Implementation of Self-Supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation (Course Project for CISC 856)

Riley Cooper
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario K7L 3N6
Email: 15mrc5@queensu.ca

Jason Harris
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario K7L 3N6
Email: harris.jason@queensu.ca

Liam Horton
Department of Mechanical and Materials Engineering
Queen's University
Kingston, Ontario K7L 3N6
Email: liam.horton@queensu.ca

***Abstract*—We implemented a self-supervised deep reinforcement learning algorithm on a simulated vehicle to learn a collision-free navigation policy. This algorithm is a generalized version of model-free and model-based learning that allows for a more robust system which uses aspects of both learning techniques. By using a list of generated actions and an image taken from a simulated camera on-board the vehicle, the probability of collision for each time step is generated and used to navigate through an unstructured environment. After implementing and testing the algorithm under different conditions, the model had difficulties learning and produced sub-optimal results. Some improvements for future work on the system were identified which are planned to be pursued by all group members. The code for this project can be found at https://github.com/JasonDHarris/CISC856Project.**

I. Introduction

Individually, our research activities are focused on robots moving autonomously in the air, on land, and on the sea. Navigation capabilities in unstructured environments is a fundamental problem facing mobile robots that operate in these mediums. Traditional approaches to robot navigation involve an agent maintaining an internal map of the environment. Based on this map, the agent can employ localization and planning strategies to successfully navigate its surroundings. These approaches can be computationally expensive, include a variety of assumptions, and are often impractical to use for new or dynamic environments. In this report, we have attempted to replicate the work completed by Kahn et al. [1] and focus on safe and robust collision avoidance. Our goal is to investigate if navigation policies can be learned in a self-supervised manner from trial-and-error, without relying on an explicit map of the environment.

II. Related Work

Many image-based autonomous controllers for uncrewed ground vehicles (UGV) have been developed over the last few decades [2]. Manderson et al. [3] categorize research on these controllers into techniques relying on imitation learning, semantic segmentation, air assistance, and sensor fusion. Recent work by [1], [3] focused on developing a solution that eliminated the reliance on maps that many image-based controllers have. Their techniques aimed to reduce computation complexity that can tax size, weight, and power (SWaP) constraints on increasingly smaller UGV and uncrewed aerial vehicle (UAV) platforms. Both groups deploy techniques built on the prior work of [4] to use a hybrid of model-free and model-based reinforcement learning to acquire an autonomous navigation policy. The researchers leverage a generalized computation graph (GCG) comprised of a convolutional neural network (CNN) to preprocess input states for a long short-term memory (LSTM) recurrent neural network (RNN). The latter group extends the technique by incorporating data from a UAV whose aerial images are from a view that is both orthogonal to and at a different scale than the UGV's images. The approach functions well in complex environments when compared to other reinforcement learning techniques. Our course project tackles the same problem as Khan et al. [1] while drawing inspiration from the implementation of [3]. We benefit from some of the observations and analysis from the latter work and deviate from previous implementations by using tools that we wish to implement in our own future research.

III. Preliminaries

A. Reinforcement Learning

Reinforcement learning problems are comprised of an agent that interacts with its environment. The system can be formalised by considering a state space for the agent, $\mathcal{S}$, as well as an action space, $\mathcal{A}$, which is a set of actions that can be taken at each state. The environment is then characterized by transition dynamics, $\mathcal{T}(s, a)$, that dictate how the agent transitions between states based on the action taken. A reward function, $\mathcal{R}$, is also defined to indicate how desirable the action taken was for the agent.

The ultimate goal of these reinforcement learning strategies is to maximize the reward the agent accrues over time and over successive attempts of solving the problem. Typically the methods of doing so fall into two categories, either model-free learning or model-based learning. The goal of model-free learning is to determine a strategy or policy, $\pi(s_t)$, which maximizes the reward based on previous observations. However, little to nothing is learned about the environment itself. Model-based learning is on the other end of the spectrum where it learns a model of the system which presents itself in the form of the transition probabilities $\mathcal{T}$. There also exist hybrid methods between these two methods which include the structure implemented in this report.

B. Generalized Computation Graphs

Generalized computation graphs are a more general approach to defining the transition function of the reinforcement learning process that encompasses both model-free and model-based learning. Depending on the implementation, a GCG could act identically to either method or a hybrid approach that leverages aspects of both.

The basic components of a GCG, $G_\theta(s_t^*, A_t^H)$, are a time horizon H, an input state, s_t^*, a sequence of H input actions, A_t^H for each time step t, a sequence of H outputs $\hat{Y}_t^H$ outputs for each time step t, and a predicted terminal output $\hat{b}_t^H$. There is also an error function $\mathcal{E}_t(\theta)$ which is used to calculate the error between the outputs $\hat{Y}_t^H$, $\hat{b}_t^H$ and corresponding labels for the data Y_t^H, b_t^H. This error signal is used to tune and update the GCG parameters θ to improve its performance for the task at hand. Figure 1 shows a diagram for the general structure of a GCG.

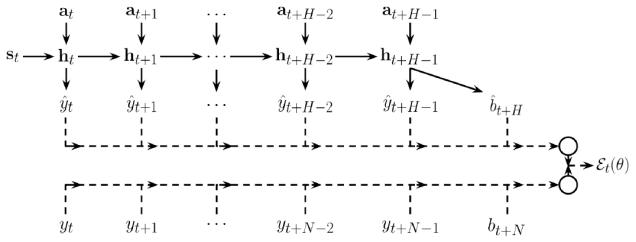

Fig. 1: A diagram highlighting the layout of a computation graph similar to the one implemented [1].

A GCG can be used in place of transition probabilities for reinforcement learning problems to produce outputs which can be used to determine the accrued reward. A reward function $J(s_t, A_t^H)$ is defined to determine these rewards using the output from the GCG. A policy, π, can be derived by looking at multiple action sequences from a given state s_t and greedily choosing the one which provides the highest reward as shown in Equation 1.

$$\pi(A_t^H|s_t) = \arg\max_{A^H} J(s_t, A_t^H) \tag{1}$$

The combination of the GCG, $\mathcal{E}(\theta)$, and $J(s_t, A_t^H)$ gives an algorithm structure that can be modified to suit the needs of a wide variety of problems. This generalized approach to learning, allows for a more robust and customizable implementation compared to many of the commonly used techniques in the field.

IV. Implementation

A. Representation

We have implemented a deep reinforcement algorithm to detect and avoid collisions as it navigates autonomously. The system attempts to predict its probability of collision at a given time and take actions to avoid future collisions. The model created is based on the works by Kahn et al. [1] and Manderson et al. [3] who successfully implemented similar systems to control vehicle navigation based on a GCG. The model has a prediction horizon of H time steps into the future for action generation. The model inputs are a state s_t consisting of an image of the vehicle's field of view and an action sequence of H discrete steering commands $(a_t, a_{t+1}, \ldots, a_{t+H})$. The output of the model is a vector $\hat{y}_t^H = (\hat{y}_t, \hat{y}_{t+1}, \ldots, \hat{y}_{t+H})$ of collision probabilities indicating the likelihood of collision at each time step. These likelihoods of collision are then used in our reward function

Algorithm 1 Reinforcement Learning and Dataset Creation

Inputs: GCG $G_\theta(s_t^*, A_t^H)$, action space $\mathcal{A}$, horizon value H, K-shooting value K

for Number of episodes **do**
  initialize empty dataset $\mathcal{D}$
    $\mathcal{D} \leftarrow \emptyset$
  get initial snapshot from simulator
    $s_0 \leftarrow$ Simulator
    for Max steps **do**
      initialize rewards list and action set list
        $Rewards \leftarrow \emptyset$
        $ActionSets \leftarrow \emptyset$
      Create stack of the past 4 states
        $s_t^* \leftarrow preprocess(s_{t-3}, s_{t-2}, s_{t-1}, s_t)$
      perform K-shooting
        for K **do**
          $A_t^H \leftarrow generateActionSequence(\mathcal{A}, H)$
          Append A_t^H to $ActionSets$
          $\hat{y}_t^H \leftarrow G_\theta(s_t^*, A_t^H)$
          $R \leftarrow J(\hat{y}_t^H)$
          Append R to $Rewards$
        end for
      get the best action set based on rewards calculated
        $R^+ \leftarrow \arg\max Rewards$
        $A_t^{H*} \leftarrow ActionSets[R^+]$
      Execute the first action in the simulator
        Simulator $\leftarrow A_t^{H*}[0]$
        $s_{t+1}, y_t \leftarrow$ Simulator
        Append $(s_t^*, A_t^{H*}, y_{labels})$ to $\mathcal{D}$
    end for
    train $G_\theta(s_t^*, A_t^H)$ on $\mathcal{D}$
end for

$J(\hat{y}_t^H)$ shown in Equation 2 to calculate the reward of each generated action sequence.

$$J(\hat{y}_t^H) = \sum_{h=0}^{H-1} -\hat{y}_{t+h} \tag{2}$$

The implemented model runs through an iterative process of collecting data from random trials in a simulation environment to generate a dataset. Training on this generated data is then performed to update the GCG. This process begins by getting an initial state from the simulator, K-shooting is then performed where K action sequences are generated and run through the GCG. The reward of each sequence is calculated and the best action sequence is chosen. The first action of this sequence is executed in the simulator and a collision label as well as a new image for the next iteration is gathered. This process is then repeated until an episode ends which occurs when the vehicle crashes or runs for a maximum number of steps without crashing. The maximum number of steps is calculated to equate to approximately 10 minutes of driving. Training on the generated dataset from the episode is then performed before beginning another episode. The algorithm architecture is shown in Algorithm 1.

The model is able to generate its own dataset for training where each entry is of the form $(s_t^*, A_t^H, (y_t, y_{t+1}, \ldots, y_{t+H}))$, consisting of a preprocessed set of images pulled from the simulator, the lowest cost action sequence, and the collision labels for the next H time steps. The collision labels are collected from the simulator at each time step and concatenated to form the label vectors. For time steps after a collision, a value of 1 is entered to indicate that a collision has already occurred.

B. Simulation Environment and Computing Resources

Kahn et al. [1] implemented their experiments in a Panda3D environment. This open source framework allowed this team to render a 3D instance of a cluttered and uncluttered hallway. Their remote controlled car was modelled separately using a Bullet physics simulator. We considered implementing our project in this environment based on this team's github repository (https://github.com/gkahn13/gcg), but it became evident that this older implementation required outdated dependencies and libraries that are not, currently, well documented. Manderson et al. [3] deployed their simulated experiments in the Unreal Engine, however they have not made their code publicly available. Our team did reach out to Travis Manderson to discuss the Unreal Engine, but time limitations meant that we had to progress our simulations before it was feasible to meet. Therefore, we looked across the suite of simulation environments that were being deployed in the autonomous vehicle research domain to hone in on an environment that would be both beneficial for this project and our future research activities.

We landed on the CARLA Simulator (version 0.9.13) [5] which is based on the Unreal Engine and focuses on providing open source tools to facilitate autonomous vehicle research. Specifically, our group leveraged a pre-built map, vehicle, collision sensor, and RGB camera for our implementation. The CARLA Simulator uses a server client model as shown in Figure 2. Our Python scripts contained all of the code necessary to implement our project and interacted directly with an instance of the CARLA server using a provided API. We configured the server to interact synchronously with our computationally expensive algorithms to control how simulation time progressed.

The simulator implementation is a result of numerous design decisions related to the problem formulation, experiments, computation resources and the simulator itself. The inputs to the simulator are a set of discrete steering actions {*left, center, right*} and a constant velocity. Our implementation overrode the standard PythonAPI throttle control input by enabling a constant velocity parameter. This eliminated some of the potentially unwanted acceleration effects that we observed in real world experiment videos by other researchers. The constant velocity parameter decomposes the input velocity along an orthogonal coordinate system. It is possible for the modelled vehicle's horizontal speed to be slightly lower than desired when the vehicle encounters obstacles in the simula-

Fig. 2: CARLA server view (left), vehicle view (right)

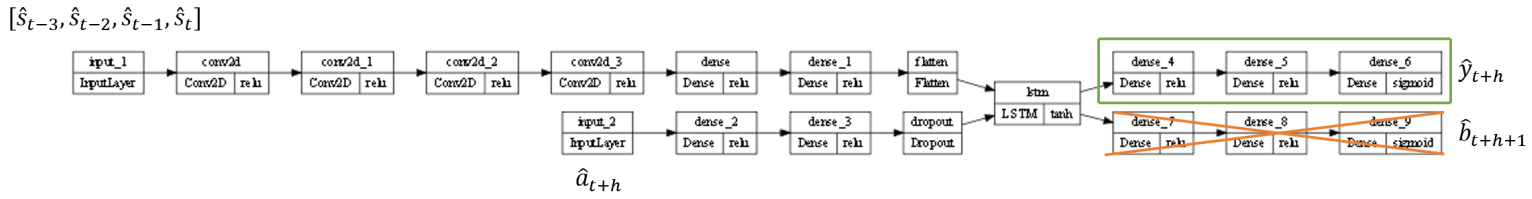

Fig. 3: The general computation graph that is used to learn collision-free robot navigation. The network uses a sequence of H actions and the past four grayscale images as inputs and outputs the probability of collision at the next time step.

tion environment. The simulation environment is configured to provide a single 72x128 pixel grayscale image (s_t, our state) and collision label ($y_t \in \{0$ - No Collision, 1 - Collision$\}$).

The CARLA Simulator has several system requirements, including an adequate GPU. Individually, our team only had access to laptops containing builtin GPUs with limited VRAM. The simulation environment, alone. was enough to consume all available computation resources on these laptops. Therefore, the entire project was moved to g4dn.2xlarge Amazon EC2 instance that could support both the simulation and the machine learning.

C. Network Architecture

To initialize the computation graph for robot navigation, we have implemented a RNN for the task of collision avoidance over a short predictive horizon. The neural network is used as a nonlinear function approximation technique to provide a mapping from states and actions to the probabilities of collision for each time step as shown in Figure 3. The network combines features from both the architectures implemented in Kahn et al. [1] and Manderson et al. [3].

The vehicles state is represented as a 72x128x4 array which was built by stacking four images together (an image from the current time step and those from three previous time steps). We decided to use the image size presented in [3] as it offered a greater spatial resolution to better encapsulate the vehicle's state compared to the smaller images used in [1]. These images were passed through a CNN to form the initial hidden state of the RNN. The CNN architecture implemented was developed based on [3] which uses four convolutional layers consisting of (32, 64, 64, 64), a kernel size of (8, 4, 4, 3) and strides of (4, 2, 2, 1) as well as two fully connected layers with filter sizes of (256, 128). This architecture gave us the ability to to produce the same size feature map as [1].The RNN also takes as input a sequence of H actions which are passed through two fully connected layers with filter sizes of (16, 16). The recurrent unit in the model comprises H LSTM cells. This was used in [3] and was simpler to implement compared to the Multiplicative Integration LSTM cell that was used in [1]. The H hidden states of the LSTM were passed through two fully connected layers with filter sizes of (16, 16) and passed through an output layer to produce probabilities of collision at each time step $\hat{y}_{t+h}$. The Rectified Linear Unit (ReLU) activation function was used for every layer except for the output where the Sigmoid activation function was selected. The model presented in [1] also produced an additional terminal output to represent the probability of future collision $\hat{b}_{t+H}$. We decided to simplify the model for our implementation and remove this parameter because we perceived that an implementation without this was analogous to some evolutionary techniques. Additionally, this output was not used in our implementation because we believed the probability of future collision was a way to bootstrap our estimate. The work presented in [1] mentioned that for models outputting collision probabilities, it can be advantageous to eliminate bootstrapping.

In order to train the model using the dataset that was generated, the Binary Cross Entropy Loss function shown in Equation 3 was used. This function was selected because we are dealing with a binary classification problem as the ground truth labels y_{t+h} are binary values that represent if a collision occurred at each time step. The error between these values and the output of the RNN $\hat{y}_{t+h}$ were calculated and used to train the model. The loss function was optimized using the Adam optimization algorithm which was initialized with a learning rate $\alpha = 0.001$ and momentum parameter $\beta_1 = 0.5$.

$$\mathcal{E}_t(\theta) = -[\sum_{h=0}^{H-1} y_{t+h} \log(\hat{y}_{t+h}) + (1 - y_{t+h}) \log(\hat{y}_{t+h})] \quad (3)$$

V. Experiments

To demonstrate the efficacy of the proposed model for collision avoidance, a base case and two experiments were performed and are summarized in Table I. These experiments involve sequentially adjusting system parameters with respect to the base case to identify the optimal network architecture for robot navigation. The network for each experiment was trained anew. Weights were initialized using a uniform distribution and biases initialized at zero.

The base case was developed using the recommended system parameters and design decisions from [1] which included a long model horizon ($H = 16$) and a discrete action space consisting solely of percentages of the maximum steering angle. These were selected as follows $\{$*-20%, 0, +20%*$\}$ because the actual steering angles were not provided in [1].

Two additional experiments were conducted each with an adjustment to the base case. For the first experiment the model horizon H was set to 8 to see if this had any impact on training time and because multiple horizons were tested across both [1] and [3]. The second experiment involved adjusting the action space to use the following action values $\{$*-30%, 0, +30%*$\}$ to

TABLE I: Experiment list

	Base Case	Experiment 1	Experiment 2
Model Horizon	16	8	16
Actions	{-20%, 0, +20%}	{-20%, 0, +20%}	{-30%, 0, +30%}

ensure a sufficient turning radius to allow for collision-free navigation.

VI. Results, Analysis and Discussion

Our current implementation does not improve its duration of collision free navigation with increased exposure to the environment. The slope of the results for the base case and both experiments in Figure 4 are constant indicating that our implementation was not learning. Our group spent considerable time reflecting on our initial results and believe that we misunderstood the role that $\hat{b}_{t+H}$ served in the algorithm. We perceived that the implementation would still learn as the K-shooting was calculating rewards for individual steps based on a finite horizon of future actions. However, it is now understood that K-shooting acts in an evolutionary manner by simply evaluating the set of actions (without regard to what they were), to determine the reward for a single action. Therefore, we believe that by eliminating $\hat{b}_{t+H}$ we've removed the agent's ability to weight short term rewards against longer term gains. Unfortunately, due to the complexity of implementation and significant training time we were unable to add the $\hat{b}_{t+H}$ functionality to our algorithm prior to report submission. We elected to run experiments to test the parameters of our current implementation.

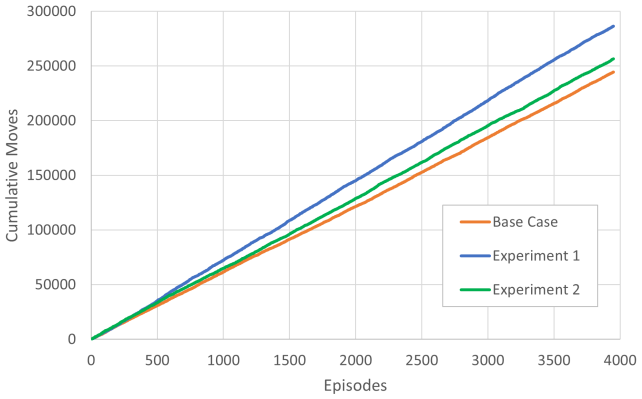

Fig. 4: Experimental Results

It is evident that the selected experiment parameters in Table I do impact the duration of collision free navigation. Adjusting the horizon parameter downward (as was done in Experiment 1) appears to provide more immediate reward as, on average, the network is less likely to predict a collision over a shorter horizon. In the absence of any discounted future rewards the agent is less likely to alter the current path as it is being rewarded for the current action. Conversely, over longer horizons the network is more likely to predict a collision and generate a lower reward. This signals to the agent that a different action is needed to avoid collisions.

The base case action assignment for steering angles appear to be limiting the vehicle's ability to make rapid changes to its trajectory, when compared to Experiment 2. This may prevent the vehicle from making turns on time or from correcting its path when it is estimated that a collision is forthcoming.

We initially produced a figure showing the number of moves per episode for each experiment. However, this graph did not have any apparent slope, because no learning occurred. However, it was observed that there were striations within the data. Figure 5 shows this data as a histogram where the structure is more easily observed. The clustering of moves within specific bins is likely associated with how the agent is re-spawned after a collision. We used CARLA map Town02 and its associated pre-defined spawn locations. Reflecting on this result it is apparent that the design decision to use a subset of seven spawn points imparted some bias into the data.

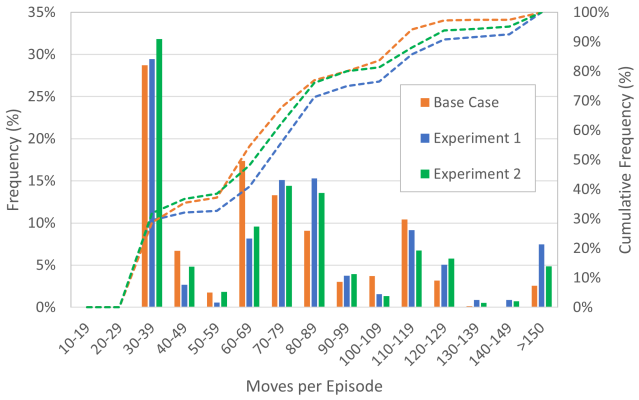

Fig. 5: Experimental Results - Frequency Distribution

The results do not necessarily provide insight into the impact of some design decisions or deviations from the original papers. Specifically, as mentioned in the Network Architecture section we used a hybrid of [1], [3] approaches in developing our CNN. It is unclear what impact, if any, this decision had on our system outputs. Similarly, our team designed the system to travel a distance between actions of 0.5 meters that was equivalent to that described in [1]. However, CARLA Simulator produced warnings about inaccurate simulation results anytime we set our time step greater than 0.1 seconds. Therefore, we adjusted our velocity to ensure that the same distance was travelled at each simulation step. The effect that these design decisions had on the implementation will need further exploration when a correct implementation of the discounted future rewards is implemented as speed, distance travelled per step and steering angle all impose limitations on the agent's ability to act within the environment. Similarly, design decisions about the image resolution, colour map and network architecture impose limitations on how the state of the environment is represented.

VII. Conclusions and Future Work

A. Conclusions

For this project, we have attempted to replicate the work completed by Kahn et al. [1] and have drawn inspiration from Manderson et al. [3] to determine if collision-free navigation policies can be learned in a self-supervised manner from trial-and-error. Several experiments were conducted to better understand the effects of model horizon and action space on developing an optimal network architecture for this task. Based on the results conducted between several experiments, it is clear our model is unable to learn a collision-free navigation policy. We believe this outcome occurred based on the decision to eliminate the terminal output $\hat{b}_{t+H}$ from the model, which removes the ability to weight actions beyond the model horizon. Although we were unable to learn an optimal policy, we do believe the experiments performed offer insight into further improvements to the model that could produce promising results in future implementations.

B. Future Work

Due to the time constraints for the term project and the results presented, there remain many improvements and extensions that can be completed to develop a successful navigation policy and extend this research beyond collision avoidance. The first improvement is to implement the additional output which represents the probability of future collision $\hat{b}_{t+h}$. This would slightly modify the network architecture by taking the final hidden state from the LSTM cell and feeding this through fully connected layers to produce the terminal output $\hat{b}_{t+H}$. If this modification promotes learning of successful navigation policies, additional experiments could be conducted to determine the optimal network architecture for collision-free navigation. To conduct additional experiments, the CARLA Simulator would need to be transferred to Queen's University GPU clusters as Amazon EC2 is a paid service. Once the optimal model is developed, field testing could be conducted for validation. Finally, adding an aerial agent that provides overhead images to the model, similar to [3], could offer greater planning foresight and make the system more robust. This would allow alternative navigation policies to be explored including finding the optimal path that exploits aerial obstructed views.

VIII. Effort

Our team members have all contributed equally to this project. In fact, all researchers are deeply devoted to a successful implementation and have committed to progressing some of the future work recommendations beyond the academic term. It is estimated that each of us has spent at least 85 hours dedicated to working on this project.

Acknowledgment

The authors would like to thank Queen's University - Electrical and Computer Engineering undergraduate students Brent Champion and Raed Fayad. They collaborated with us to deploy the CARLA Simulator on an Amazon EC2 instance after all previous attempts by us to deploy the simulation on Queen's University GPU clusters failed. Their collaboration allowed us to progress our project without getting held up in learning the intricacies of implementation on remote servers.

References

[1] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine, "Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 5129–5136.

[2] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 1. Morgan-Kaufmann, 1988. [Online]. Available: https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf

[3] T. Manderson, S. Wapnick, D. Meger, and G. Dudek, "Learning to drive off road on smooth terrain in unstructured environments using an on-board camera and sparse aerial images," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 1263–1269.

[4] J. Oh, S. Singh, and H. Lee, "Value prediction network," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/ffbd6cbb019a1413183c8d08f2929307-Paper.pdf

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.