Decision Support System for Traffic Jams by using Artificial Intelligence

Luka Bjelica, Svetozar Vulin, Anja Buljević

***Abstract*—This paper presents one possible decision support system as partial solution of traffic jam problem using autonomous vehicles. Proposed solution is based on deep learning algorithms and artificial neural network (ANN) models which task is to make a decision whether to move the car to a side lane, or stay in the current lane. The optimality criteria is maximization of the elapsed distance in traffic by training ANN, using a supervised learning paradigm from labeled data, to control the car. Gradient descent algorithm is used for the network's parameters estimation. Verification, testing and simulation application is also presented in this paper.**

Keywords: self-driving, artificial neural networks, deep learning, gradient descent, traffic jam.

I. Introduction

Traffic congestion has become one of the biggest modern life problems. Time spent in traffic jams is irreversibly wasted. Moreover, some researches have also shown that traffic congestion represents negatively impacts the Earth's climate, leading to global warming [1]. One solution to this problem is to minimize the time spent in traffic using autonomous vehicles. This solution not only reduces air pollution and global warming rate, but also saves time for each commuter. Autonomous cars are vehicles which are driven by digital technologies without any, or little, human intervention. They are capable of driving and navigating themselves on the roads by sensing the environmental impacts. They are designed to occupy less space on the road in order to avoid traffic jams and reduce the likelihood of accidents. Autonomous cars are one of the biggest challenges in industry nowadays, so various solutions solving the automation of control of cars on the road have been widely studied in the literature [2], [3].

This paper concerns the automation of the car movement in a highway traffic jam, based on deep learning algorithms, and ANN models, like Multilayer Perceptron (MLP) [4], in order to make decision whether to move the car to a side lane, or stay in the current lane. The idea is to maximize the average speed of the car or to maximize the distance that car would elapse in certain amount of time. The car is controlled by trained ANN. Gradient descent algorithm is used for the network's parameters estimation. Simulation application could be found in [5].

L. Bjelica (bjelicaluka@uns.ac.rs), S. Vulin (svetozar.vulin@uns.ac.rs), A. Buljević (anjabuljevic@uns.ac.rs), University of Novi Sad, Faculty of Technical Sciences, Department of Computing and Control Engineering, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia.

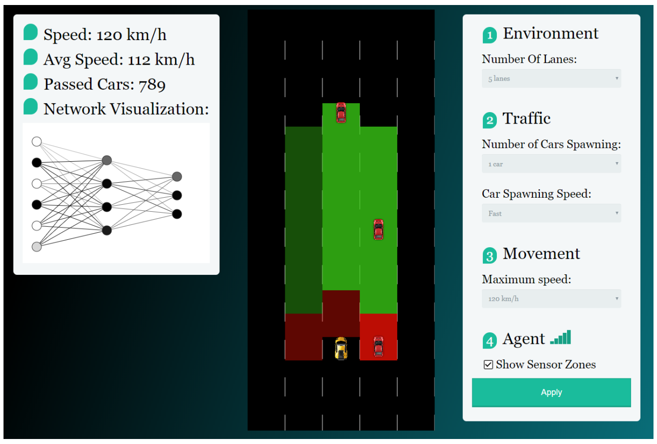
After Introduction, the paper is organized in the following manner: description of self-driving component, model of the environment, data collecting procedure and the agent controlling the car are explained in Section II, explanation of the custom deep learning library can be found in Section III, technical details about the implementation are introduced in Section IV, simulation results are shown in Section V and the concluding remarks are given in the final Section VI.

II. Self-Driving Component

As previously mentioned, the topic of this paper is the automation of car movement through a dense highway traffic environment. The real-world environment is represented in a computer simulation and due to its complexity, only main dynamic characteristics are emphasised. For the need of this paper, it is expected that the vehicle already has automatic acceleration and braking systems implemented so the emphasis is on the component responsible for highway traffic jams. That component is called self-driving component. The purpose of this component is to make sure that the agent gets out of highway traffic jams as fast as possible (maximum distance optimization problem). It is important to know that self-driving component does not ensure that the vehicle is completely autonomous, thus it only provides a solution to dense traffic.

Environment (2D traffic simulation)

The model of the environment is a simplified representation of the real world (Fig. 1). It is represented in two dimensional (2D) space. The environment consists of N traffic lanes, passing cars and the vehicle that is being controlled (agent).


Speed: 120 km/h
Avg Speed: 112 km/h
Passed Cars: 789
Network Visualization:

1 Environment
Number Of Lanes: 5 lanes
2 Traffic
Number of Cars Spawning: 1 car
Car Spawning Speed: Fast
3 Movement
Maximum speed: 120 km/h
4 Agent
☑ Show Sensor Zones

Fig. 1. Environment screenshot

Cars can be created in any of N lanes in random order and are moving in constant speed relative to each other. The agent has three sensors: front, left-side and right-side sensors, each consisting of a red and green zone. The green zone provides a distance between the car controlled by the agent and the nearest facing car in that lane. The red zone, or the safety zone, activates the braking system that manages to stop the controlled car when the car in front is detected and prohibits sidetracking when the car is detected on either left or right side. If the front sensor has not detected any cars in the red zone, the controlled car automatically accelerates until it reaches the maximum speed. This ensures that the car has a built-in safety system and that the agent has the ability to accelerate automatically when there are no cars in front of it. The environment implementation details are explained in the Section IV.

Data (collecting and labeling)

Data is collected and labeled while the user is controlling the car. Each time the user makes a decision (turns left, right or continues to move forward) a snapshot of each sensor data is collected. A snapshot is a list of data containing the following parameters:

- Left-side sensor green zone distance
- Left-side sensor red zone active status
- Front sensor green zone distance
- Front sensor red zone active status
- Right-side sensor green zone distance
- Right-side sensor red zone active status.

The label represents the user's decision (-1 for left, 0 for forward and 1 for right). After the user has provided enough training samples (the number of training samples is 28), the dataset is created which can then be used for training the neural network model.

Agent Controlling the Car

On a fixed time interval the agent queries the trained network with the current state of sensors. Agent expects to get a response containing the prediction about the action it should make. Based on the network's decision, the agent parses the response and either moves the car to a side lane, or stays in the current lane. If the car gets stuck because the network is continually making bad decisions, the user can correct the decision of a network. The corrections are saved and the user can put them in the dataset and retrain the model.

III. Deep Learning Library

For need of this paper, a deep learning library is implemented from scratch. Its model is based on the most popular deep learning framework Keras [6]. The library consists of a Model that represents the main component, Neural Network that acts as a container of parameters (weights and biases), Initializers, Operators, Losses, Optimizers, and Regularizers. The model is implemented as a Feed-Forward MLP [4], which means that each node (neuron) from every layer is connected to each node in the previous and following layer. It supports only the Supervised Learning paradigm [4], in which the desired outputs are known and the model is trained to predict future outcomes (output) depending on given (input) data.

Initializers

Initializers provide the initial values for the model parameters at the start of training. Initialization plays an important role in training deep neural networks, because bad parameter initialization can lead to slow or no convergence. Parameters of the network are initialized as small random weights drawn from the normal distribution [7].

Operators

Operators are the basic building blocks of any neural network. They are vector-valued functions that transform the data. Some commonly used operators are:

- *layers* (linear, convolution, and pooling)
- *activation functions* (Rectified Linear Unit (ReLU), Sigmoid, SoftMax, Tanh and etc.)

The only type of operators that are implemented for the need of this paper are activation functions. They are used to normalize the output of each neuron in the desired range.

Losses

Losses are differentiable mathematical expressions given in closed-form that are used as surrogates for the optimization objective of the problem at hand. Loss functions provide feedback on the training progression. They map a vector of values to a number that represents the quality of the network at a certain moment of training.

The cost or loss function has an important job in that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model. [8]

Regularizers

Regularizers provide the necessary control mechanism to avoid overfitting and promote generalization. L2 weight regularization [9] is implemented that makes the weights sparser and uniform, respectively.

Optimizers

Optimizers provide the iterative update of model parameters with respect to the optimization objective. In this context, the optimization objective is to minimize the loss function that depends on the network's parameters. The parameters are considered optimal when the loss function reaches its global minimum. Loss function determines the error between the expected value and output value. Gradient Descent optimizer [8], [10] is used for fitting the model. Gradient Descent is an iterative optimization algorithm used for finding the minimum of a differentiable function. With a goal of minimizing the loss function, this algorithm takes steps of the steepest descent which is calculated by the negative value of the gradient of the function. Iteratively moving to the minimum, the size of each step is determined by the learning rate. For network training using Gradient Descent [4], the following steps are taken:

1) **Creating a mini-batch**
   In this step, the portion of data is taken from the dataset, which is used in the present iteration. The size of the portion is equal to the batch size option that the user provides
2) **Forward pass and Calculating Loss**
   The feed-forward function is used to calculate the output of each layer. Outputs are stored in a list and are later used for calculating the gradients. Next, the value of loss function is calculated based on the output of the network.
3) **Backward pass**
   In this step, the error is being calculated and passed backward through the network. After the backward pass, deltas (the portion of error) have been calculated for each layer. It is important to apply the derivative of activation function to each delta in order to deactivate it.
4) **Backpropagation**
   This is the main step in which the gradients, based on small changes in a gradient direction, are calculated and the outputs of the layer from the feed-forward pass. In simple terms, after each *forward pass* through the network, error is calculated and *backward pass* is being performed while adjusting the network's **parameters** (*weights and biases*).
   The goal is to adjust each parameter in proportion to how much it contributes to the overall error.
   Partial derivative, of the loss function, concerning each parameter, represents the value of gradient.
   To calculate derivatives with respect to any nested variable in the equation ($\frac{\partial L}{\partial w_j}$ or $\frac{\partial L}{\partial b}$) we use the method called *chain rule* [7]. The *chain rule* shows that for the given:
   a) *preactivation* z of input x and bias b with respect to weight matrix transposed w^T is:
   $$z = w^T x + b$$
   b) value of the *preactivation* passed through *activation function* σ:
   $$\hat{y} = \sigma(z)$$
   c) *loss function (in this case cross entropy)* determines the error between the target value y and the output value of the network $\hat{y}$:
   $$L(y, \hat{y}) = -y \log(\hat{y}) - (1-y)\log(1-\hat{y}),$$
   the derivatives of the *loss function* with respect to w and b are
   $$\begin{aligned}\frac{\partial L}{\partial w_j} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_j}\\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b}.\end{aligned}$$
   After the partial derivatives of nested equations have been calculated and substituted into the chain rule the obtained result are following
   $$\begin{aligned}\frac{\partial L}{\partial w_j} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_j}\\ &= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})w_j\\ &= (\hat{y}-y)w_j\end{aligned}$$
   $$\begin{aligned}\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b}\\ &= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})\\ &= (\hat{y}-y).\end{aligned}$$
   In vectorized form with m training examples (when using Batch or Mini-Batch GD) the following equations are obtained
   $$\begin{aligned}\frac{\partial L}{\partial w} &= \frac{1}{m}X(\hat{y}-y)^T\\ \frac{\partial L}{\partial b} &= \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)}-y^{(i)}).\end{aligned}$$
   After the gradients have been calculated, the parameters are ready to be adjusted (updated) accordingly
   $$\begin{aligned}w^{(i+1)} &= w^{(i)} - \gamma\frac{\partial L}{\partial w}\\ b^{(i+1)} &= b^{(i)} - \gamma\frac{\partial L}{\partial b}.\end{aligned}$$
   γ (*learning rate*) determines how big a step the adjustment is making in the negative direction of the gradient.
5) **Adjusting Weights and Biases**
   In this step, network's parameters are updated according to the calculated gradients. If γ is too big, algorithm would diverge and never find the minimum. On the other hand, if γ is too small, it would take too much time to find the minimum. So it is necessary to find optimal value for γ in order to find minimum in reasonable time. For more details about choosing optimal γ, see [11].

Model

Model is the main component used to bind all of the above-explained components together. It represents a public Application Programming Interface (API) through which users can interact with the library, define the network's architecture and train the network.

Knowing that representability and trainability are two main attributes that describe the network, layer structure has a huge impact on the network's performance. The network is "representable" if it can represent a solution to the problem with a certain level of complexity. The higher the number of layers, the more complex problems a network can represent. Trainability sets a threshold to the number of hidden layers and the overall complexity of the network. Too many layers and a number of nodes in each layer can lead to slow convergence or high computational power demands.

The task of the network is to make a decision, whether the car should turn left, right or stay in the same lane. That decision is represented with a column vector consisting of three numbers that represent probabilities for each action. Softmax is our activation function which helps the network achieve this by squashing the outputs of each unit to be in the interval $[0, 1]$, and also dividing each output such that the total sum of the outputs is equal to 1.

IV. System Implementation

The system is distributed in two main components: the React.js application that implements the environment and a Flask web application that implements the neural network model. They communicate over HyperText Transfer Protocol (HTTP).

Network Architecture

The architecture of a network [4] used for training the self-driving agent is determined by a numerical experiment through a large number of simulations and it is shown in Fig. 2. The implemented network consists of three layers: an input layer with 6 neurons, a hidden layer with 4 neurons and an output layer with 3 neurons. The input layer does not have an activation function or more precisely, the activation function of the input layer is the identity function. The activation function used in the hidden layer is the ReLU.

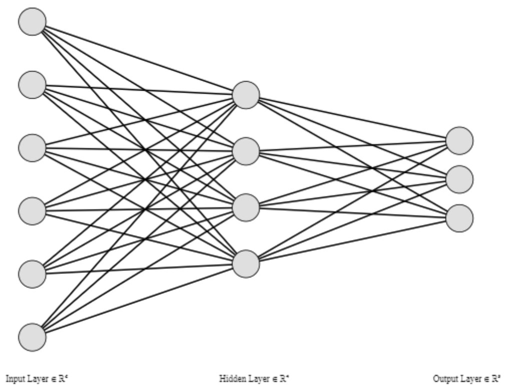


Fig. 2. Network architecture

Environment

The environment (Fig. 1) is represented through the React.js application. It is divided into three main components: highway environment, settings panel, and results pane.

The highway environment is represented through HyperText Markup Language (HTML) canvas. It consists of traffic lanes, passing cars and the controlled car. Traffic movement and canvas dynamics are managed using the p5.js library. Passing cars are spawned at the top of the screen and are moving with the constant speed relative to each other. The controlled car's sensor zones are displayed around him with a transparent background color. If a passing car is detected in any of the zones, the transparency is reduced and the color becomes more solid which indicates that the sensor is active. This is useful when collecting training data because the user can see the boundaries of the agent's "vision" and drive accordingly. The settings panel is used for changing the parameters of the environment. It is divided into three sections: environment, traffic, movement and agent settings. In the environment section, the user can change the number of traffic lanes in range of 4 to 10. In the traffic section, the car creation interval and the number of cars spawned on each interval can be modified. The maximum speed of the controlled car can be changed in the movement section and visibility of the sensor zones can be toggled in the agent subsection. Next to the agent label, the connection status with the backend application is displayed. Ability to change the parameters enables the option to change the "difficulty", or the complexity, of the environment.

In the results pane, the current speed of the car, average speed and the number of passing cars are displayed. That is the direct indicator of how well the agent is controlling the car. Below the results pane, there is a canvas that holds the figure of the neural network model. According to the outputs of each layer calculated in the forward pass, the nodes and the connection lines are shaded.

Collecting the Data: In manual mode, the user can control the car using the arrows keys and data is collected that way. The data is represented as a list of 6 numbers, a snapshot that contains information gathered from the sensors. Three of those six numbers represent a distance from the nearest car that is in the green zone of one of three sensors. In order to normalize input data, each distance is divided by the maximum distance, the green zone length, which ensures that those numbers are in interval $[0, 1]$. The other three numbers are boolean values, 0 or 1, which only indicate the presence of a car in the red zone of that sensor. Apart from the snapshot, a label that represents the user's decision is also collected in the form of a whole number between -1 and 1 (-1 for left, 0 for forward and 1 for right) and is pushed to the list. So the final result is a list of 7 numbers. The result is then added to the list of collected data that represents the dataset for training the model.

Training the Model: The model is created using the following code:

```
model = Model([[6], [4, "relu"], [3,
↪   "softmax"]],
↪   loss_function='cross_entropy',
↪   learning_rate=0.001)
```

After the model is initialized, training examples and labels are extracted from the saved .csv dataset and passed as arguments for training the model.

```
model.train("gradient_descent",
 ↪  inputs=np.array(inputs),
 ↪  labels=np.array(labels), epochs=500,
 ↪  batch_size=1)
```

Each input argument, such as learning rate, batch size or a number of epochs, has an impact on the time spent training and the quality of the results. The main goal is to find the best proportion of those two factors.

One epoch represents only one forward and backward pass of the dataset through the network. In every epoch, weights and biases are changed and adjusted. One pass leads to underfitting the curve in gradient descent algorithm, so more epochs are needed in order to achieve better results. On the other hand, too many epochs lead to a big increase in time spent training and overfitting the curve. It has been determined experimentally that the most satisfying number of epochs for this problem is 1000.

In case that the dataset is too big, data is separated into batches. Using this technique, the data can be passed in smaller groups, achieving faster training time, and having more training examples in each update of parameters. Because the self-driving dataset is not too big, the chosen batch size is 1, which means that only one training example is used for each parameter adjustment.

Agent Controlling the Car: After training the model and starting the Flask web service holding the network, the user can turn on the auto mode. In auto mode, the car is controlled by the agent. The current state of sensors and the name of the trained model are sent to the Flask web service every 500ms through an HTTP request. The input data is passed to the trained model. The model makes a prediction based on the passed data and the response is generated and sent back to the agent. The response comes in the form of a probability distribution vector. The index of the element in the vector that has the highest probability represents the action that the agent should make. The agent then moves the car based on the parsed prediction.

Each time the agent gets stuck in the traffic, the user is able to move the controlled car and correct the agent's decision. Correcting the agent means switching to manual mode. Each correction is saved in a list. Then, the user is able to download saved corrections in the form of a new dataset, add them to the initial dataset and to refine the agent's decision making by retraining the model.

Because it is expected that the controlled car has a built-in safety system, the acceleration and braking are performed automatically. When the front sensor detects that a car has entered the safety zone, it triggers the braking system and the controlled car starts to decelerate in proportion to the distance from the facing car and the current speed. If there are no detected cars in the facing line, the controlled car starts to accelerate in proportion to the current speed until it reaches the maximum speed of N km/h. N takes the maximum speed out of a set $80, 120, 160$ km/h.

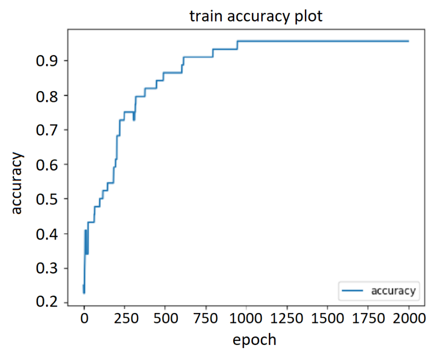
V. Simulation Results

This section presents the results of training the model, testing the agent's behaviour and comparison to the user's behaviour.

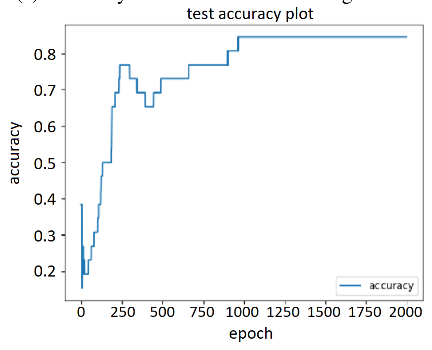
Model Training

The accuracy of the model represents how good the predictions that the model makes according to the given dataset are. It is calculated by comparing the received results from the model and expected results from the dataset represented in percentages.

Using the network architecture and training parameters explained in Section IV, the calculated accuracy over the training dataset is 95% and the calculated accuracy over the test dataset is 85%. Each training parameter affects the accuracy of the model. With an increased number of epochs, the accuracy may stay the same, but the time spent training increases. After the successful training, the obtained results are following.



(a) Accuracy of the model for training dataset



(b) Accuracy of the model for test dataset

Fig. 3. Accuracy of the model

Fig. 3a presents the accuracy of the model for training dataset and it can be noticed that the accuracy score rises from 20% to 95% within first 1000 epochs, and stays stable until the end of the training. Fig. 3b presents the accuracy of the model for test dataset and it can be noticed that the accuracy score rises from 10% to 85% within first 1000 epochs, and stays stable until the end of the training.

It is expected that the accuracy for test dataset is not as good as the accuracy for training dataset, but still test results are very satisfying.

Agent's behaviour based on a quality of the dataset

Clear information about lane changing, means positioning the agent in a good spot for evading the incoming car without hitting the safety (red) zone. A good dataset does not need to have a lot of examples, it needs to have a clear set of instructions and cover most typical scenarios making sure that it contains a balanced number of labeled decisions. It is not good if any of the three decisions is dominant in the dataset.

Training is considered not adequate if the user does not give clear information about his actions and many decisions with similar, or same, inputs are in contradiction, which makes the model disorientated and not able to move in the right direction. Even though the accuracy of the trained model is at a high percentage, the agent still acts disoriented and makes wrong turns, often getting stuck in the traffic.

Users vs Agent driving

Simulation experiment is based on three main drivers categories. First experienced user, then inexperienced user and finally, trained agent. First two categories covers manual driving with ten different participants in both categories and averaging their results. Different scenarios were tested with various maximal speeds of car, number of lanes, number of cars spawning and car spawning speed. Results of simulation are presented by total distance that car could pass in 30 minutes. Three representative results are presented in Fig. 4. Maximal speed was $120km/h$, five lanes, one spawning car with all spawning speeds modes. From this figure, it is clear that in most cases after 30 minutes of simulation, agent approach shows the best behaviour that is maximal elapsed distance.

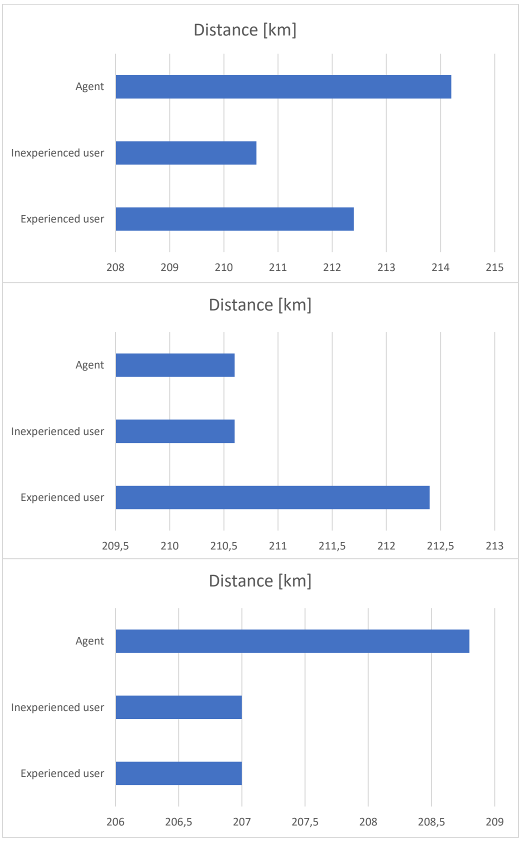


Fig. 4. Simulation results for three different scenarios. Maximal speed $120km/h$, five lanes, one spawning car with spawning speeds modes: slow (upper figure), normal (middle figure) and fast (bottom figure).

VI. Conclusion

Multilayer Perceptron Model and supervised learning paradigm used in this paper present one possible approach for training the model to autonomously drive itself trough traffic jams. The aim of this project was to achieve successful autonomous driving in a highway traffic jam in terms of maximizing elapsed distance in traffic by maximizing the average speed of the car.

After network's parameters estimation and training the model, agent's behaviour was tested in simulations. Although this solution does not ensure that the vehicle is completely autonomous, obtained training results have proved satisfying on a tested system in terms of getting out of the traffic jam as fast as possible.

In the future work, we will be focused on a new approach that will allow us to generate realistic camera image for simulation directly using sensor data collected by a self-driving car.

VII. Acknowledgment

This paper has been supported by the Ministry of Education, Science and Technological Development through the project no. 451-03-68/2020-14/200156: "Innovative scientific and artistic research from the FTS domain". (A.B.)

References

[1] J. Baron, "Thinking about global warming," *Climatic Change 77, 137–150 (2006). DOI: 10.1007/s10584-006-9049-y*, 2006.

[2] A. Takacs, I. J. Rudas, D. Bosl, and T. Haidegger, "Highly automated vehicles and self-driving cars.," *IEEE Robotics Automation Magazine 25(4):106-112, DOI: 10.1109/MRA.2018.2874301*, 2018.

[3] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving.," *Journal of Field Robotics, Online ISSN:1556-4967, DOI: 10.1002/rob.21918*, 2019.

[4] A. P. Engelbrecht, "Computational intelligence: An introduction," *Wiley Publishing, ISBN:978-0-470-03561-0*, 2007.

[5] L. Bjelica, S. Vulin, and A. Buljević. http://self-driving.bjelicaluka.live/, 2020.

[6] "Keras documentation." https://keras.io/.

[7] D. Kriesel, "A brief introduction on neural networks." http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf.

[8] J. Nocedal and S. J. Wright, "Numerical optimization, second edition," *Springer, New York, ISBN: 978-0-387-30303-1, DOI: 10.1007/978-0-387-40065-5*, 2006.

[9] A. Nagpal, "L1 and l2 regularization methods." https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c, 2017.

[10] M. J. Kochenderfer and T. A. Wheeler, "Algorithms for optimization," *The MIT Press. ISBN:978-0-262-03942-0*, 2019.

[11] Y. xiang Yuan, "A new stepsize for the steepest descent method," *Journal of Computational Mathematics Vol. 24, No. 2 (MARCH 2006), pp. 149-156*, 2006.