Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Zoltán Lőrincz

IMITATION LEARNING IN THE DUCKIETOWN ENVIRONMENT

SUPERVISORS

Márton Szemenyei (BME IIT)

Róbert Moni (BME TMIT)

BUDAPEST, 2020

Contents

Kivonat

Napjainkban egyre nagyobb figyelem irányul az önvezető autókra és a vezetést támogató rendszerekre mind az ipar, mind a kutatás területéről. Ennek következtében egyre több jármű tartalmaz modern biztonsági funkciókat, ezáltal a közlekedés is egyre biztonságosabbá és hatékonyabbá válik. A nagymértékben megnövekedett számítási kapacitásnak, illetve a gépi tanulás területén elért kutatási eredményeknek köszönhetően a korszerű vezetést támogató megoldások többsége mély tanuláson alapul.

A mély tanulás egyik alterülete az imitációs tanulás, mely során a tanuló algoritmus az adott feladat megoldásához szükséges optimális stratégiát egy szakértő demonstrációin keresztül tanulja meg, ezáltal tulajdonképpen utánozza, "imitálja" a szakértő viselkedését. A módszer sajátossága, hogy a tanításhoz nem felcímkézett adatokra, hanem a feladat elvégzése során készített demonstrációkra van szükség. Ezek generálása bizonyos problémák esetén sokkal egyszerűbb és kevésbé költséges, mint a tanító adatok felcímkézése. Ez különösen igaz az önvezető autókra, ahol a szükséges demonstrációk összegyűjthetők közönséges vezetéssel is.

Munkám során a Duckietown környezet szimulátorában vizsgáltam meg különböző, imitációs tanuláson alapuló algoritmusokat. Az említett környezet egy nyílt forráskódú platform, melyben különféle önvezető feladatokat oldhatunk meg. Az ágensek kamerával felszerelt differenciális hajtású robotok, melyek a miniatűr városokban közlekednek. Az általam megvalósított feladat a sávkövetés volt.

Dolgozatomban először bemutatom a szakirodalmat: az imitációs tanulás különböző fajtáit és az egyes módszereket, algoritmusokat. Ezt követően ismertetem a Duckietown környezetet, az egyes feladatokat és a kiértékeléshez szükséges metrikákat. Ezután bemutatom az általam felhasznált imitációs tanuláson alapuló algoritmusokat, ezek implementálásának részleteit, a felhasznált modelleket, a tanítás menetét és az egyes algoritmusokkal elért eredményeket. Végül összehasonlítom az egyes algoritmusokat az általuk elért eredmények alapján.

Abstract

Nowadays autonomous vehicles and advanced driver-assistance systems gain more and more focus from both the industry and researchers. As a result, the number of vehicles equipped with such systems is growing quickly, and therefore road safety and traffic efficiency is greatly increasing. Due to the rapid advancements in Machine Learning and computational power, most of the modern driver-assistance solutions are Machine Learning-based approaches.

Imitation Learning is a subfield of Deep Learning, where the learning algorithm uses demonstrations of an expert to uncover an optimal policy for the given task. This approach does not require a labelled dataset for training. Instead, it uses demonstrations of the given objective. Consequently, Imitation Learning can be an advantageous solution in case of problems where collecting demonstrations is easier and cheaper than creating a labelled dataset. Therefore, this method could be applied for autonomous driving as well, since the required demonstrations could be easily collected by casual driving.

During my work I experimented with different Imitation Learning algorithms in the simulator of the Duckietown environment to solve a lane following task. Duckietown is an open-source platform, which can be used to solve different self-driving tasks. In this environment the agents are differential wheeled robots that are equipped with a monocular camera. They operate in small, miniature cities.

In this paper, I first introduce the different Imitation Learning approaches and algorithms. Next, I present the Duckietown environment, the self-driving tasks and the evaluation metrics. In the next part, I describe the theory and the implementation details of the algorithms I applied to solve the lane following task. The training process, the used models and the achieved results are presented afterwards. At the end of the paper, I summarize the different algorithms based on the results.

1 Introduction

Each day information and computer technology are becoming more and more advanced and this can be observed in the automotive industry as well. Due to the increasing computational power, vehicles are being equipped with cutting-edge safety and convenience features, therefore they are becoming smarter, safer and more advanced. As a result, the development of advanced driver-assistance systems is a key area in automotive research. It is just a matter of time until fully automated driving is achieved. This would be quite satisfactory, as self-driving vehicles would bring several improvements in transportation technology. Road safety and traffic efficiency would significantly improve, and most likely the air pollution would be reduced as well.

The rapid advancement in this field lately has been caused by the huge progress in computer vision, Machine Learning and Deep Learning methods. These techniques allow us to solve complicated real-world problems that we could never solve with the previous, traditional approaches. Despite of this, the problem of autonomous driving is still an extremely complex task that requires further research in this topic. One possible way to tackle this problem could be with the help of Imitation Learning. This Machine Learning approach uses the demonstrations of an expert to learn its optimal policy, which allows the agent to “imitate” the expert, to behave in the environment just like him. Imitation Learning has been a popular approach in the research of self-driving cars, especially in the recent years. The main reason for this is the fact that this method does not require labeled data, all it needs are expert demonstrations, which are easy to acquire through normal, casual driving.

In this work, a study has been carried out on the state-of-the-art Imitation Learning approaches. After this study I have implemented different Imitation Learning techniques to solve a self-driving task: the right-lane following challenge in the Duckietown [1] environment. I used the solutions with the best performance to compete in the AI Driving Olympics [2] contest and managed to achieve decent results.

The structure of this work is the following: Chapter 2 provides the theoretical background for neural networks and Imitation Learning methods. Chapter 3 presents the previously mentioned lane following task and the Duckietown platform, which has been the main environment of the experiments. Chapter 4 presents the design, the experimental work and different solutions in detail. Chapter 5 summarizes the results of the experiments. Chapter 6 gives a general conclusion about the thesis’ work.

2 Background

This chapter provides the theoretical background for this work. It focuses on Deep Learning and Imitation Learning techniques.

2.1 Deep Learning

Deep Learning is a subset of Machine Learning, where we use a deep neural network (a neural network with a large number of layers) to solve the given problem. In this section I will present the different techniques and methods related to Machine Learning and Deep Learning.

2.1.1 Machine Learning

Machine Learning is a subset of Artificial Intelligence. It is a data driven approach: the models are not explicitly programmed, instead, they use training data to learn how to perform the given task. During the training phase, the models improve by learning from previously committed mistakes.

There are three main types of Machine Learning techniques:

- **Supervised Learning:** This approach uses labelled training data: each input value has a corresponding output value, which is also known as ground truth data. The method trains the algorithm to produce the desired output of a given input by minimizing the error between the predicted and the predefined output values.
- **Unsupervised Learning:** In the case of Unsupervised Learning, the training data is not labelled. Instead of learning based on the ground truth values, the algorithm tries to learn a hidden, underlying structure of the data.
- **Reinforcement Learning:** In Reinforcement Learning an agent (which serves as the learning algorithm) interacts with an environment by following a policy. In each state of the environment, it takes an action based on the policy, and as a result, receives a reward and transitions to a new state. The goal of the agent is to learn an optimal policy which maximizes the long-term cumulative rewards.

2.1.2 Supervised Learning

The general Supervised Learning algorithms can be described with the following equation:

$$y = f(x, \vartheta)$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are the inputs and the outputs of the algorithm respectively, while $\boldsymbol{\vartheta}$ refers to the trainable parameters, which are responsible for the algorithm's operation. We also define a loss function and an optimizer. The loss function measures the performance of the model on the given task. The optimizer determines how to change the trainable parameters to achieve better results. During the training, the parameters are updated using the optimizer to minimize the loss function. The algorithm also has hyperparameters, which cannot be optimized using the common optimization methods.

2.1.3 Neural Networks, Neuron

Neural Networks have been inspired by modeling biological neural systems. The basic building block of a Neural Network is the neuron. A neuron implements a simple mapping: a non-linear function is applied to a single matrix multiplication:

$$y = f(Wx + b)$$

where $\boldsymbol{x}$ is the input, $\boldsymbol{y}$ is the output, $\boldsymbol{W}$ is the weight matrix (or parameters), $\boldsymbol{b}$ is the bias vector (because it influences the output, but without interacting with the actual data) and $\boldsymbol{f}$ is the non-linear activation function. Neural networks consist of layers, which are created by connecting multiple neurons.

The simplest layer type is the **fully-connected layer**, in which neurons between adjacent layers are connected, but neurons inside the same layer are not. A neural network that consist of only fully-connected layers is called the multi-layer perceptron (see Figure 1).

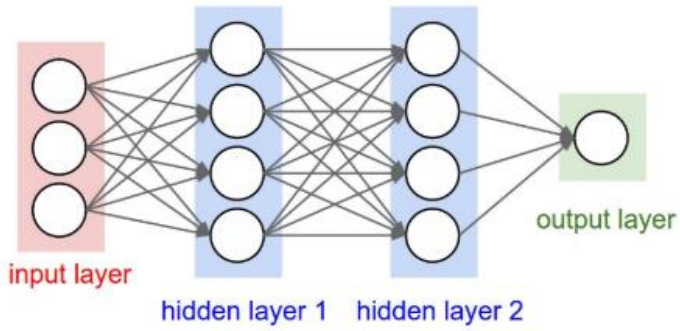

Figure 1: The multi-layer perceptron. Source: [31]

The first layer of a neural network is called the input layer, the last layer is the output layer, and all the other layers are called hidden layers.

2.1.3.1 Activation functions

Activation functions introduce nonlinearities into the model. This is needed because the goal of a neural network is to model a complex nonlinear function. There are several types of activation functions and selecting the right one is quite important for the model to work efficiently.

In the past, **sigmoid** and **tanh** (hyperbolic tangent) were the most commonly used activation functions. However, both of these functions have a drawback: the derivative (gradient) is close to zero for the most part of the function's range. The small gradient values in these regions can cause the famous vanishing gradient problem: as we propagate backward, the gradients of the activations become smaller and smaller. As a result, the first few layers of the network are not updated at all and the network fails to learn. This is particularly true for very deep neural networks.

The **Rectified Linear Unit (ReLU) [3]**, which is currently the most popular activation function, mitigates this problem. The ReLU function is the following: $f(x) = \max(0, x)$. The derivatives of this function are zero in the negative and one in the positive region. An improved version of the ReLU is the **Leaky ReLU [5]**. Instead of the function being zero in the negative region, a Leaky ReLU has a small negative slope, which can be set with a hyperparameter. Another version of this activation function is the **PReLU [4]** (Parametric ReLU), where the steepness of the negative slope is not a hyperparameter, but a trainable parameter that the network can learn.

The sigmoid, tanh and ReLU activation functions are demonstrated by Figure 2.

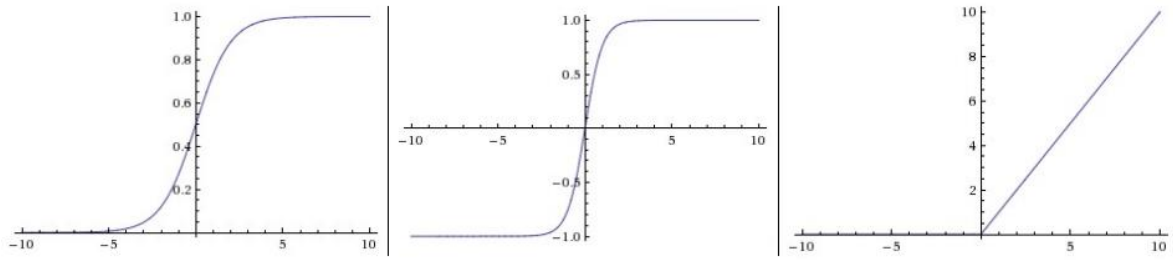

Figure 2: Sigmoid (a), Tanh (b) and ReLU (c) activation functions

In my work I have used the ReLU and the Leaky ReLU activation functions.

2.1.3.2 Loss functions

The goal of the loss function is to provide feedback on how well the model is performing. The typical loss function compares the neural network's output value with the ground truth data and computes a value that measures the model's performance. In an ideal scenario (where the model has been trained perfectly) this value would be zero, if the model is performing poorly, this value is high.

The appropriate loss function mainly depends on the task of the model. Generally, there are two main Deep Learning problems: regression and classification. In case of regression, the output of the network is a real-valued vector. For this task, it is common to use **mean absolute error** (L1 norm) or **mean squared error** (L2 norm). Classification is the task of identifying which of a set of categories the input of the model belongs to. The two most commonly used loss functions in this setting is the **Multiclass Support Vector Machine (SVM)** loss (or hinge loss) and the **Cross-entropy** loss (or Softmax loss).

The mean absolute error and mean squared error loss functions measure the predicted output value's L1 and L2 distance from the correct, ground truth value. The formulas of these loss functions are the following:

$$L_i = \sum_j \left| y_{j_{corr}} - y_{j_{pred}} \right| \qquad L_i = \sum_j (y_{j_{corr}} - y_{j_{pred}})^2$$

where $y_{j_{corr}}$ is the j-th correct (ground truth) value, $y_{j_{pred}}$ is the j-th predicted value.

In case of the SVM loss, the output values of the model are scores. If the score of the correct class is higher than the score of the incorrect classes by a fixed margin Δ, the error is zero. Otherwise the error is increasing linearly. The formula of the SVM loss is the following:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

where $\boldsymbol{s_{y_i}}$ is the score for the correct class, $\boldsymbol{s_j}$ is the score for the j-th class.

The Cross-entropy loss is used together with the Softmax classifier. This means that the output of the neural network is a softmax function $\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$: it takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one. This provides us with "probabilities" for each class (opposed to the SVM loss, where each class had a score). Together with the Cross-entropy loss, we are therefore minimizing the negative log likelihood of

the correct class, which can be interpreted as performing Maximum Likelihood Estimation. The complete loss function is formulated as (with the same notation as before):

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

2.1.3.3 Regularization

There is an issue with the previously presented loss function: if we have a set of parameters $\boldsymbol{W}$ that correctly classify every example, this set of $\boldsymbol{W}$ is not unique. For example, if we multiply $\boldsymbol{W}$ by a positive number (that is also bigger than 1), these parameters will also give zero loss. This issue might lead to numeric problems and an overconfident model.

To solve this problem, we extend the loss function with a **regularization penalty** $R(W)$, that only depends on the weights, not the data. The most common regularization penalties are the L1 and L2 regularization terms, but an elastic regularization is also widely used. These methods are formulated as following:

$$R_{L1} = \sum_k \sum_l |W_{k,l}| \qquad R_{L2} = \sum_k \sum_l W_{k,l}^2 \qquad R_{EL} = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

The complete loss function that contains the regularization term as well is the following:

$$L = \frac{1}{N}\sum_i L_i + \lambda R(W)$$

where $\boldsymbol{N}$ is the number of training examples, $\boldsymbol{L_i}$ is the data loss for the i-th example and $\boldsymbol{R(W)}$ is the regularization weighted by a hyperparameter $\boldsymbol{\lambda}$.

2.1.3.4 Optimization

Throughout the training phase, the neural network's weights are updated using optimization methods to minimize the loss function. In the case of neural networks, the most commonly used optimization algorithms are gradient-based methods. The way these methods work is quite simple. First, we compute the gradient of the loss function with respect to the weights (which means we differentiate the function). After this, we use the calculated gradients to update the network's parameters. If we update the weights along the gradient's direction, we "take a step" towards the loss function's local maximum, therefore we maximize it – this is called **Gradient Ascent**. Updating the parameters along the negative gradient direction minimizes the loss function – this is called **Gradient Descent**. The gradient descent's formula is the following:

$$W_{k+1} = W_k - \alpha \frac{\partial \|E\|^2}{\partial W}$$

where $\boldsymbol{\alpha}$ is the **learning rate** (or step size). This is one of the most important hyperparameters.

In practice, however, we usually have large-scale applications where the training data can have millions of examples. Therefore, computing the full loss function over the entire training set at each time we perform parameter update would be extremely time-consuming. To solve this issue, a very common approach is to compute the gradient over batches of the training data and use this batch to perform a parameter update. This approach is called **Stochastic Gradient Descent (SGD)**. The batch size is yet another hyperparameter.

There are several ways to improve the performance of the gradient-based optimizer. One such way would be adding a **momentum** term to the update rule. This means that at each step we take the weighted average of the actual and the previous gradients and we perform the parameter update using this newly calculated gradient. As a result, the parameter vector will build up velocity in any direction that has consistent gradient. Using the momentum update makes the model parameters less likely to get stuck in a local minimum or a saddle point.

Another way to improve the SGD optimizer is to use an **adaptive learning rate**. During the optimization, we calculate the sum of squared gradients for every parameter. This is then used to normalize the parameter update step, element-wise. As a result, the weights that usually receive high gradients will have their effective learning rate reduced, while weights that receive small updates will have their effective learning rate increased.

In this work I have used the well-known and commonly used **Adam [6]** optimizer algorithm, which is an adaptive learning rate method, that uses a combination of momentum and moving average of squared gradients.

2.1.3.5 Backpropagation

To calculate the loss function's gradient with respect to the weights, we use a method called backpropagation. Backpropagation [7] is a way of computing gradients of expressions through recursive application of chain rule. This method interprets the neural network as a computational graph, where each node is an analytically differentiable function. If we know the input values of a node, we can simply calculate the output by applying the function to the inputs. This is called the **forward pass**. We can apply a **backward pass** to the outputs, which starts at the end of the graph and recursively applies the chain rule to compute the gradients all the way to the inputs of the graph. The backward pass is illustrated by Figure 3.

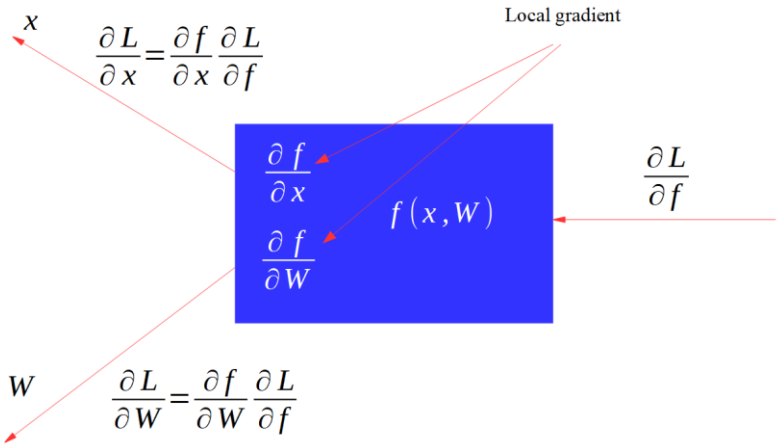

Figure 3: The backward pass. Source: [30]

Each iteration during training, we first perform a forward pass on the neural network: we take the input values, feed them through the network and compute the output. After this, we calculate the loss function using the predicted and the ground truth values. Then, we perform a backward pass: we backpropagate the loss trough the network and compute the gradients. This process is shown on Figure 4.

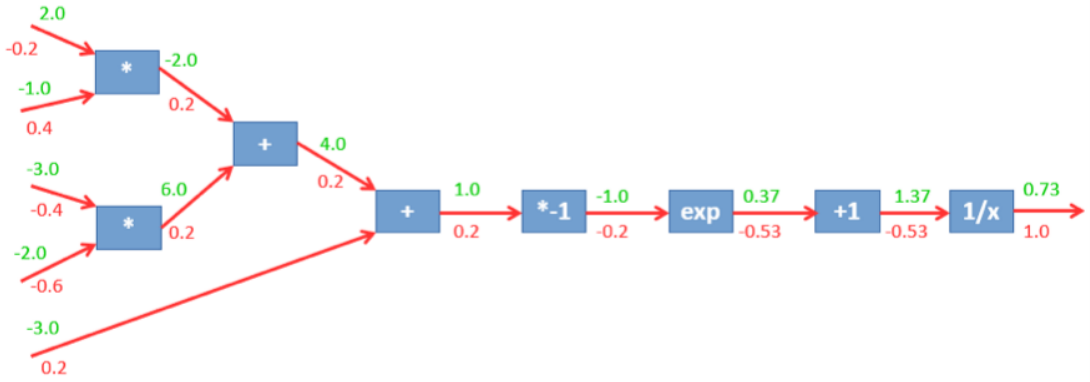

Figure 4: Doing backpropagation on a computational graph. The activations are green, the derivatives are red. Source: [30]

2.1.4 Difficulties related to training neural networks

The training of a neural network is a complex task, during which many difficulties can occur that would result in a poor performance. The most common problems are convergence issues such as vanishing or exploding gradients, or underfitting and overfitting, which are problems related to the model's and training set's complexity.

Exploding and vanishing gradients are numerical issues that prevent the convergence of the neural network's weights. During the forward and backward passes of a network, we perform

several matrix multiplications. If the norm of the weight (or gradient) matrices is too big (bigger than 1), the result will explode (will be close to infinity), which is the case of **exploding gradients**. The problem of the **vanishing gradients** is similar: the norm of the matrices is too small (close to 0) and therefore, the result will be close to zero as well. To prevent these convergence issues, we have to make sure that the norm of the matrices is close to 1. This can be achieved by choosing the right activation functions, initializing the weights of the network properly, normalizing the input data and using batch normalization.

The goal of a neural network is to generalize well from the training data to be able to make predictions in the future for data that the model has never seen before. To ensure this, it is critical to appropriately select the complexity of both the model and the training data, otherwise we can encounter issues such as **underfitting** or **overfitting**. In case of underfitting, the model is not complex enough to be able to learn from the training data and generalize from it. On the contrary, if overfitting occurs, the model is either too complex or the number of training examples is too low. As a result, instead of generalizing, the model memorizes the training data. Although it performs remarkably on the training set, it produces poor results on the test dataset. Figure 5 illustrates underfitting (left), overfitting (right) and good fitting (middle) on the training data.

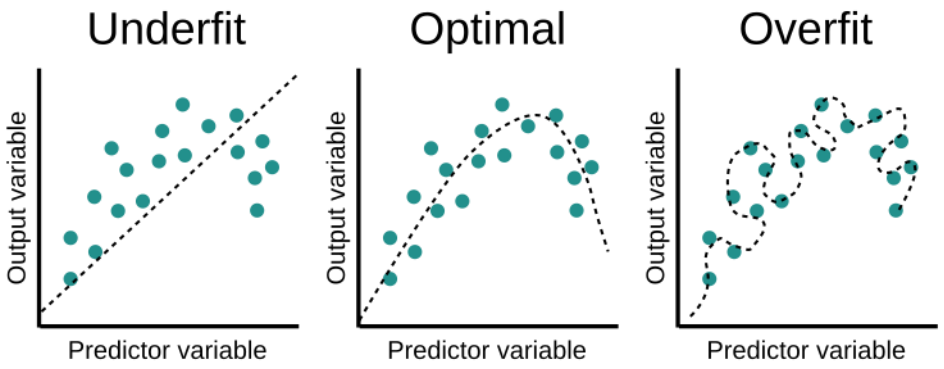

Figure 5: Underfitting (left), overfitting (right) and good, optimal fit (middle). Source: [32]

To prevent the underfitting and overfitting issues from happening, it is important to have a complex, variable dataset with a lot of examples. It is also advised to split the data into training, validation and test datasets. There is also a number of techniques that mitigate these issues, such as early stopping, dropout or batch normalization.

2.1.4.1 Learning rate

The most important hyperparameter of a deep learning application is the **learning rate**, which is the amount that the weights are adjusted during training. Setting the right value for the

learning rate is critical in the training process. A too small value might result in a long training process that could get stuck. In the case of a high learning rate, the training gets close to the optimum quite quickly, but cannot reach it as the step size is too big. Consequently, the weights remain sub-optimal. Furthermore, a very high learning rate causes drastic updates resulting in an unstable training process. This behavior is illustrated by Figure 6 (on the left).

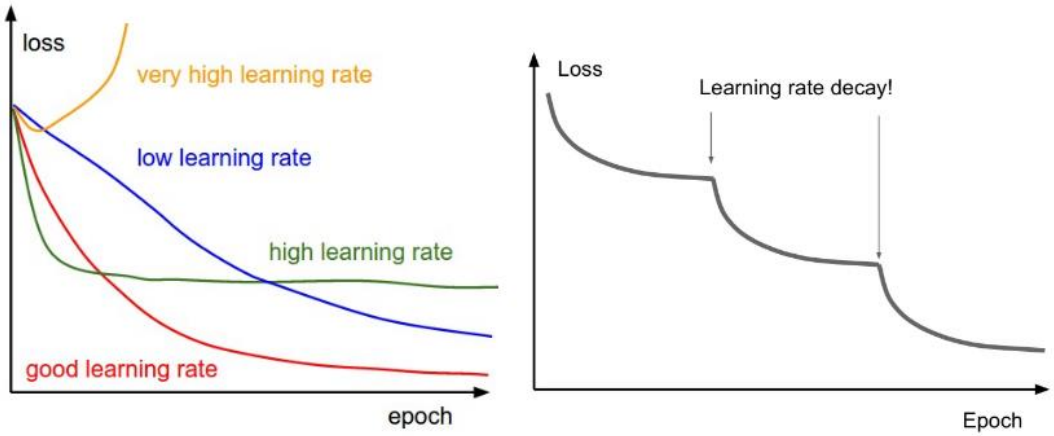

Figure 6: The effects of different learning rates (left) and Learning rate decay (right). Source: [30]

A common approach to use the proper learning rate during training is called **learning rate decay** (see Figure 6, on the right). This method reduces the learning rate by some factor every few epochs (or when the value of the loss function stops decreasing).

2.1.4.2 Weight initialization

As mentioned in section **2.1.4**, it is important to properly initialize the weights of the neural network in order to prevent convergence issues from happening. The two most common methods are the Xavier [8] and He [9] weight initialization formulas.

2.1.4.3 Early stopping

Early stopping is a regularization approach that is used to avoid overfitting. During training, each time the model's performance improves on the validation dataset, the model is saved. This method has a hyperparameter: **patience**, which is a number of training epochs. If the model's validation error stops improving for this number of epochs, we stop the training and load the network's weights from the last checkpoint. The method is demonstrated by Figure 7.

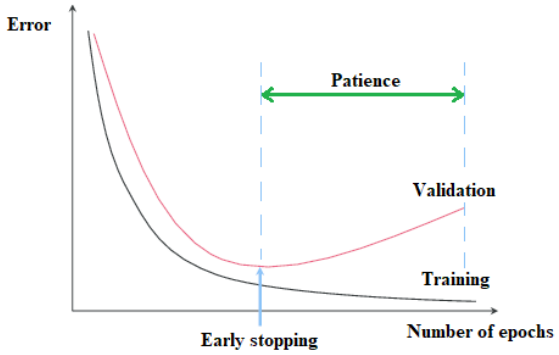

Figure 7: Early stopping and patience

2.1.4.4 Dropout

Dropout [10] is another regularization technique that is designed to eliminate overfitting. It is performed only in the training phase, during testing and inference this operation is not used. Dropout can be interpreted as sampling a neural network within the full neural network, and only updating the parameters of the sampled network based on the input data. This forces the network to generalize and learn in a redundant way. The method is illustrated by Figure 8.

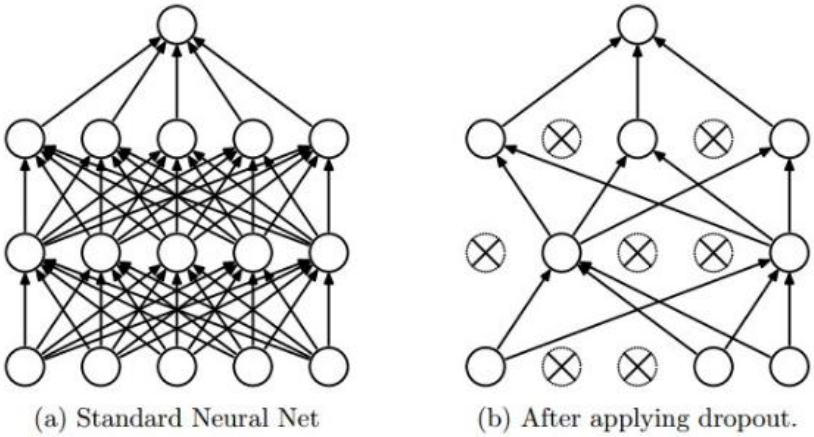

Figure 8: Dropout. Source: [10]

2.1.4.5 Batch normalization

Batch normalization [11] is a regularization technique, that not only reduces overfitting, but also mitigates the convergence problems related to bad weight initialization. The method explicitly forces the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. Batch normalization is formulated by the following:

$$x_{BN} = \frac{x - \mu}{\sigma^2 + \epsilon}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ is the expected value and the standard deviation of the $\boldsymbol{x}$ input, while $\boldsymbol{\epsilon}$ is a constant.

2.1.5 Convolutional Neural Networks

A convolutional neural network (CNN) is special type of neural networks, which is most commonly used for different computer vision problems. These networks are able to detect image features: low-level features in the first few layers and high-level features in the deeper layers.

The basic building block of a CNN is the **convolutional layer**, which parameters consist of a set of learnable **filters**. During the forward pass, we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. This produces a 2-dimensional activation map that gives the responses of that filter at every spatial position. This procedure is demonstrated by Figure 9.

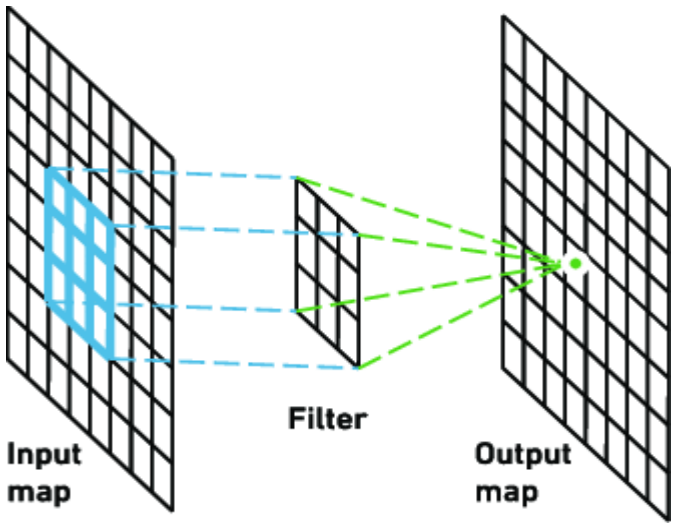

Figure 9: The forward pass of a convolutional layer. Source: [33]

The convolutional layer has three hyperparameters: **depth**, **stride** and **zero-padding**. Depth is the number of filters that the layer uses, the stride defines the displacement of the window, and the padding defines how the border of the input image should be filled.

In a convolutional neural network, it is common to periodically insert a **Pooling layer** in-between convolutional layers. Its function is to progressively downsample the volume spatially to reduce the number of parameters and computation in the network. The most common downsampling method is max pooling (see Figure 10).

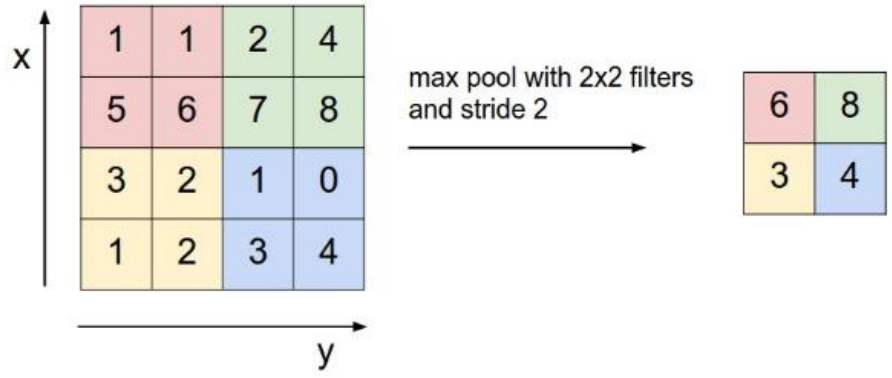

Figure 10: Max pooling operation. Source: [31]

2.1.5.1 ResNet architecture

There are several convolutional neural network architectures that are used today for different applications. One of the most commonly used architecture is the **Residual Network [12]**, which features special skip connections called **residual blocks** (see Figure 11). These blocks help to mitigate the vanishing gradient problem. The architecture uses a lot of batch normalization and it only has one fully connected layer at the end of the network (which outputs the class scores).

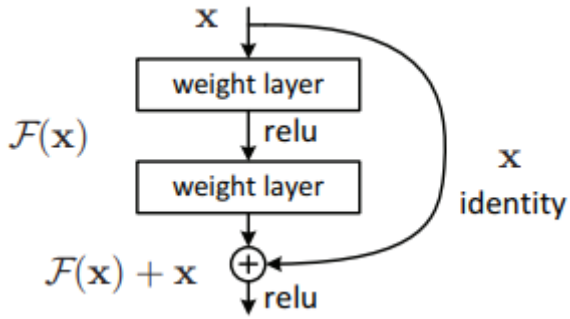

Figure 11: The residual block. Source: [12]

In this work, I have used the ResNet architecture as a feature extractor model.

2.1.6 Generative Adversarial Networks

Generative Adversarial Networks [13] (GANs) are today's state-of-the-art generative models. Overall, the aim of a generative model is to estimate or learn the training data's distribution to be able to later generate new data that appears to be sampled from the training distribution. In other words, the goal of a generative model is to generate data that is very similar to the training examples.

The GAN architecture (see Figure 12) consists of two neural networks: the **generator** and the **discriminator**. The generator's task is to produce new data, which is similar to the real, training

examples. It takes a sample from a given distribution (for example a random noise) and transforms it into the fake, generated data. Practically, the goal of the generator is to learn the mapping from the sample distribution to the real data distribution. The discriminator's objective is to differentiate the real and the fake (generated) data. It should output a value between 0 and 1: 1 for real and 0 for fake examples. Basically, the discriminator is a binary classifier that is trained with Supervised Learning.

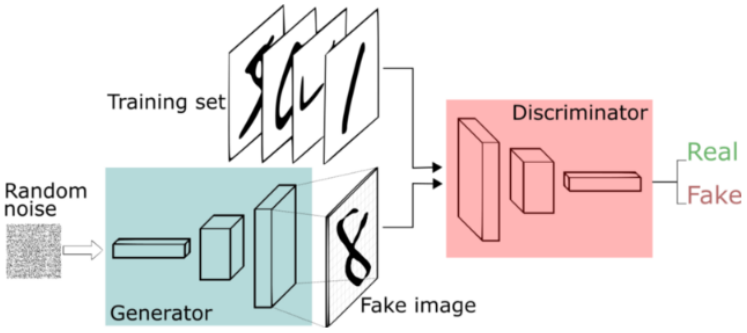

Figure 12: The GAN architecture. Source: [34]

Training GANs is basically a two-player minimax game between the two networks: the generator wants to minimize the game's objective, while the discriminator wants to maximize it. The game's objective is the following formula, which is also known as the **adversarial loss**:

$$\min_{\theta_g} \max_{\theta_d} \left[E_{x \sim p_{data}} \log D_{\theta_d}(x) + E_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

where $\boldsymbol{\theta_g}$ and $\boldsymbol{\theta_d}$ are the parameters (weights) of the generator and the discriminator, $\boldsymbol{z}$ is the sampled distribution, $\boldsymbol{D_{\theta_d}(x)}$ is the discriminator's output for real data and $\boldsymbol{D_{\theta_d}\left(G_{\theta_g}(z)\right)}$ is the discriminator's output for the generated $\boldsymbol{G_{\theta_g}(z)}$ data.

The interpretation of the objective is the following. The discriminator is trained to maximize the probability of assigning the correct binary label to both real and generated samples. The generator is trained to minimize the probability that the discriminator detects fakes. It essentially tries to fool the discriminator by generating real-like samples.

Discriminator and generator are trained alternately by gradient ascent and gradient descent. While one network is trained, other remains constant.

2.1.7 Reinforcement Learning

Reinforcement Learning is an interesting area of Machine Learning, where the learning algorithm is trained via interaction and feedback: an agent interacts with an environment, receives rewards for it and improves its decision-making process based on the rewards. The procedure is modeled as a **Markov Decision Process** (**MDP**), which consists of the following elements:

- $\boldsymbol{S}$: the set of possible states in the environment
- $\boldsymbol{A}$: the set of possible actions
- $\boldsymbol{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$: the transition model – the probability that an action $\boldsymbol{a}$ performed in the state $\boldsymbol{s}$ leads to state $\boldsymbol{s}'$
- $\boldsymbol{R}(\boldsymbol{s},\boldsymbol{a})$: the reward function – the value of the reward that the agent receives after executing the action $\boldsymbol{a}$ in the state $\boldsymbol{s}$
- $\boldsymbol{\pi}$: the agent's policy – the decision-making function (control strategy) of the agent, which represents a mapping from states to actions

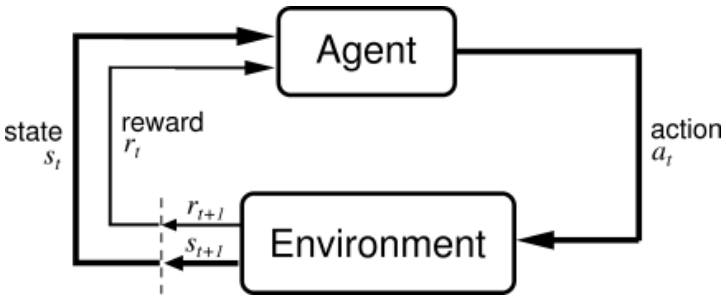

Figure 13: The Markov Decision Process

The MDP formulation used in Reinforcement Learning is demonstrated by Figure 13. At a given time step $\boldsymbol{t}$, the Reinforcement Learning agent is situated in state $\boldsymbol{s_t}$, which is observed by the agent trough observation $\boldsymbol{o_t}$. The agent selects an action $\boldsymbol{a_t}$ based on its policy $\boldsymbol{\pi_\theta}$ and performs it. Due to this, the agent moves to a new state $\boldsymbol{s_{t+1}}$ based on $\boldsymbol{P}(\boldsymbol{s_{t+1}}|\boldsymbol{s_t},\boldsymbol{a_t})$ and receives a reward $\boldsymbol{r_t}$ based on $\boldsymbol{R}(\boldsymbol{s_t},\boldsymbol{a_t})$. The goal of the agent is to improve its policy by modifying parameters $\boldsymbol{\theta}$ so that in the long run it collects as much reward as possible.

The main advantage of Reinforcement Learning is that this method does not require labelled training data, unlike Supervised Learning.

2.2 Imitation Learning

Imitation Learning is a Deep Learning approach. It is similar to Reinforcement Learning in terms of the training setup: it also uses the Markov Decision Process formulation to describe its procedure. In addition, it assumes, that we have access to an expert, which can solve the given problem efficiently, optimally. This expert provides us with demonstrations of the task, which we use during the agent's training. The agent eventually learns the expert's policy, which allows him to "imitate" the expert, to behave in the environment just like him.

While Reinforcement Learning methods perform well in general, in some cases the teaching process can be extremely challenging. This is especially true for an environment where the rewards are sparse. To help with this issue, we can manually design rewards functions, which provide the agent with more frequent rewards. In certain scenarios, the manual approach is even necessary, as there isn't any direct reward function (e.g. teaching a self-driving vehicle). However, manually designing a reward function that satisfies the desired behavior can be extremely complicated.

A feasible solution to this problem is Imitation Learning, which learns using the expert demonstrations and not via a reward function. In general, Imitation Learning is useful when it is easier for an expert to demonstrate the desired behavior rather than to specify a reward function which would generate the same behavior or to directly learn the policy.

Imitation Learning has two main areas: **Direct Policy Learning** and **Inverse Reinforcement Learning**. In the first approach the agent learns the policy directly from the expert's demonstrations, hence the name of the method. The aim of the second approach is to uncover a reward function by the means of the demonstrations, which is then used to learn the policy using Reinforcement Learning.

During the project I have experimented with both Direct Policy Learning and Inverse Reinforcement Learning methods. The algorithms I tried out were the following: Behavioral Cloning, DAgger and GAIL. In the following section I will present these algorithms in detail.

2.2.1 Behavioral Cloning

The simplest form of Imitation Learning is Behavioral Cloning [14], which is a Direct Policy Learning method. It focuses on learning the expert's policy using Supervised Learning. This approach works the following way: given the expert's demonstrations, we divide these into state-action pairs, we treat these pairs as i.i.d. examples and finally, we apply Supervised Learning. The algorithm therefore is the following:

1. Collect expert demonstrations – $\boldsymbol{\tau^* = (s_0^*, a_0^*, s_1^*, a_1^*, \dots)}$ trajectories
2. Treat the demonstrations as i.i.d. state-action pairs: $\boldsymbol{(s_0^*, a_0^*), (s_1^*, a_1^*)}$, ...
3. Learn $\boldsymbol{\pi_\theta}$ policy using Supervised Learning by minimizing the loss function $\boldsymbol{L(a^*, \pi_\theta(s))}$

Although in some simpler applications this algorithm can work excellently, it is bound to fail in case of tasks where long-term planning is required. The main reason for this is the i.i.d. assumption: while Supervised Learning assumes that the state-action pairs are distributed i.i.d., in the MDP an action in a given state induces the next state, which breaks the previous assumption. This also means, that errors made in different states add up, therefore a few mistakes made by the agent can easily put him into a state that the expert has never visited and the agent has been never trained on. As a result, the agent does not know how to recover from such states. This problematic behavior is illustrated by Figure 14.

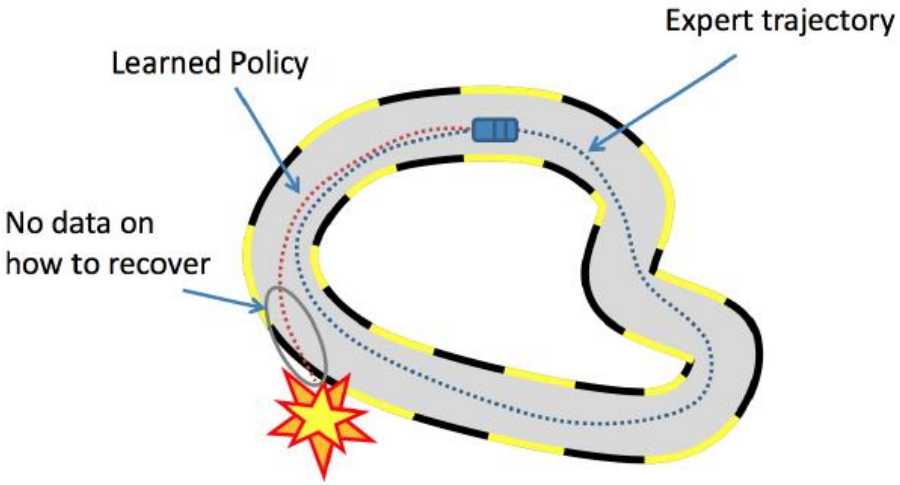

Figure 14: The main problem of Behavioral Cloning. Source: [35]

Despite all of this, Behavioral Cloning can work quite well in certain applications, where long-term planning is not needed or where the expert's trajectories cover the entire state space. The method's main advantage is its simplicity and efficiency.

2.2.2 Iterative Direct Policy Learning

Iterative Direct Policy Learning is an improved version of Behavioral Cloning. This method assumes, that we have access to an interactive demonstrator at training time, who we can query. The general Direct Policy Learning algorithm works the following way. First, we start with the initial predictor policy that we have uncovered from the original expert demonstrations using Supervised

Learning. Then, we execute a loop until we converge. In each iteration, we collect trajectories by rolling out the current policy (which we obtained in the previous iteration) and using these we estimate the state distribution. Then, for every state we collect feedback from the expert (what would he have done in the same state). Finally, we train a new policy using this feedback. The procedure is demonstrated by Figure 15.

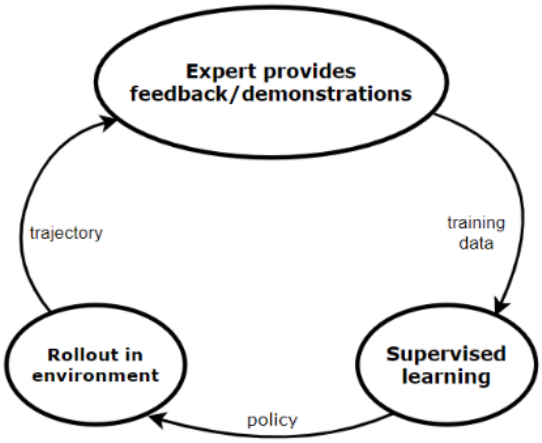

Figure 15: The general Direct Policy Learning algorithm

For the algorithm to work efficiently, it is important to use all the previous training data during the teaching, so that the agent "remembers" all the mistakes it made in the past. There are several algorithms to achieve this, the two most commonly used are: **Data Aggregation [15]** and **Policy Aggregation** (e.g. **SEARN [16] & SMILe [17]**). Data aggregation trains the actual policy on all the previous training data. Meanwhile, policy aggregation trains a policy on the training data received on the last iteration, and then combines this policy with all the previous policies using geometric blending. In the next iteration, we use this newly obtained, blended policy during the roll-out. Both methods are convergent, in the end we receive a policy which is not much worse than the expert in terms of the performance.

The full algorithm is the following:

1. Initial predictor: $\boldsymbol{\pi_0}$
2. For $m = 1$:
    a. Collect trajectories $\boldsymbol{\tau}$ by rolling out $\boldsymbol{\pi_{m-1}}$
    b. Estimate state distribution $\boldsymbol{P_m}$ using $\boldsymbol{s} \in \boldsymbol{\tau}$
    c. Collect interactive feedback $\{\boldsymbol{\pi}^*(\boldsymbol{s}) \mid \boldsymbol{s} \in \boldsymbol{\tau}\}$
        i. Data Aggregation (e.g. **DAgger**)

1. Train $\boldsymbol{\pi_m}$ on $\boldsymbol{P_1} \cup ... \cup \boldsymbol{P_m}$

ii. Policy Aggregation (e.g. **SEARN & SMILe**)

1. Train $\boldsymbol{\pi'_m}$ on $\boldsymbol{P_m}$
2. $\boldsymbol{\pi_m = \beta \pi'_m + (1 - \beta)\pi_{m-1}}$

Iterative Direct Policy Learning is a very efficient method, which does not suffer from the problems that Behavioral Cloning does. The only limitation of this method is the fact, that we need an expert that can evaluate the agent's actions at all times, which is not possible in some applications.

2.2.3 Inverse Reinforcement Learning

Inverse Reinforcement Learning is a different approach of Imitation Learning, where the main idea is to learn the reward function of the environment based on the expert's demonstrations, and then find the optimal policy (the one that maximizes this reward function) using Reinforcement Learning. This method's main procedure is described below.

First, we start with a set of expert's demonstrations (we assume these are optimal) and then we try to estimate the parameterized reward function, that would cause the expert's behavior/policy. We repeat the following process, until we find a good enough policy: we update the reward function parameters, then we solve the Reinforced Learning problem (given the reward function, we try to find the optimal policy) and finally, we compare the newly learned policy with the expert's policy.

The general Inverse Reinforcement Learning algorithm is the following:

1. Collect demonstrations: $\boldsymbol{D} = \{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \dots, \boldsymbol{\tau_m}\}$
2. In a loop:
    a. Learn reward function: $\boldsymbol{R_\theta(s_t, a_t)}$
    b. Given the reward function $\boldsymbol{R_\theta}$ learn policy $\boldsymbol{\pi}$ using Reinforcement Learning
    c. Compare $\boldsymbol{\pi}$ with $\boldsymbol{\pi^*}$ (expert's policy)
    d. STOP if $\boldsymbol{\pi}$ is satisfactory

Depending on the actual problem, there can be two main approaches of Inverse Reinforcement Learning: the **model-given** and the **model-free** approach. The algorithm for these approaches is different, this can be seen on Figure 16.

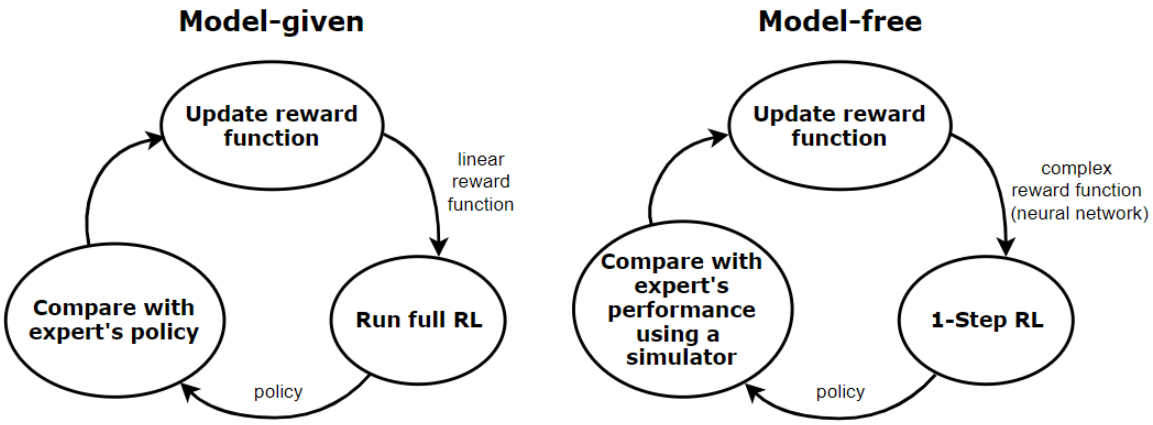

Figure 16: Model-given (left) and Model-free (right) Inverse Reinforcement Learning

In the **model-given** case the reward function is assumed to be linear. In each iteration, the full Reinforcement Learning problem is solved. To be able to do this efficiently, it is required that the environment's (the MDP's) state space is small. We also suppose that the state transition dynamics of the environment is known, which is needed to be able to compare the learned policy with the expert's one effectively.

The **model-free** approach is the more general case. It is assumed that the reward function is complex, which is usually modeled with a neural network. We also suppose that the state space of the MDP is large or continuous, therefore in each iteration only a single step of the Reinforcement Learning problem is solved. In this case, the state transitions dynamics of the environment is unknown, however, an access to a simulator or the environment is possible. As a consequence, comparing our policy to the expert's on is a more challenging task.

In both cases, however, learning the reward function is ambiguous. The reason of this is that many rewards functions can correspond to the same optimal policy (the expert's policy). To solve this issue, the most common approach is to use the maximum entropy principle proposed by Ziebart [18]: we should choose the trajectory distribution with the largest entropy.

2.2.4 Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning [19] (GAIL) is a method that adopts the GAN architecture to carry out Inverse Reinforcement Learning. It is a model-free approach, therefore it is capable of imitating complex behaviors in large, high-dimensional environments.

Similarly to GANs, the GAIL architecture (see Figure 17) also consists of two neural networks: the **policy network** (which can be interpreted as the generator) and the **discriminator**. The policy network acts as the agent's policy: it receives the agent's state (or the observation of this

state) in the environment as an input and outputs the adequate actions. The discriminator is a binary classifier which tries to distinguish the received state-action pairs from the trajectories generated by the agent and the expert. Basically, this network can be interpreted as the cost function (opposite of reward function) that provides the learning signal to the policy. It receives a corresponding state-action pair and outputs a value between 0 and 1 based on who performed the action: 0 for the expert and 1 for the agent. Note, that in opposition to the reward function, the cost function's task is to output low cost value for a policy with good performance (e.g. expert's policy) and high cost value for a policy with bad performance (e.g. any other policy).

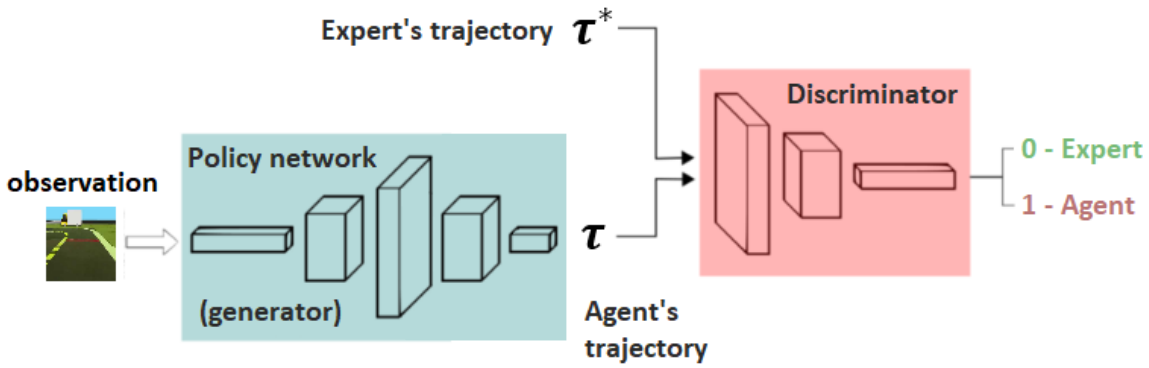

Figure 17: The GAIL architecture. Image created based on: [34]

Similarly to the GAN training procedure, training a GAIL algorithm is also a minimax game, where the policy network wants to minimize and the discriminator wants to maximize the game's objective. The GAIL objective is denoted as:

$$\min_{\pi_\theta} \max_{D_\omega \in (0,1)^{S \times A}} \left[E_{\pi_\theta} \log D_\omega(s,a) + E_{\pi^*} \log(1 - D_\omega(s,a)) - \lambda H(\pi_\theta) \right]$$

where $\boldsymbol{\pi_\theta}$ is the agent's policy with parameters $\boldsymbol{\theta}$, $\boldsymbol{\pi^*}$ is the expert's policy that we wish to imitate, $\boldsymbol{D_\omega}$ is the discriminative classifier with parameters $\boldsymbol{\omega}$ and $\boldsymbol{H(\pi_\theta) = E_{\pi_\theta} \log \pi_\theta(a|s)}$ is the $\boldsymbol{\gamma}$-discounted causal entropy of the policy $\boldsymbol{\pi_\theta}$ (proposed by Ziebart) with $\boldsymbol{\lambda}$ as a hyperparameter.

Optimization over the GAIL objective is performed by alternating between an Adam gradient step to increase the objective with respect to the discriminator parameters, and a Trust Region Policy Optimization [21] (TRPO) step to decrease the objective with respect to $\boldsymbol{\theta}$. The TRPO step prevents the policy from changing too much due to noise in the policy gradient.

Instead of directly learning a reward function, GAIL relies on the discriminator to guide the agent into imitating the expert's policy, as the policy network tries to fool the discriminator by generating expert like actions.

3 Problem Statement

This chapter focuses on the Duckietown project, which was the main environment for the experiments carried out in this work. It also presents the different tasks and challenges of the AI Driving Olympics, as well as the performance metrics and the baseline solutions.

3.1 The Duckietown platform

Duckietown is an open-source, inexpensive platform for autonomy education and research. The project was started in 2016 as a university course at MIT. Today, however, Duckietown is used at several universities worldwide and it is also featured in the AI Driving Olympics (AI-DO) global competition. The platform's main advantage is its cost-efficiency: while researchers can still solve complex and challenging problems in the field of robotics and autonomous driving, setting up the Duckietown environment is significantly cheaper than a real-world system with a fleet of fully sized vehicles. Developing such a real-world infrastructure would be also quite time consuming. Duckietown, however, provides us with several tutorials, baseline solutions and examples, which speed up this process.

The Duckietown platform consists of two main parts: the small autonomous vehicles called "Duckiebots" and the tiny cities called "Duckietowns". Each of these are standardized: there are strict specifications about both the robots and the road tiles so that the environment is the same for everyone. This is quite useful, as this way researchers can compare and benchmark their algorithms.

A Duckietown (see Figure 18) has three main elements: road tiles, traffic signs and "pedestrians", which are small rubber duckies. The road tiles are made from black foam boards and the lane marks are created from duct tape.

Figure 18: The Duckietown environment. Source: [36]

A Duckiebot (see Figure 19) is a differential wheeled robot with two driven wheels and a third, free turning wheel. It consists of a chassis, two DC motors that drive the wheels, a DC motor shield that controls the motors, a battery, a wide angle RGB camera and a Raspberry PI. The goal of Duckietown researchers is to implement an algorithm on the Raspberry PI that processes the camera's images and sets the control signals of the motors based on the extracted information.

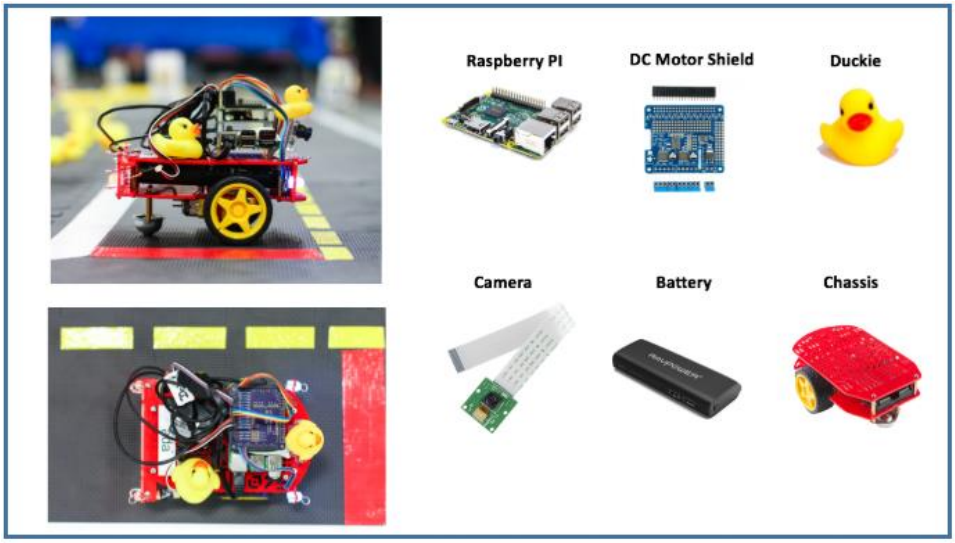

Figure 19: A Duckiebot and its components. Source: [36]

Additionally, the Duckietown platform provides an OpenAI Gym [23] environment simulator, which comes with many useful features (domain randomization, reward augmentation wrappers, etc.) that makes the training and development of algorithms quite convenient. Furthermore, the simulated environment is fairly similar to the real one (see Figure 20). The complete Duckietown software stack also has a lot of tutorials, baseline solutions, calibration routines, etc. that further supports the work of the developers. Another key feature of the software package is the evaluation interface, which makes it very easy to evaluate algorithms and solutions based on the AIDO performance metrics.

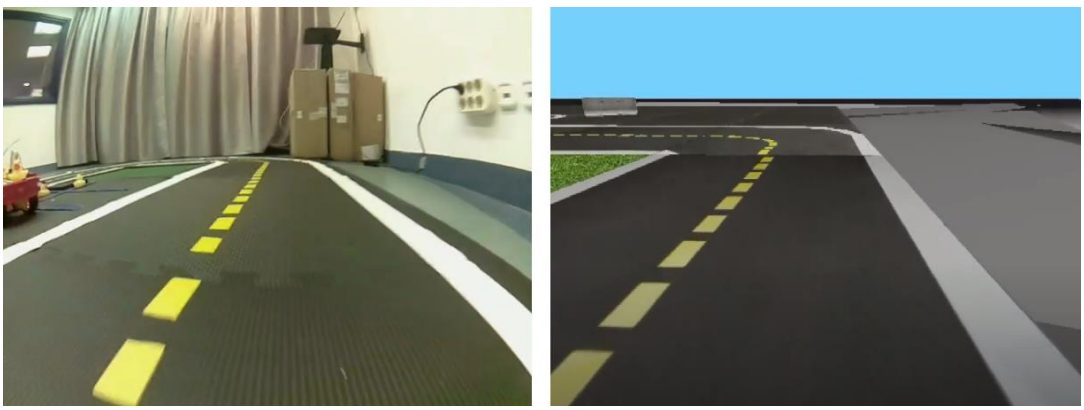

Figure 20: Real (left) and simulated (right) Duckietown environment

3.2 AI Driving Olympics

The AI Driving Olympics are global competitions organized by the Duckietown Foundation every half year since December 2018. The competition is based on the Duckietown platform and focuses around AI for self-driving cars and robotics. The whole event consists of two rounds: the online selection round and the finals. During the selection round competitors around the whole world can submit their solutions using the Duckietown online evaluation platform. The submissions are first tested in the simulator environment and if their performance is sufficient, they are later evaluated on the real physical robots in a Duckietown Autolab (also called as robotarium). The entries that reach the best scores on real Duckietowns advance to the finals. These are live events held at prestigious Machine Learning and robotics conferences around the globe, such as NeurIPS and ICRA.

The competition has 3 main challenges:

- **Lane following (LF)** where a single Duckiebot drives within the right lane in a closed road loop without intersections nor obstacles.
- **Lane following with vehicles (LFV)** where vehicles are added to the road and the Duckiebot running the competitor's algorithm has to avoid making contact while still following the right following.
- **Lane following with vehicles and intersections (LFVI)** where the competing Duckiebot navigates a road map that includes intersection while also avoiding contact with other driving vehicles.

In this work I focus on the Lane following challenge. I aim to solve this task in the simulator environment using different Imitation Learning algorithms.

4 Proposed methods

This chapter presents the conducted experimental work and the implementation details of the different solutions.

4.1 Imitation Learning

In this work, several experiments were carried out in the Duckietown simulator environment to solve the presented lane following task using Imitation Learning. As discussed earlier, Imitation Learning requires an expert to collect demonstrations, which are then used to train the agent to achieve the desired behavior. Therefore, general training procedure of my experiments was the following. First, the expert was rolled out in the environment to collect and save demonstrations. Next, these demonstrations were loaded as training data and the agent was trained on them based on the Imitation Learning algorithm. Finally, the agent was released in the environment and evaluated based on the official Duckietown metrics.

The utilized Imitation Learning algorithms are presented in section 324.1.1. The implementation details of the expert demonstrator are described in section 4.1.2.

The demonstrations are sequences of state-action pairs. In the case of the Duckietown simulator, the states are observations of the environment: images from the Duckiebot's front camera; and the actions are PWM signals that specify the Duckiebot's left and right motor speeds. To achieve better performance, I simplified both the observations and the actions during my experiments by applying a preprocessing step to the images and a postprocessing step to the actions. The details of the pre- and postprocessing steps are presented later in sections 4.2.1 and 4.2.2.

The Duckietown simulator has a built-in domain randomization functionality, which slightly changes the textures and the lightning each time the simulator is reset (see Figure 21). Even though this is primarily a Transfer Learning technique, it can also be used as a regularization method to avoid overfitting and improve the model's generalization ability. Therefore, domain randomization was turned on during the process of collecting demonstrations.

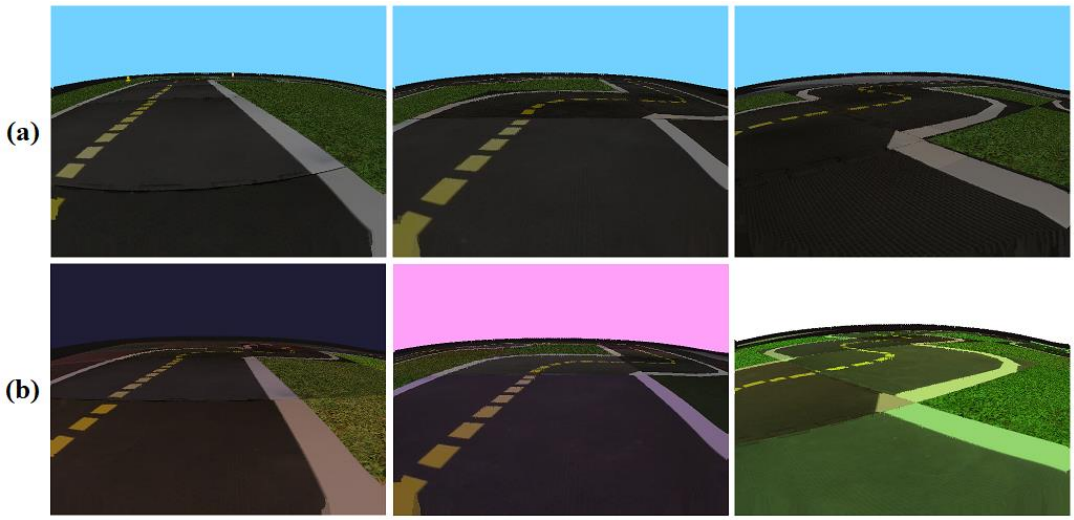

Figure 21: Observations from the standard (a) and the domain randomized (b) environment

The demonstrations were collected on multiple maps. When the expert completed its current trajectory, the environment was reset, and a new map was randomly selected from the available set of maps (see Figure 22). Most of the maps contain several objects on the side of the roads to increase variability. This multi-map training approach further improves the model's generalization ability and provides robustness on unseen track layouts.

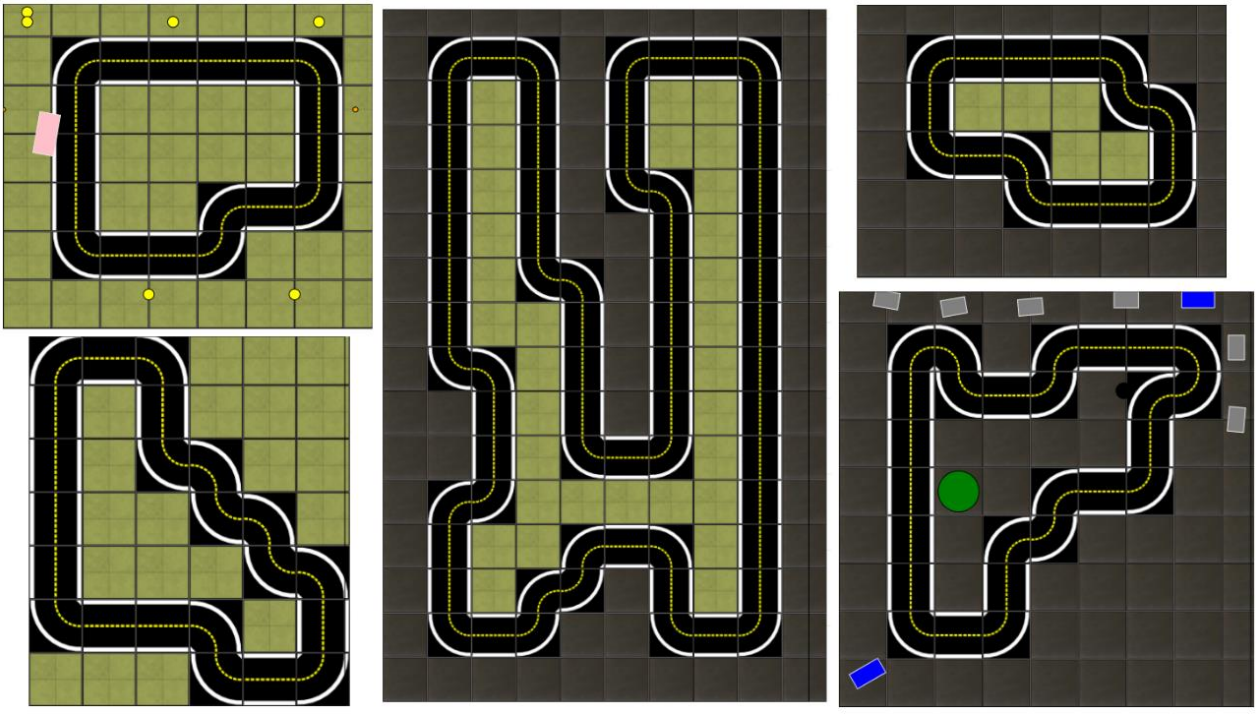

Figure 22: The maps that were used during the training

4.1.1 Applied algorithms

I have deployed three Imitation Learning algorithms: Behavioral Cloning, DAgger and GAIL. The first two methods are Direct Policy Learning techniques and the third method is an Inverse Reinforcement Learning algorithm. The theoretical background of these algorithms is described previously in section 2.2, therefore in this section I only present their implementation details.

The implementation of Behavioral Cloning was fairly simple: we perform Supervised Learning by loading in the expert's demonstrations and feeding them through the model. I used mean squared error as the loss function. The image preprocessing and action postprocessing steps were both applied during training. Two different Behavioral Cloning models were trained. The first model was trained on binary images, because the image thresholding step was applied to the observation preprocessing. In case of the second model, this step was omitted, therefore it was trained on RGB images.

The DAgger model's implementation is similar. The training procedure starts with a Behavioral Cloning pretraining step, after which, the training is continued by executing a loop. In each iteration, the current agent is rolled out in the environment and for every single observation the expert is queried to predict the corresponding action. The expert's observation-action pairs are then appended to the previously collected demonstrations and the agent is trained on the complete dataset using Supervised Learning. This loop is continued until the agent's performance is satisfactory. I used mean squared error as the loss function. The image preprocessing and action postprocessing steps were both applied during training. Two different DAgger models were trained. The first model was trained on binary images with the image thresholding step applied to the preprocessing routine. The second model was trained on RGB images without image thresholding.

The GAIL algorithm's implementation was based on a publicly available codebase [25]. This refines the standard GAIL training process presented in section 2.2.4 with several modifications. It changes the policy network's optimization method: replaces the TRPO (Trust Region Policy Optimization) step with the PPO [22] (Proximal Policy Optimization) algorithm. Similarly to TRPO, the PPO step prevents the newly acquired policy from changing too much. In addition, it utilizes first order optimization methods, which is less computationally demanding compared to the TRPO's second order optimization. As a result of the PPO algorithm, the network architecture is extended with a value function estimator network. The image preprocessing and action postprocessing steps were both applied during training process. I also used a replay

buffer [24] to reduce the variance of the gradient estimation. The policy network was also pretrained using Behavioral Cloning.

The modified GAIL training procedure is the following. We start with the pretrained policy network and execute the following training loop. First, we roll out the current policy in the environment and collect trajectories of the agent, which we store in the replay buffer. After this, we take samples of the same size from both the expert's and agent's trajectories (by sampling from the demonstrations and the replay buffer respectively). Next, we train the discriminator network, which is followed by a PPO step that updates both the value estimator and the policy estimator networks. This loop is continued until the agent's performance is satisfactory.

4.1.2 Expert implementation

I considered two options when selecting the expert: a human demonstrator and a controller-based lane following agent.

The Duckietown software stack contains an implementation of such agent: a pure pursuit controller. This algorithm uses the Duckiebot's relative position and orientation to the center of the right driving lane to calculate the adequate actions of PWM signals. It selects a point on the ideal driving line at a certain distance from the agent and controls the robot to move towards this point. This is demonstrated by Figure 23. I modified the pure pursuit controller to use different velocity and steering gain values depending if the agent is located in a straight or in a corner. The original implementation used the same value for both the straights and the corners.

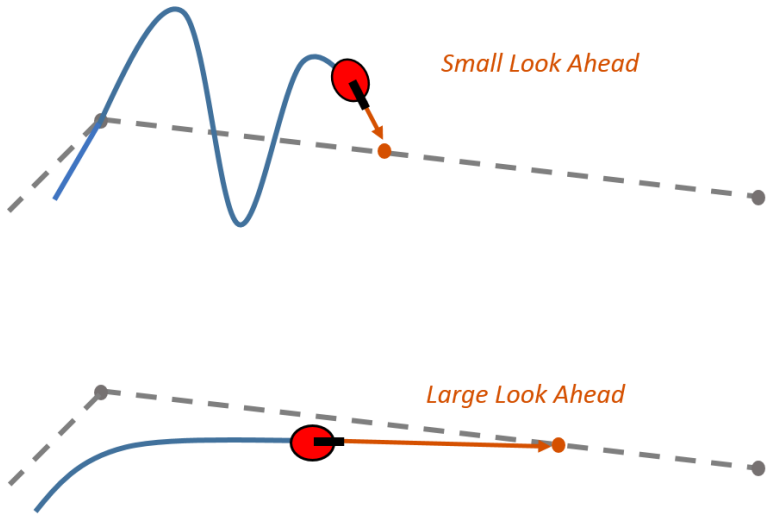

Figure 23: The operation of the pure pursuit controller. Source: [37]

I manually fine-tuned the parameters of the controller to my best abilities. The resulted algorithm is still not perfect, as it deviates around the center of the right lane, and in some cases, it also cuts the corners. However, this algorithm greatly outperforms the human demonstrator (controlling the robot with a joystick or a keyboard). Therefore, I chose the pure pursuit controller as the expert demonstrator in the Imitation Learning experiments.

4.2 Simplifying observations and actions

4.2.1 Observation preprocessing

Observations taken from the Duckietown simulator are RGB images with the resolution of 480×640 (height × width). Images of this size introduce a few problems to the learning algorithms. The high resolution results in a high-dimensional state-space, which makes it harder for the algorithm to learn a proper feature extractor. It also slows down the inference and training time of the neural network. Therefore, before feeding the images to the models, several preprocessing steps are performed in order to reduce image complexity and increase training and inference speed. The preprocessing steps are the following:

- **Downscaling**: The images are resized to a smaller resolution of 60×80 to reduce the dimensionality of the state-space and increase training speed.
- **Cropping**: The image size can be further reduced by cropping parts of the image that does not contain any useful information about the agent's position. Therefore, the top third part of the image is removed, as it is usually above the horizon: it shows the sky, not the road. After applying this step, the resolution is reduced to 40×80.
- **Image thresholding**: This is an optional step, it is not used in case of every model. In this step, the important parts of the image (the yellow and white road lane markings) are extracted to help the neural network at detecting these. This is performed by converting the image to the HSV color space and applying thresholding for white and yellow colors. If the preprocessing procedure includes this step, the outcome is a binary image (with 1 channel). If this step is omitted, the observation remains an RGB image (with 3 channels).
- **Normalization**: The pixel values are converted to floating-point numbers and are normalized to the [0.0, 1.0] range. This is a commonly used data preprocessing method that helps the training process by alleviating numerical problems of the optimization.

In case of the GAIL algorithm, these preprocessing steps (without the image thresholding) are followed by feeding the preprocessed image through feature extractor: a ResNet network that was pretrained on the ImageNet [26] dataset.

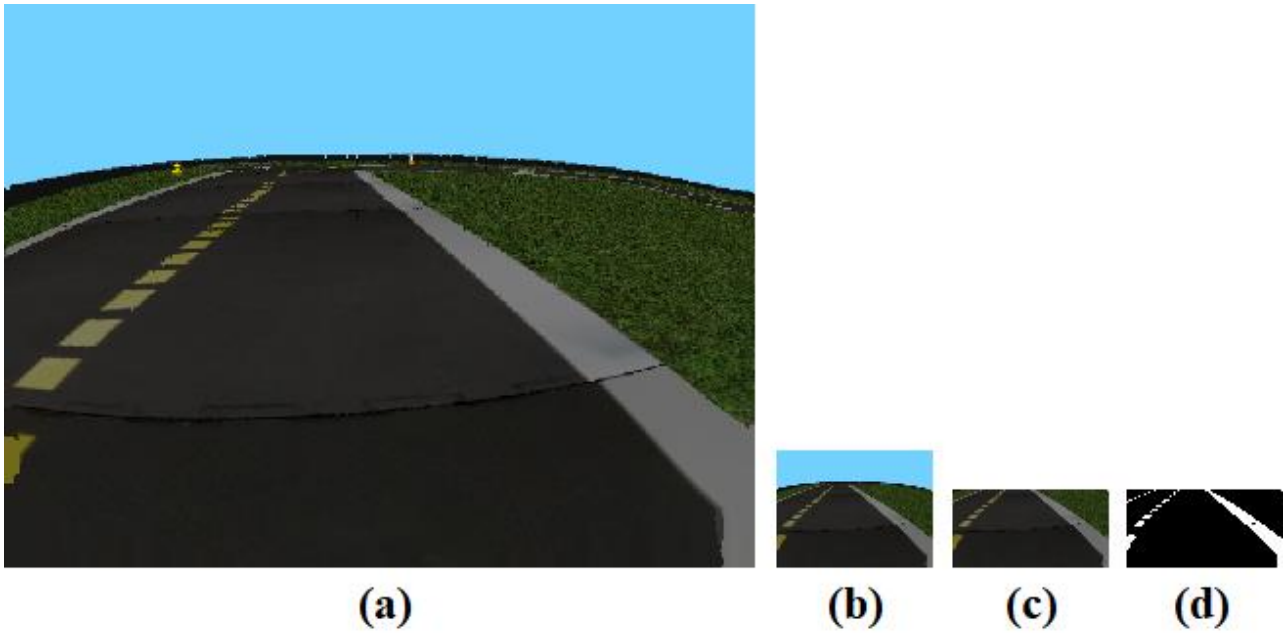

Figure 24: The applied image preprocessing steps
original image (a) resized image (b) cropped image (c) thresholded image (d)

An example of the image preprocessing steps is demonstrated by Figure 24.

4.2.2 Action postprocessing

In the simulator of Duckietown environment, the Duckiebots are controlled by actions of PWM signals, which represent the left and right motor velocities. However, during my experiments the models are trained to predict two actions: throttle and steering angle. The throttle action is a scalar value between 0.0 and 1.0, where 0.0 causes the agent to stop and in case of 1.0 the agent moves at full speed. The steering angle action is a scalar value between -1.0 and 1.0, where -1.0 and 1.0 causes the agent to turn fully to the left and right respectively, and in case of 0.0 the agent moves in a straight line. The actions predicted by the networks are then converted to PWM signals to suit the simulator.

4.3 Network architectures

In the course of the experiments I have used two different neural network architectures.

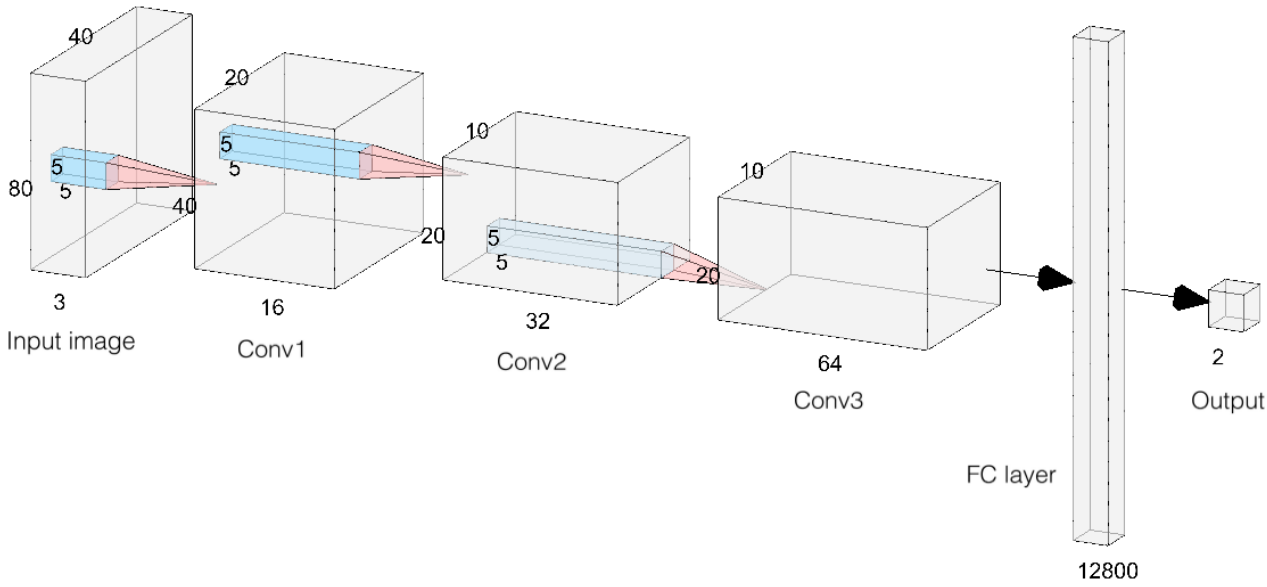

Figure 25: The neural network architecture of the Behavioral Cloning and DAgger methods

The Behavioral Cloning and the DAgger algorithms use a simple and small convolutional neural network that can be seen on Figure 25. This network consists of 3 convolutional layers, each followed by a Batch Normalization layer, a LeakyReLU activation function and a Max Pooling layer. The convolutional layers have 16, 32 and 64 filters, each with the kernel size of 5×5. The first two MaxPool layers reduce the input size to half of its resolution, the third MaxPool layer keeps the input resolution. The final convolutional layer is followed by a fully connected layer with 12800 parameters. This layer also generates the output of the network, which is a two-element vector. The input of the network is either a 3-channel RGB or single-channel binary image (depending on whether image thresholding was applied throughout image preprocessing) with the resolution of 40×80 pixels (height × width).

The GAIL algorithm uses a network architecture that consists of three single neural networks: a policy network, a discriminator network and a value function estimator network. The networks were created based on the InfoGAIL [20] architecture.

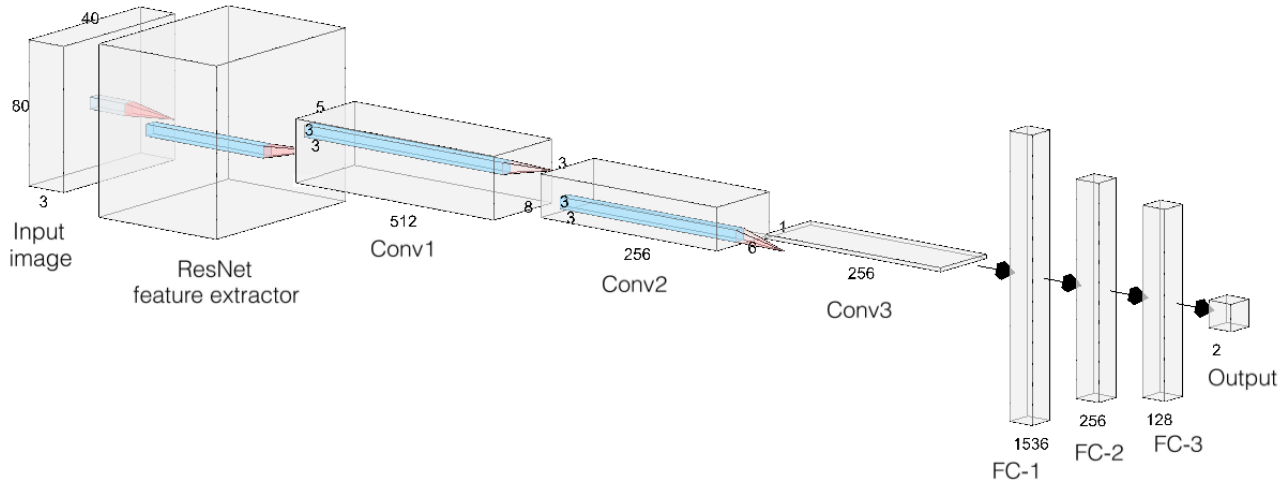

Figure 26: The neural network architecture of the policy network

The policy network is demonstrated by Figure 26. It starts with a ResNet-based feature extractor, which is created by removing the last 2 convolutional blocks of a ResNet network that was pretrained on the ImageNet dataset. This receives a 3-channel RGB input image with the resolution of 40×80 pixels (height × width) and outputs the image features with 512 channels. The feature extractor is followed by two convolutional layers, both of these have 256 filters with the kernel size of 3×3. Each convolutional layer is followed by a LeakyReLU activation function. The final convolutional layer is followed by three fully connected layers with 1536, 256 and 128 parameters. The output of the network is a two-element vector.

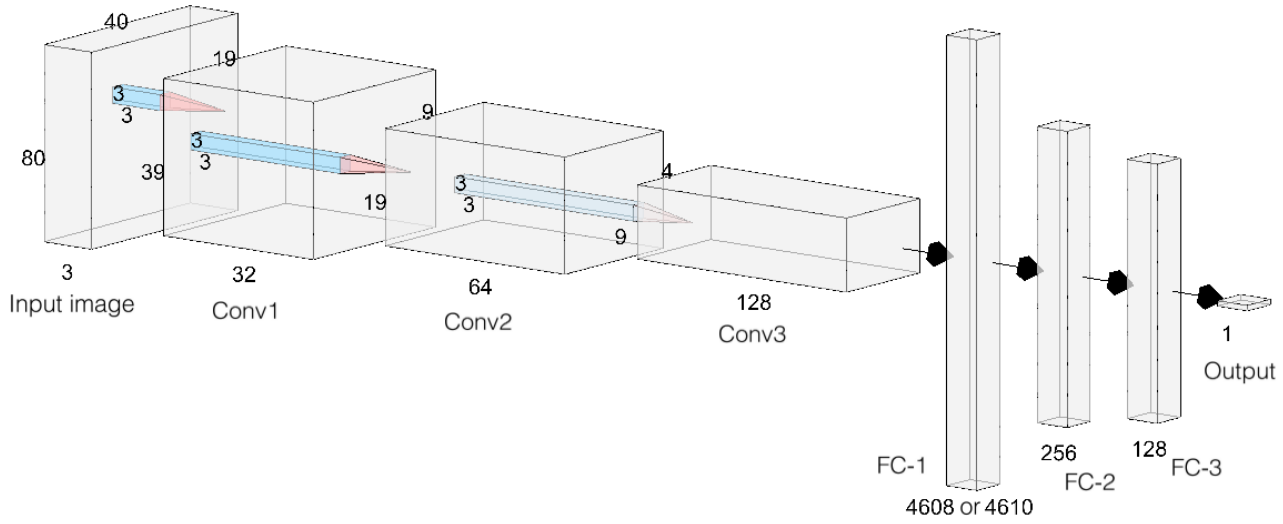

Figure 27: The neural network architecture of the discriminator and the value function estimator networks

Both the value function estimator and the discriminator networks use the same network architecture with slight modifications. The base of this network is illustrated by Figure 27. The network consists of three convolutional layers with 32, 64 and 128 filters respectively. Each filter

has a kernel size of 3×3, each convolutional layer is followed by a LeakyReLU activation function. The final convolutional layer is followed by three fully connected layers. The first of these differs in the two networks: in case of the value network, this layer has 4608 parameters, while the discriminator has extra 2 parameters (4610) because the action input is concatenated here. The remaining 2 layers have 256 and 128 parameters respectively. The output of both networks is a single value, however in the case of the discriminator network an extra Sigmoid activation function is added at the end of the network.

4.4 Training

In this study, I have conducted 5 different training experiments, which were performed with the following algorithms and setups: Behavioral Cloning with image thresholding, Behavioral Cloning without image thresholding, DAgger with image thresholding, DAgger without image thresholding and GAIL with Behavioral Cloning-based pretraining.

For both Behavioral Cloning experiments and GAIL pretraining phase, 32768 demonstrations were collected (128 episodes × 256 timesteps). Throughout both DAgger experiments, the agent was rolled out for additional 128 episodes, for 512 timesteps per each episode. The acquired 65536 demonstrations were annotated by the expert and combined with the initial demonstrations, which resulted in 98304 training examples.

The training procedure of the Behavioral Cloning and the DAgger algorithms was performed using early stopping with patience set to 20 epochs. The collected demonstrations were randomly shuffled and split into training and validation datasets using 80% and 20% of the training data. The models were trained with the Adam optimizer with the learning rate set to 0.0001. The batch size was set to 32.

The training of the GAIL method started by pretraining the policy network. After this was completed, the entire GAIL algorithm was trained for 30 epochs. In each epoch, the agent was rolled out in the environment for 15 times, each trajectory consisted of 256 timesteps. The replay buffer could store 75 trajectories from the agents, which is 19200 observation-action pairs. The models were trained with the Adam optimizer with the learning rate set to 0.0001. The batch size was set to 32.

The Duckietown DAgger baseline was trained for 50 epochs with the default parameters.

All experiments were ran on a single NVIDIA GeForce RTX 2060 GPU.

4.5 Software environment

Every software used in this work was written in the Python 3 language. The neural networks and their training procedure was implemented in the PyTorch framework. The implementation of the GAIL algorithm was based on the PyTorch-RL public github repository. This work also relies heavily on the Duckietown software stack, mainly on duckietown-gym [27] (simulator environment, baselines, example codes, etc.) and duckietown-world [28] (evaluator tools, map creator tools etc). I also used TensorBoard to log the losses of the training procedure.

5 Results

This chapter presents the results of the conducted experiments.

5.1 Evaluation procedure

5.1.1 Baseline solutions

5.1.1.1 Official DAgger baseline

The Duckietown software stack contains different baseline solutions for the challenges. One of the Imitation Learning baselines is a DAgger algorithm [29], which has a training procedure that is somewhat similar to my implementation (see section 4.1.1). I trained this model with the default parameters, based on the instructions that were provided in the authors' description. I used the resulted model as a baseline to measure and compare the performance of my algorithms.

5.1.2 Performance metrics

To evaluate the performance of each algorithm in this work, I have used the 4 most important official Duckietown metrics. As I mentioned before in section 3.1, the Duckietown software environment provides an evaluator interface, which deploys the given submission in the simulator, measures its performance by calculating each of the several performance metrics and creates a final report that contains all the results. In such a result file there are more than 20 different performance metrics, however, only 4 of these are used to rank the submissions on the official leaderboard. These 4, most important performance metrics are the following:

- **Traveled distance:** This is the median distance traveled, along a lane. (That is, going in circles will not make this metric increase.) This is discretized to tiles. This metric only measures the distance that was travelled continuously (without cease) in the right driving lane. This metric encourages both faster driving as well as algorithms with lower latency.
- **Survival time:** This is the median survival time. The simulation is terminated when the car goes outside of the road or it crashes with an obstacle.
- **Lateral deviation:** This is the median lateral deviation from the center line. This objective encourages "comfortable" driving solutions by penalizing large angular deviations from the forward lane direction to achieve smoother driving.

- **Major infractions:** This is the median of the time spent outside of the drivable zones. This objective means to penalize “illegal” driving behavior, for example driving in the wrong lane.

The evaluation procedure runs the submission for 5 episodes in the environment, which means that the robot starts from a random position and operates for a fixed amount of time. The median values are calculated from the results of these 5 runs.

Even though there is an official evaluator tool to calculate the performance metrics, this requires access to the Duckietown servers, and the code/algorithm has to comply to the appropriate submission format. To be able to display the performance metrics during training time and evaluate the algorithms locally, I created my own version of the evaluator that calculates the exact same values. My solutions were evaluated using this tool.

5.2 Results of the experiments

The models were evaluated with the Duckietown evaluator tool using the AIDO performance metrics (see section 5.1.2). Table 1 presents the best results for each training algorithm. The importance of the metrics are in an decreasing order from the left to the right (left being the most and right being the least important metric).

Median of the metric over 5 episodes	**Survival time [s]**	**Traveled distance [m]**	**Lateral deviation* [ms]**	**Major infractions* [s]**
BC w/ thr.	**15**	5.45	0.51	**0**
BC w/o thr.	**15**	5.44	0.75	0.63
DAgger w/ thr.	**15**	**5.73**	0.67	**0**
DAgger w/o thr.	**15**	5.67	0.63	**0**
pretrained GAIL	**15**	5.34	0.68	**0**
GAIL	13.55	4.78	0.71	1.27
DAgger baseline	**15**	3.97	**0.35**	**0**

Table 1: The results of the experiments

algorithms in **bold** denote own implementations
*** a lower value implies better performance**

5.3 General conclusions

Both Behavioral Cloning and DAgger algorithms managed to train a reasonably well performing model. This means that the agents were able to follow the right driving lane, without committing any crucial mistakes such as leaving the road. The Behavioral Cloning-based, pretrained policy network of the GAIL architecture achieves equally good performance. The GAIL algorithm, however, fails to improve this model. In fact, the contrary happens: the model's performance decreases. The reason of this phenomenon might be complexity of the training procedure: the parameters of the training process are probably not well chosen. Therefore, further optimization is needed for the GAIL algorithm.

The best model seems to be the one that was trained using image thresholding with the DAgger algorithm.

5.3.1 Behavioral Cloning vs. DAgger

Based on the results of the experiments, the DAgger algorithm outperforms Behavioral Cloning. However, this was expected, as the DAgger training procedure not only collects more training samples, but these examples are also more diverse, they have a bigger variability, and thus they cover a bigger state-space.

5.3.2 Image thresholding

Image thresholding helps the model to focus on the important parts of the observations: the road marking lines. Due to this, the experiments where the thresholding step was applied tend to slightly outperform those, where this step was omitted. This method speeds up the training time as well, because the model does not need to learn how to extract the useful information from the observations. In addition to this, the model inference time is also reduced, as the input images are binary images with a single channel, in opposition to the RGB images with 3 channels.

5.3.3 Comparing the results to the baseline

Most of the trained models (expect GAIL) outperform the baseline model in terms of the travelled distance (GAIL outperforms it as well, but it achieves lower value in terms of the survival time metric, which is a more important metric type). The baseline, however, has a significantly lower lateral deviation. This is due to the fact that the baseline agent moves a lot slower than the trained agents.

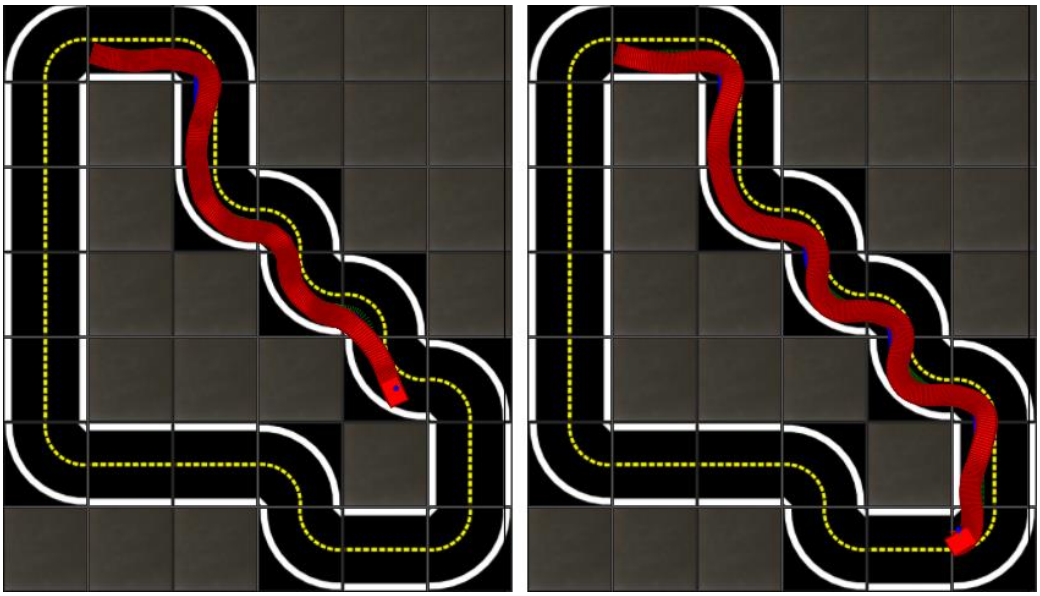

Figure 28: The trajectories of the baseline (left) and the DAgger (right) agents

Figure 28 shows a comparison between the trajectories of the baseline and the best trained model (DAgger with image thresholding).

5.3.4 Comparing the results to the AIDO 3 competition

I used the model with the best performance (the DAgger algorithm with the image thresholding) to compete in lane-following challenge of the AIDO3 competition. I managed to achieve the 1st place in the simulation-testing and 2nd place simulation-validation categories. The leaderboard of these categories is shown on Figure 29.

Leaderboard for challenge "LF - Lane following (simulation, validation)"

Rank (user)	User	Submission	complete	User label	Traveled distance↑	Survival time↑	Lateral deviation↓	Major infractions↓
1	frank_qcd_qk	5981	1/1	2019-12-09|FrankNet|v3-LFSIM	6.2	16	1.02	0
2	BME-Conti	5759	1/1	lzoltan	5.58	16	0.83	0.8
3	rrwiyatn	4676	1/1	challenge-aido_LF-baseline-duckietown	4.96	16	0.68	2.2
4	saryazdi	4769	1/1	upduck	4.96	16	0.72	0
5	bayesianduckie	4969	1/1	challenge-aido_LF-baseline-duckietown	4.96	16	0.72	1.2

Leaderboard for challenge "LF - Lane following (simulation, testing)"

Rank (user)	User	Submission	complete	User label	Traveled distance↑	Survival time↑	Lateral deviation↓	Major infractions↓
1	BME-Conti	5760	1/1	lzoltan	4.96	16	0.79	1
2	frank_qcd_qk	5986	1/1	2019-12-09|FrankNet|v3-LFSIM	4.96	16	0.83	1.2
3	miksaz	5554	1/1	chameleon-4	4.34	16	0.85	0.8
4	JBRRussia1	5940	1/1	challenge-aido_LF-baseline-IL-0226	4.34	16	0.87	1.2
5	kaland	4490	1/1	reinforcement-learning-2234s	3.72	16	0.6	0

Figure 29: Duckietown challenges leaderboard

On the leaderboard, my submissions have different metric values compared to ones I presented in Table 1. This is caused by the fact that the evaluation procedure of the official submissions differs from the local evaluations. There are some differences between the two environments as well.

6 Summary

The goal of this work was to use Imitation Learning techniques to solve a complex self-driving task: the lane following challenge in the Duckietown simulator environment. After performing a thorough research in the field of Imitation Learning, I proposed three different algorithms. Different variations of these algorithms were implemented and trained; the resulting models were evaluated based on the official Duckietown metrics.

Each of the presented methods achieved adequate performance as the trained agents were able to follow the right driving lane. I selected the best model to compete in the AI Driving Olympics and achieved great results with it.

6.1 Future works

In the future, I would like to ensure that the lane following algorithms have great performance in the real-world domain as well. To achieve this, I aim to bridge the gap between the simulated and the real environment using different Transfer Learning and Domain Adaptation methods.

In addition to this, I would like to perform further optimizations on the GAIL algorithm, as this model was the one with the poorest performance.

Acknowledgements

I would like to express my gratitude to both of my supervisors, Róbert Moni and Márton Szemenyei for continuously supporting me with useful ideas and advices through the course of this project. The research presented in this work has been supported by Continental Automotive Hungary Ltd.

References

[1] L. Paull et al.," Duckietown: An open, inexpensive and flexible platform for autonomy ducation and research," 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 2017, pp. 1497-1504, doi: 10.1109/ICRA.2017.7989179.

[2] Julian Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F. Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, A. Kirsten Bowser, Matthew R. Walter, Ruslan Hristov, Sunil Mallya, Emilio Frazzoli, Andrea Censi, and Liam Paull. The ai driving olympics at neurips 2018. arXiv preprint arXiv:1903.02503, 2019.

[3] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10), pages 807–814, 2010.

[4] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. arXiv preprint arXiv:1502.01852, 2015.

[5] Maas, Andrew L, Hannun, Awni Y, and Ng, Andrew Y. Rectifier nonlinearities improve neural network acoustic models. In ICML, volume 30, 2013

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[7] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In Neural networks for perception, pages 65–93. Elsevier, 1992.

[8] Glorot, Xavier & Bengio, Y.. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research - Proceedings Track. 9. 249-256.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

[10] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, & Ruslan Salakhutdinov (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting*Journal of Machine Learning Research, 15*(56), 1929-1958.

[11] Sergey Ioffe, & Christian Szegedy. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. (2015). Deep Residual Learning for Image Recognition.

[13] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, & Yoshua Bengio. (2014). Generative Adversarial Networks.

[14] Bain, M., Sammut, C.,” A Framework for Behavioural Cloning,” Machine Intelligence 15, 15:103, 1999.

[15] S. Ross, G. J. Gordon, and D. Bagnell.,” A reduction of imitation learning and structured prediction to no-regret online learning,” In AISTATS, pages 627–635, 2011.

[16] Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. CoRR, abs/0907.0786, 2009.

[17] Stephane Ross and Drew Bagnell. Efficient reductions for imitation learning. In Yee Whye Teh and Mike Titterington, editors, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, volume 9 of Proceedings of Machine Learning Research, pages 661?668, Chia Laguna Resort, Sardinia, Italy, 13?15 May 2010. PMLR.

[18] Ziebart, Brian & Maas, Andrew & Bagnell, J. & Dey, Anind. (2008). Maximum Entropy Inverse Reinforcement Learning. 1433-1438.

[19] J. Ho and S. Ermon, “Generative adversarial imitation learning,” in Advances in Neural Information Processing Systems, pp. 4565–4573, 2016.

[20] Y. Li, J. Song, S. Ermon, “InfoGAIL: Interpretable Imitation Learning from Visual Demonstrations,” arXiv preprint arXiv:1703.08840, 2017.

[21] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, & Pieter Abbeel. (2017). Trust Region Policy Optimization.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, & Oleg Klimov. (2017). Proximal Policy Optimization Algorithms.

[23] OpenAI. Open AI Gym, https://gym.openai.com/

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015

[25] PyTorch implementation of reinforcement learning algorithms https://github.com/Khrylx/PyTorch-RL

[26] J. Deng, W. Dong, R. Socher, L. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[27] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. https://github.com/duckietown/gym-duckietown, 2018.

[28] Duckietown-world. https://github.com/duckietown/duckietown-world

[29] Duckietown baseline DAgger algorithm. https://docs.duckietown.org/DT19/AIDO/out/embodied_il_sim_dagger.html

[30] Máron Szemenyei, Computer Vision Systems, http://deeplearning.iit.bme.hu/notesFull.pdf

[31] CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/

[32] Educative.io, Overfitting and underfitting, https://www.educative.io/edpresso/overfitting-and-underfitting

[33] Yakura, Hiromu & Shinozaki, Shinnosuke & Nishimura, Reon & Oyama, Yoshihiro & Sakuma, Jun. (2018). Malware Analysis of Imaged Binary Samples by Convolutional Neural Network with Attention Mechanism. 127-134. 10.1145/3176258.3176335.

[34] Anirudh Thatipelli, Inverse Tone Mapping using GANs, https://medium.com/@thatipellianirudh/inverse-tone-mapping-using-gans-9ff82bef6b20

[35] CS234: Reinforcement Learning Winter 2019. http://web.stanford.edu/class/cs234/CS234Win2019/slides/lecture7.pdf

[36] Duckietown official website: https://www.duckietown.org/

[37] Mathworks, Pure Pursuit Controller, https://www.mathworks.com/help/robotics/ug/pure-pursuit-controller.html