in autonomous driving vehicles [7]. Much progress has been made in this field, and much more remains to be refined, especially with respect to the vehicle control algorithms and object recognition [12] [15]. One technology that has aided in the pursuit of the goal of fully autonomous driving is the use of LiDAR sensors [4].

LiDAR sensors do not solve all issues and must be used strategically to be effective [9]. One such way to ensure successful application is the efficient placement of each sensor. To aid in the creation of standards for the placement of such sensors, various methods have been developed, especially from Kim and Park [8], and Berens et al [6]. This paper aims to use the recent advances in machine learning to tackle these problems for an arbitrary number of sensors.

All Python code can be found in a GitHub Repository at

`https://github.com/jdrew1/SensorOpt`

1.1 Simulator

This paper uses the CARLA [5] autonomous driving simulator, a widely used open-source environment for autonomous vehicle development, but these methods can be abstracted to other simulators as well. The CARLA Simulator operates as a physics simulation from which the environment data including the position and momentum of objects over time, can be extracted.

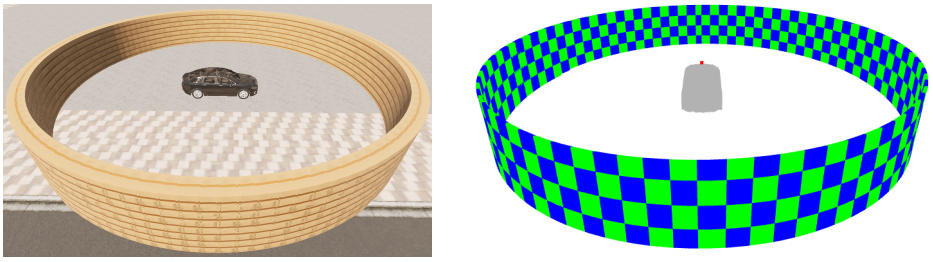
The CARLA environment allows access to ray-casting LiDAR sensors. The attributes of theses sensors such as rotation speed, range, and field of view can be manipulated

arbitrarily.

In order to evaluate the amount of its surroundings any sensor can see, a cylindrical test surface is used. The height h and radius r of this cylinder can be altered to account for differing applications, as seen in Figure 1.1a. To take advantage of symmetries, the coordinates are converted from cartesian to cylindrical coordinates defined by radius ρ, azimuth angle ϕ and height z.

The cylinder is then split into N_a sections along ϕ and N_h sections along z In order to ensure equal spacing along each dimension, $N_{a,h}$ can be calculated with: $N_a = \frac{2\pi r}{\delta}$ and $N_h = \frac{h}{\delta}$ where δ is a constant equal to the number of divisions per unit (fig. 1.1b).

A LiDAR sensor is then created and evaluated based upon the number of sections on the cylinder to which it can successfully ray-cast. The exact calculation is discussed in Section 2.4.



(a) Cylinder in CARLA (b) Cylinder split into sections

Figure 1.1: Test Cylinder for LiDAR Occupancy Calculation

1.2 Deep Neural Networks

The deep neural network and machine learning methods used in this paper are derived from the Tensorflow [1]and Keras [3] libraries.

Several types of neural networks were tried and evaluated during the development phase. The final algorithm uses a Deep Deterministic Policy Gradient (DDPG) [10]. This is a type of deep reinforcement learning based on a modified version of Q-Learning [17] [11].

Standard Q-Learning is based around using a neural network to approximate an environment's Q function (3.1) which maps the combined state-action space (s, a) to

the reward $a(s) = r$ space.

The DDPG approach alters this method in a few key ways:

- The action space is generalized to a bounded continuous space via the use of policy gradient.
- This approach makes use of two neural networks. The actor and critic networks approximate the Q function and reward functions respectively.
- Deep Deterministic Policy Gradient maintains two versions of each actor-critic models, the training and target models. The training actor-critic network has a larger learning rate and more noise to allow for more efficient searching of the environment.
- DDPG uses experience replay. The experience buffer stores a set of D experiences containing $\{observation,\ action,\ reward\}$ This allows the network to learn from multiple episodes simultaneously.

2 CARLA Environment

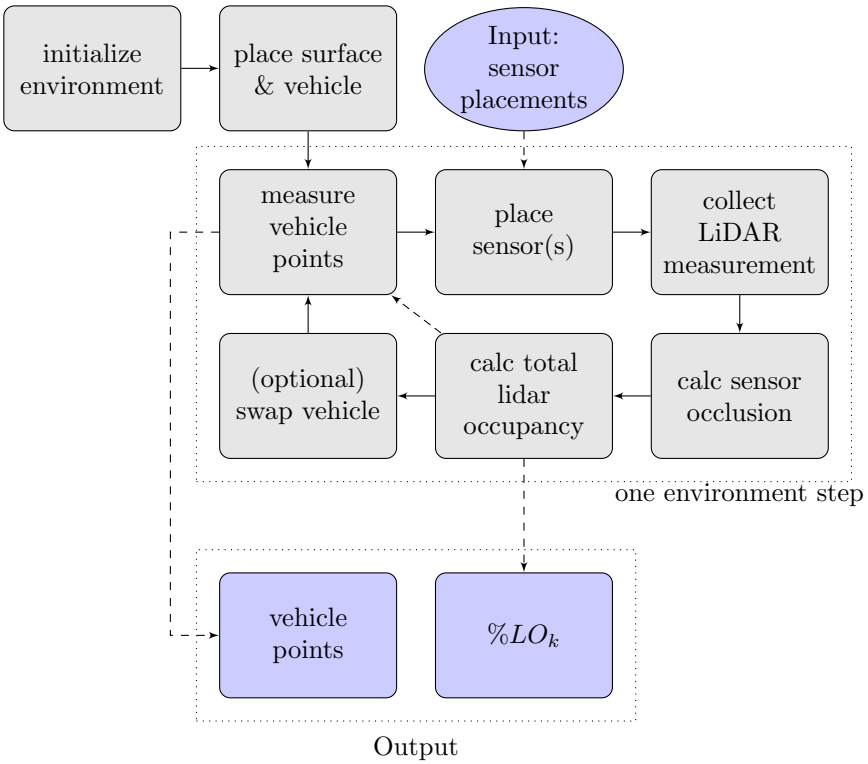


Figure 2.1: CARLA Environment Time Step Cycle

2.1 Overview

Methods were built within the CARLA simulator to control the placement and evaluation of a virtual LiDAR sensor, see Figure 2.1. The evaluation of the sensor follows the Total LiDAR Occupancy methods outlined by Kim and Park [8] and reiterated in the Section 2.4. To summarize, a cylindrical test surface is created at a radius of 10 meters and vehicle is placed in the center. The environment measures and returns a set of points

describing the vehicle's shape, and a LiDAR sensor is placed in a potential location. A set of points is collected from the sensor representing its range of view including any potential occlusion, and the LiDAR Occupancy objective function is calculated. This represents a single time step during the training.

2.2 Pre-Processing of Vehicle Data

The collection of vehicle points is non-deterministic and results in a randomly shuffled set of points. This is useful as it allows us to provide the network with more expressions of the same data simply by measuring the vehicle each time step. To reduce training complexity and allow access to larger patterns and structures, however, the set of vehicle points is sorted in ascending order along the x then y then z axes before being passed to the DDPG network, see Figure 2.2.

Figure 2.2: Sorted and Unsorted Points Collected

2.3 Sensor Occlusion

In the CARLA simulator, each LiDAR sensor is infinitesimally small. The occlusion of one LiDAR by another must be done manually.

The sensors are modeled as a cylinder of diameter $0.1m$ and height $0.07m$ (as with the Velodyne Puck), and the set of subareas bounded by the maximum and minimum azimuthal and inclination angles are calculated for each occluding sensor, shown in red in Figure 2.3.

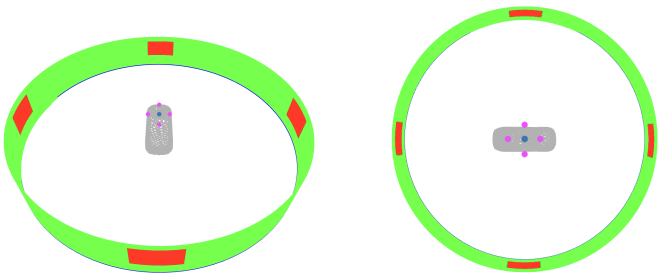


Figure 2.3: Visualization of Sensor Occlusion. Red areas are blocked by sensors surrounding the central measuring sensor.

The final set of points collected is the difference between the set of un-occluded points and all occluded areas, shown in green in Figure 2.3. This occlusion is included during the calculation of $\%LO_k$ (eq. 2.3).

2.4 LiDAR Occupancy

The LiDAR Occupancy is the total portion of the test cylinder which can be seen by the sensor. The cylinder is divided into N_a segments along the azimuthal plane, and N_h segments along the z axis, giving us $N_a \cdot N_h$ subsections, as seen in Figure 1.1b. the LiDAR occupancy function of a single subsection is then as follows:

$$lob_1^p(i,j) = \begin{cases} 1 & \text{if n(i,j)} \geq \text{p} \\ 0 & \text{if n(i,j)} < \text{p} \end{cases} \tag{2.1}$$

$$\text{with } i \in \{1,\ 2,\ 3,\ \ldots\ N_a\}, \quad j \in \{1,\ 2,\ 3,\ \ldots\ N_h\}$$

$n(i,j)$ is the number of sensors which have at least one ray-cast point in the subsection (i,j) (p is only written if $p \neq 1$).

To consider overlap of multiple sensors, the following weighting function is used:

$$lob_k(i,j) = \frac{2 - \frac{1}{2}^{m(i,j)-1}}{2 - \frac{1}{2}^{k-1}} \tag{2.2}$$

$$\text{with m} = \sum_{p=1}^{k} lob_1^p(i,j), \text{ and } k = \text{number of LiDAR sensors.}$$

This function prioritizes having at least one sensor able to see a subsection, with diminishing returns for every subsequent sensor.

The result is then normalized as a fraction of the total surface and the final LiDAR Occupancy is given by:

$$\%LO_k = \frac{\sum_{i=1}^{N_a} \sum_{j=1}^{N_h} lob_k^p(i,j)}{N_a \cdot N_h} \tag{2.3}$$

3 Deep Deterministic Policy Gradients

3.1 Overview

Methods were written to create, train, and evaluate a neural network based upon the Deep Deterministic Policy Gradient approach [10]. The network was tasked with optimizing the sensor placement on a vehicle based on the highest $\%LO_k$ (eq. 2.3).

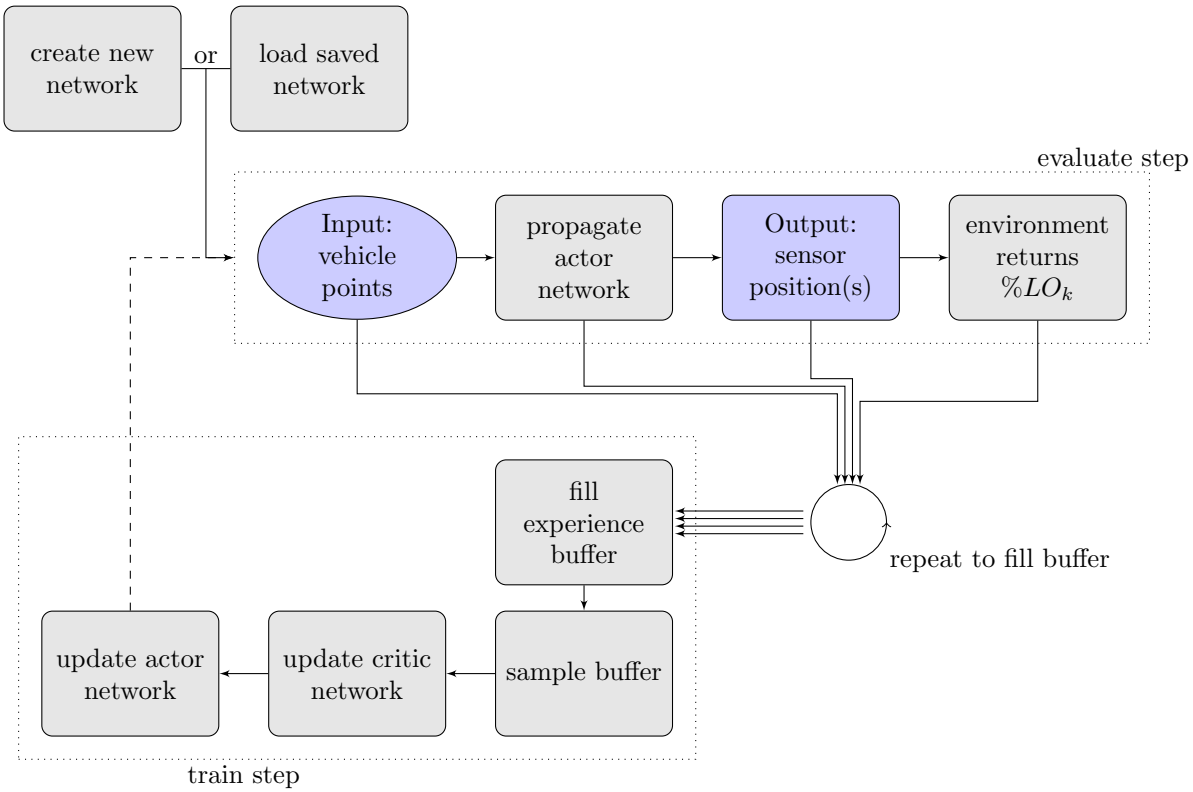


Figure 3.1: Reinforcement Training Cycle

Once the network is initialized, it interfaces with the CARLA environment and several

examples may be calculated. When evaluating the network, the algorithm concludes here, having returned optimal LiDAR sensor position(s).

During training, several environment time steps are collected and stored in a memory buffer. The buffer is sampled, and the neural network weights are updated to optimize towards a higher $\%LO_k$ (eq. 2.3) over time.

This results in training the network against an older version of itself. For this reason, Deep Deterministic Policy Gradient is considered and off-policy algorithm.

3.2 Deep Q Learning

Standard Q-Learning is based around and environment's Q function which maps the combined state-action space to the reward space given a policy π. Each time step may potentially alter the internal state of the environment.

$$r(s,a) = Q_\pi(s,a) \tag{3.1}$$

In this example, the environment state is the vehicle being optimized. The action then corresponds to the LiDAR placement, and the reward is the $\%LO_k$ (eq. 2.3).

The task of a Deep Q network is then to approximate the Q function such that the action corresponding to the largest reward can be found for a given state: (eq. 3.1). This optimal equation obeys the Bellman Equation:

$$Q^\star(s,a) = \underset{s' \sim D}{E}\left[r(s,a) + \gamma max_{a'} Q^\star(s',a')\right] \tag{3.2}$$

with s' denoting the next step.

To update the approximation of the optimal Q function, the mean square error of the Bellman Equation for a policy π after $d \in D$ experiences is calculated:

$$L(\pi, D) = \underset{(s,a,r)\sim D}{E}\left[\Big(Q_\pi(s,a) - \big(r(s,a) + \gamma max_{a'} Q_\pi(s',a')\big)\Big)^2\right] \tag{3.3}$$

Because this example of optimization does not have a time component, the discount factor γ is not needed, and the loss calculation can be simplified to:

$$L(\pi, D) = \underset{(s,a,r)\sim D}{E}\left[\left(Q_\pi(s,a) - r(s,a)\right)^2\right] \tag{3.4}$$

The loss values calculated from this equation (3.4) are then used to update the neural network weights. The final trained Q function contains the optimal action for a given state with:

$$max_a r(s,a) = max_a Q^\star(s,a) \tag{3.5}$$

3.3 Extension to Continuous Action Space

Computing the max function in equation 3.5 is simple for environments with discretized action spaces.

This is typically done via a 'one hot' encoding of the output vector. For example, the output $\{-0.6, 0.2, 0.8, 0.1\}$ in an environment with four possible actions represents a strong disinclination to take action 0, no preference for actions 1 or 3, and a preference for action 2.

In the continuous case, the direct vector output of the network is used as placement for the sensor.

As example, the output $\{0.3, 0.6, 1.4\}$ represents a sensor at $\begin{bmatrix} x = 0.3 \\ y = 0.6 \\ z = 1.4 \end{bmatrix}$ in euclidean space.

Because of continuity, there is no way to exhaustively search this output space. To solve this issue, a trained actor function is used (eq. 3.6) which maps environment states to actions based on parameters θ.

$$\mu_\theta(s) = a \tag{3.6}$$

3.4 Actor-Critic Approach

The task of calculating the max return of the Q function can be taken by the trained actor network such that the optimal actor function is given by:

$$\mu_\star(s) = max_\theta \underset{s\sim D}{E} [Q_\pi(s, \mu_\theta(s))]) \tag{3.7}$$

With a continuous action space, the Q space of an environment can be assumed differentiable with respect to action, giving the following chain rule to update the actor network:

$$\nabla_{\theta\mu} J = \mathbb{E}_{s_t\sim\rho^\beta} \left[\nabla_a Q_\pi(s,a)|_{s=s,a=\mu(s_t)} \nabla_{\theta,\mu}\ \mu_\theta(s)|_{s=s_t}\right] \tag{3.8}$$

In this fashion, the actor-critic pair of networks is trained iteratively against the ground-truth of the environment, optimizing the return ($\%LO_k$ or eq. 2.3) over time. In practice, the optimization of both actor and critic networks loss function is done through the Tensorflow Adam optimizer, as it is far more advanced and robust than simple gradient descent.

This actor network approximates the optimal policy of the Q function, and is also known as a policy gradient network.

3.5 Time Dependent Noise Reduction

An arbitrary noise function $N(\sigma)$ which returns a probability distribution around a mean of 0 is used to randomly sample noise for exploration during training.

During training, the standard deviation σ of the noise function is decreased. This allows for large variability during the beginning exploration phase, while maintaining a smaller randomness to ensure convergence during later training.

For this paper, the Ornstein–Uhlenbeck Process [14] is used to generate a random walk, and the inverse function is used to decrease standard deviation monotonically.

$$\sigma = \frac{1}{\frac{t_s}{\lambda} + 1} \tag{3.9}$$

where $\lambda \in \mathbb{R}^+$ is a scaling factor to control the speed of reduction and $t_s \in \mathbb{N}$ is the

number of elapsed training steps.

3.6 Target Models

To aide in convergence of training, DDPG maintains two versions of each actor-critic models, the training and target models.

The term $r + \gamma(1 - d)max_{a'}Q_{\pi}(s', a')$ of equation 3.3 is known as the target, and represents the final training output. Because the policy π depends on the same weights ϕ being trained, the minimization is unstable. To solve this, a separate copy of our target network is maintained and its weights ϕ updated by averaging the weights of the learning network over time. This is done via weighted Polyak averaging such that:

$$\phi_{\text{target}} = \rho\phi_{\text{target}} + (1 - \rho)\phi_{training} \tag{3.10}$$

where $\rho \in [0, 1]$ is the target rate, typically close to one.

3.7 Experience Replay

The gradient calculations (eq. 3.8) during training benefit from more input data for a single iteration. To provide several measurements during a single training iteration, DDPG uses experience replay. The experience buffer stores a set of $d \in D$ experiences containing

$$D \ni d = \{state,\ action,\ reward\} \begin{cases} \text{state} = \text{vehicle points} \\ \text{action} = \text{LiDAR placment(s)} \\ \text{reward} = \%LO_k \end{cases} \tag{3.11}$$

This allows the network to learn from multiple episodes simultaneously, but must be tuned. If the buffer is too short, the model over-fits the most recent training data. If the buffer is too long, the final training rate is greatly slowed, or too dependent on earlier to time steps to train at all.

4 Results

4.1 Overview

Training the network is sensitive to hyper parameters and training approach, but settles on an optimum solution and maintains accuracy when encountering new examples.

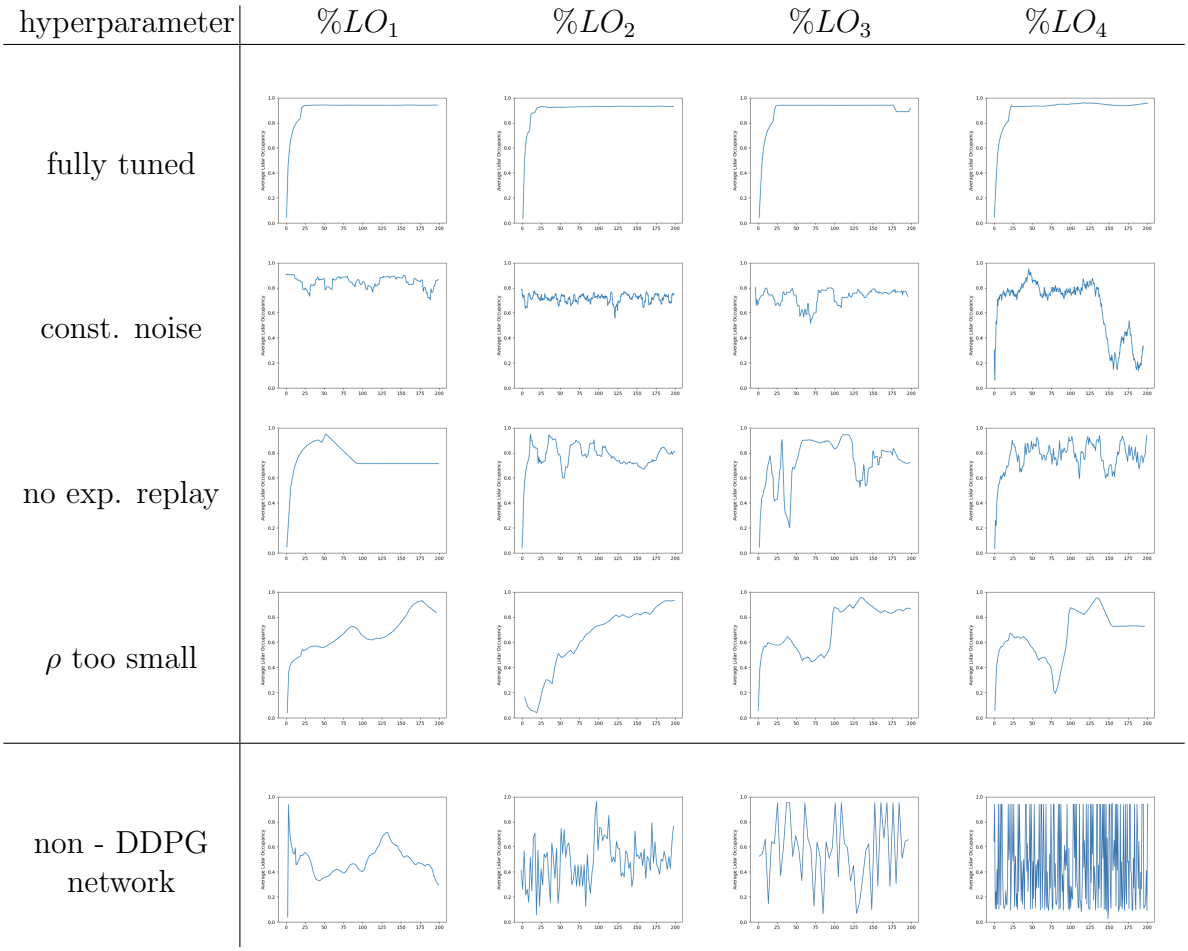


Figure 4.1: Training Data from multiple network configurations

Unless otherwise stated, the $\%LO_k$ (2.3) is calculated using a cylinder division size of $0.2m$, a cylinder radius of $10m$, and redundancy scaling as in equation 2.2. The minimum and maximum values of an output are 0.0 and 1.0 respectively.

Figure 4.1 shows some of the issues that can be encountered during training, and the resulting $\%LO_k$ (eq. 2.3) over 200 training iterations. Each column contains the calculation for different number of sensors (1 through 4), and represents increasing action space complexity.

Each row attempts to train the network using non-optimal hyper-parameters. The clear difference in training over time demonstrates some of the sensitivity to hyper parameters during training.

For the single sensor case, non DDPG machine learning algorithms could be capable of solving the optimization, but are quickly overwhelmed when the complexity increases.

4.2 Visualization of Trained and Untrained Q Function

As a rationality test, the critic network's (section 3.4) Q function (eq: 3.1) can be visualized across a vehicle for the trained and untrained case. The trained example in Figure 4.2b has the same overall structure of the ground truth in Figure 4.2c.

(a) Untrained Critic (b) Trained Critic (c) Ground Truth

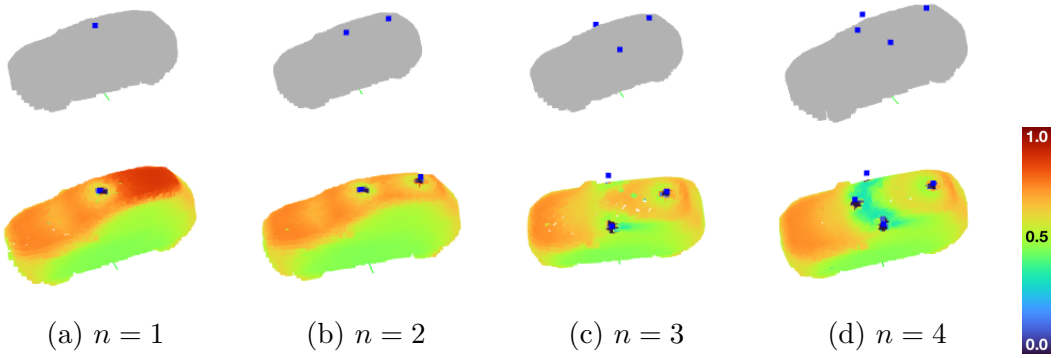
Figure 4.2: Visualization of Critic Network Q Function for a Single Sensor

The untrained critic network is fundamentally random and unrelated to the ground truth. After training, the network has learned the structure of the reward space. The minimum expected value is however higher than the true minimum $\%LO_k$ (eq. 2.3).

The actor network always acts on the highest expected return and this difference has no practical effect.

4.3 Congruence with Previous Research

To ensure that the new methods produce sensor placements in accordance with the previous algorithms, the Carla 'audi.etron' vehicle was evaluated as in the paper by Berens et al. [6]. The placement as seen in figure **??** closely resembles the optimum found in previous research, though slightly different due to stochastic search methods.



(a) $n = 1$ (b) $n = 2$ (c) $n = 3$ (d) $n = 4$

Figure 4.3: First Row: Placement of n Sensors after Training
Second Row: $\%LO_k$ of an Additional Sensor

4.4 Detailed Training Example

For a more in depth look at the training, an example vehicle will be considered over the first 100 training iterations.

The first 20 iterations (fig. 4.4a) are largely random due to the large noise and untrained network, this phase can be thought of as the exploration phase. By evaluating examples sampled across the entire vehicle, the network trains the larger structure of the action space.

(a) 1-20 (b) 21-40 (c) 41-60 (d) 61-100

Figure 4.4: Example of training convergence over 100 episodes for a single sensor

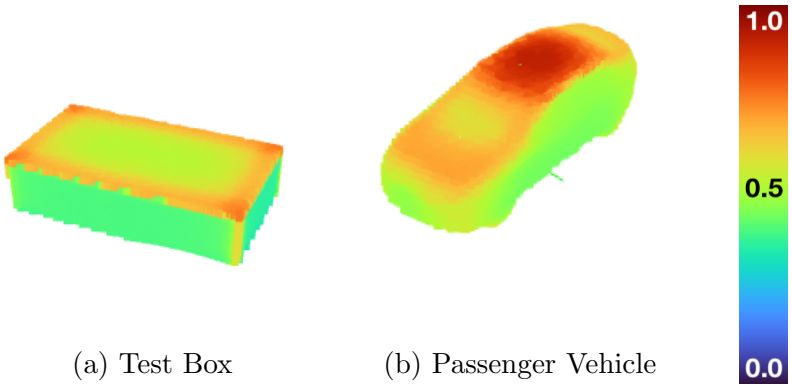
The second section (fig. 4.4b) retains the randomness, but is more centralized. The areas of lowest reward have been learned and are avoided.

Next, the network begins converging towards the top of the car (4.4c). This section can be thought of as extensively searching the area of highest return

Finally, the network converges to an optimal placement on the top of the vehicle (4.4a). The noise is now greatly reduced thanks to the time-dependent noise function (eq. 3.9), and any subsequent iterations retain this placement.

4.5 Verification Data Set

To verify the training methods, a verification data set is used. This is the test box of size $3m \star 2m \star 1.8m$. The optimal solution has been shown in Berens et al [6] to be a set of sensors along the top perimeter of the shape.



(a) Test Box (b) Passenger Vehicle

Figure 4.5: $\%LO_1$ across vehicle and verification meshes

After training on the various vehicle meshes, the test box was run through the network without updating the internal weights to evaluate its accuracy. The actor network consistently returns an evenly spaced set of sensors along the top perimeter after sufficient training.

Table 4.1: Result of Test Box evaluation with 4 sensors after 200 training iterations given different data. Results collected from 20 iterations of the CARLA environment.

Training Data	Average $\%LO_4$	Standard Deviation	Placed on Corners
Test box	0.94	0.03	Yes
All vehicles	0.90	0.11	Yes
$\frac{2}{3}$ of all Vehicles	0.85	0.16	Yes
Untrained	0.47	0.29	Inconsistent

The average $\%LO_k$ and correct placement of the network trained only on the test box in table 4.1 demonstrates a clear ability of the network to optimize a single example.

The similar results for all but the untrained network demonstrates the ability of the network to use stored data from previous training and extrapolate for a novel example.

4.6 Train - Test Data Set

Another method of evaluating the training of a reinforcement model is to withhold some of the training data set as a Test Set. After the model is trained, the Test Set is evaluated.

In this evaluation, the network is not trained on the Test Set.

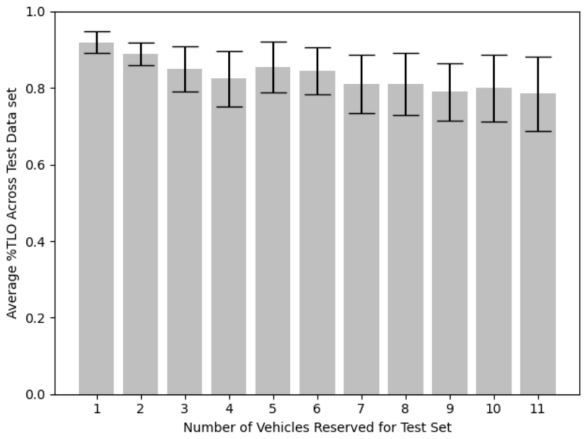


Figure 4.6: Result of Test Set evaluation after a portion of the training set is reserved.

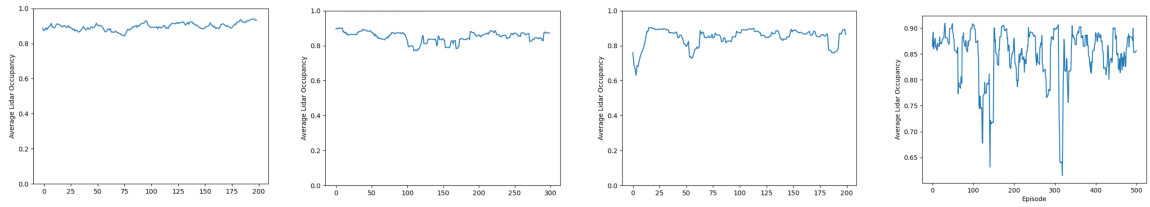
This evaluates the network's ability to use stored optimization data from previous training and successfully evaluate novel examples.

Figure 4.6 shows the network capable of responding to new examples and retaining effectiveness when encountering new examples. The CARLA environment has an included vehicle library of 33 vehicles.

In the extreme case, when a third of all examples were reserved, the DDPG maintained an average $\%LO_k$ of 0.81, only a 12% decrease in accuracy.

4.7 Training of Further Examples

To assess the DDPGs ability to quickly and effectively learn new input data, the network was trained on a subset of vehicles. After, the network was then trained on the remaining examples. With the aid of previous experience (even from an incomplete training set), the training of new examples is also greatly improved. This further demonstrates the effectiveness and expandability of Deep Deterministic Policy Gradients.



(a) Fully Trained (b) Subset of Vehicles (c) Test Box (d) Untrained

Figure 4.7: Training Data from Test Set after various training data over 200 iterations

Even when the network was trained only on the test box surface, the 'memory' provided by the previous experience greatly increases the training efficiency of further examples.

5 Conclusion

As seen in the results section, the network quickly converges on an optimal solution. Once the model is trained, the topological data is stored within the network. The evaluation of further examples then requires only a single forward propagation of the network to produce output, and can be further optimized with training of the novel example.

The non-convergence issues inherent to a continuous action space make the training stage of the network sensitive to the exact hyper-parameters, and is somewhat prone to over-fitting. Once the network has been successfully trained, the output remains stable through multiple evaluation examples.

The networks in this paper were trained on examples of passenger vehicles and test surfaces. Further research will be needed to expand the training data set from the 33 CARLA vehicles and custom test surfaces.

Additionally, the previous genetic algorithms could be used in combination with the newer deep learning solutions, 'evolving' a stable, generalized network.

These techniques have shown another, if somewhat unsurprising use case of neural networks and deep learning. With the expand-ability and modularity inherent to machine learning and expanded upon by recent research, these methods can be included in a wide range of design-optimization solutions, further closing the gap towards a fully automated design and production supply chains.

A List of Figures

B Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Jamil Ahmad, Haleem Farman, and Zahoor Jan. "Deep Learning Methods and Applications". In: *Deep Learning: Convergence to Big Data Analytics*. Singapore: Springer Singapore, 2019, pp. 31–42. ISBN: 978-981-13-3459-7. DOI: 10.1007/978-981-13-3459-7_3. URL: https://doi.org/10.1007/978-981-13-3459-7_3.

[3] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[4] Yaodong Cui et al. "Deep Learning for Image and Point Cloud Fusion in Autonomous Driving: A Review". In: *IEEE Transactions on Intelligent Transportation Systems* 23.2 (2022), pp. 722–739. DOI: 10.1109/TITS.2020.3023541.

[5] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[6] Markus Reischl Felix Berens Stefan Elser. "Genetic Algorithm for the Optimal LiDAR Sensor Configuration on a Vehicle". In: *IEEE Sensors Journal* 22, no. 3 (2022), pp. 2735–2743.

[7] Sorin Grigorescu et al. "A survey of deep learning techniques for autonomous driving". In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386. DOI: https://doi.org/10.1002/rob.21918. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21918. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21918.

[8] T.-H. Kim and T.-H. Park. "Placement optimization of multiple LiDAR sensors for autonomous vehicles". In: *IEEE Trans. Intell. Transp. Syst.* 21, no.5 (2020), 2139–2145.

[9] Ying Li et al. “Deep Learning for LiDAR Point Clouds in Autonomous Driving: A Review”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.8 (2021), pp. 3412–3432. DOI: 10.1109/TNNLS.2020.3015992.

[10] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[11] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[12] Khan Muhammad et al. “Deep Learning for Safe Autonomous Driving: Current Challenges and Future Directions”. In: *IEEE Transactions on Intelligent Transportation Systems* 22.7 (2021), pp. 4316–4336. DOI: 10.1109/TITS.2020.3032227.

[13] Pramila P. Shinde and Seema Shah. “A Review of Machine Learning and Deep Learning Applications”. In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. 2018, pp. 1–6. DOI: 10.1109/ICCUBEA.2018.8697857.

[14] G. E. Uhlenbeck and L. S. Ornstein. “On the Theory of the Brownian Motion”. In: *Phys. Rev.* 36 (5 1930), pp. 823–841. DOI: 10.1103/PhysRev.36.823. URL: https://link.aps.org/doi/10.1103/PhysRev.36.823.

[15] Ayşegül Uçar, Yakup Demir, and Cüneyt Güzeliş. “Object recognition and detection with deep learning for autonomous driving applications”. In: *SIMULATION* 93.9 (2017), pp. 759–769. DOI: 10.1177/0037549717709932. eprint: https://doi.org/10.1177/0037549717709932. URL: https://doi.org/10.1177/0037549717709932.

[16] Xizhao Wang, Yanxia Zhao, and Farhad Pourpanah. “Recent advances in deep learning”. In: *International Journal of Machine Learning and Cybernetics* 11.4 (2020), pp. 747–750. DOI: 10.1007/s13042-020-01096-5. URL: https://doi.org/10.1007/s13042-020-01096-5.

[17] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.