Comparing Model-Free and Model-Based Reinforcement Learning for Collision Avoidance

Scott Andreas Fiskerstrand Sørensen

Thesis submitted for the degree of
Master in Robotics and Intelligent Systems
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Comparing Model-Free and Model-Based Reinforcement Learning for Collision Avoidance

Scott Andreas Fiskerstrand Sørensen

© 2020 Scott Andreas Fiskerstrand Sørensen

Comparing Model-Free and Model-Based Reinforcement Learning for Collision Avoidance

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Autonomous cars are increasingly utilizing artificial intelligence in their systems [1][2][3]. The problem of collision avoidance for autonomous cars can be approached using reinforcement learning (RL). In this thesis we look at two approaches to RL; one where an RL agent learns a direct mapping from observations to actions called model-free RL, and one where an RL agent learns to act by using a separate, learned predictive model of the environment called model-based RL. Both model-based and model-free RL for collision avoidance has been researched and shown to be useful and effective solutions to the problem of collision avoidance [4][5][6][7][8]. However, research comparing these two methods for collision avoidance in a controlled and systematic manner seems to be lacking. In this thesis, the costs and benefits of model-free reinforcement learning versus model-based reinforcement learning for predicting and avoiding collisions in traffic situations are investigated.

These systems are trained and tested in the Carla simulation environment, a complex 3D traffic simulator [9]. The model-free RL algorithm was able to quickly learn a collision-avoidance policy, while the model-based RL algorithm failed to learn from the learned predictive representation of the environment. Additional experiments are performed, indicating that the applied model-based RL technique isn't powerful enough to create an accurate representation of the complex environments used in this thesis. Further experiments show that false predictions in the predictive component of the model-based RL technique creates a disconnect between real-world events and predicted events, hindering learning for the model-based RL algorithm.

An internal predictive model facilitates the transfer of knowledge to new tasks in environments with the same underlying rules. Once the model of the environment is learned, it can be used for many different tasks. The two systems' ability to reuse past knowledge for new tasks is tested in a transfer learning experiment. The model-based system saw a slight benefit in performance on the new task, while the model-free system trained with transfer learning performed similarly to the system trained from scratch. This indicates that reusing past knowledge is something that model-free systems struggle with, and further research on model-based reinforcement learning is necessary so that we can reap the benefits of transfer learning in RL.

Acknowledgements

I would like to thank my supervisor Kai Olav Ellefsen. This project would not have been possible without the weekly discussions, active guidance, support and all the great feedback you have given throughout this project.

I would also like to thank my co-students and friends for all the valuable discussions and tips along the way, as well as all the good memories. The ROBIN research group at the Department of Informatics provided me with a great working environment and necessary resources for this project, which I am grateful for.

Finally I would to thank my family for supporting me throughout my studies and my masters degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Every year, 1.25 million people die in traffic-related accidents globally [10], and in 93 % of cases, the accident is caused by human error [11]. Autonomous cars or autonomous assistants to human-controlled vehicles can contribute to lowering these statistics. Autonomous cars are increasingly utilizing artificial intelligence in their systems [1][2][3]. To develop safe autonomous cars, more research on the best way to train their models for collision avoidance is necessary. One way to approach the problem of collision avoidance is to use reinforcement learning, where an agent learns from interacting with the environment, relying on feedback in the form of rewards and punishment. In the field of reinforcement learning, we can either have model-free reinforcement learning, where the algorithm learns from the environment as it is, or we can have model-based reinforcement learning, where the algorithm learns from a representation of its environment.

With model-free reinforcement learning the focus is on figuring out the best way to behave directly from interactions with the environment, and there is no model or explicit representation of reality involved. The algorithm is simply given a representation of its environment as it is. Many of the recent success in reinforcement learning have been model-free, for example deep Q learning, which showed the power of using reinforcement learning to play video games, and the deep deterministic policy gradient algorithm, which showed that reinforcement learning can also be used to solve continuous control problems [12].

Research shows that humans and animals depend on mental simulations of how objects respond to interaction, representing an internal model [12]. When learning a new skill, humans already have past knowledge we can use to assist in learning that skill. For example when learning to drive a car we already know that the steering wheel is for steering, and the gas pedal is for controlling speed. We have spent a lifetime learning how our vision works, and we understand the physical rules that control our world. This allows us to quickly learn new skills. By giving reinforcement learning these abilities trough model-based reinforcement learning, we can unlock

sample efficiency, and easier transfer past knowledge to new tasks. Because of this, some researchers believe model-based reinforcement learning is the next big step forward in AI.

> "The next big step forward in AI will be systems that actually understand their worlds. The world is only accessed through the lens of experience, so to understand the world means to be able to predict and control your experience, your sense data, with some accuracy and flexibility. In other words, understanding means forming a predictive model of the world and using it to get what you want. This is model-based reinforcement learning."
> Richard Sutton[1]
> Primary Researcher at the Alberta Machine Intelligence Institute

Recent developments in deep learning have made it possible to learn internal models of physical systems from observing and collecting large datasets of the system in question [13]. This system has enabled computers to predict future inputs of physical systems before they happen, for example how a tower of blocks is likely to collapse, or how a ball on a pool table will roll [14]. These predictive systems can be used together with reinforcement learning, potentially giving the reinforcement learning system an advantage in its environment. This is model-based reinforcement learning, where a predictive model of the environment is learned, and then this model is used by a reinforcement learning algorithm to solve a problem. If the model of the environment is correct it can be used to generate a policy and plan ahead without testing out actions in the real environment [12].

Model-based RL methods can save on training time, while model-free methods need far more sets of actions in the real environment to learn. However model-free RL is less computationally complex, and it needs no model of the environment, which can be difficult to create. Both model-based and model-free reinforcement learning for collision avoidance has been researched and shown to be useful and effective solutions to the problem of collision avoidance [4][5][6][7][8]. However, research comparing these two methods for collision avoidance in a controlled and systematic manner seem to be lacking. Researching how these methods compare may give us insights into what benefits and drawbacks the two methods have, and where further research is necessary.

This thesis will investigate the costs and benefits of using the two techniques outlined above for predicting and avoiding car collisions in traffic. The model-based approach, where an explicit internal model of the physical systems of cars is learned, will be compared to the model-free approach, where learning an explicit model is avoided and only mappings from video images are learned.

[1] https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698

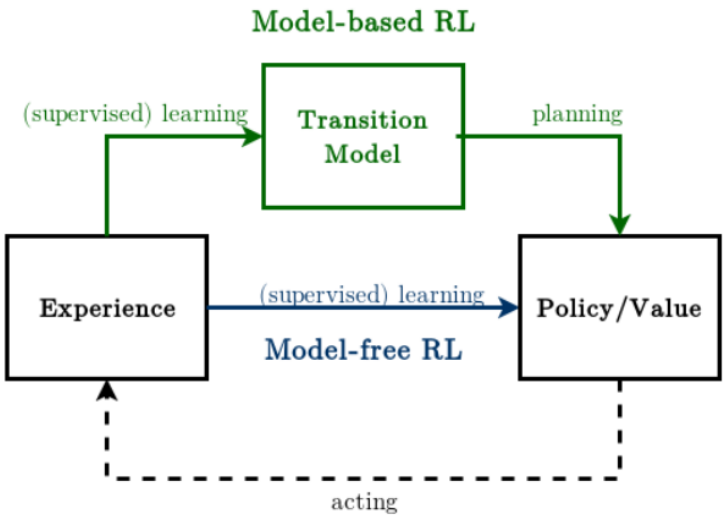

Figure 1.1: Green shows model based RL while blue shows model free RL. Figure from Moerland et al, 2017 [15]

1.2 Research questions

The main research question in this thesis is:

- Can reinforcement learning benefit from explicitly modeling the environment when learning to avoid collisions?

By explicitly modeling the environment, a representation which makes the environment easier to understand is created. This representation should allow the reinforcement learning algorithm to easier predict what actions it should take to maximize its reward. By modeling the environment the reinforcement learning algorithm should also need less training time for subsequent learning, since it needs less time to learn and understand the environment, as it is explicitly modelled already. This leads to several sub-research questions:

- How does training time and computational resources vary between model-based and model-free reinforcement learning?
- Can model-based reinforcement learning easier reuse knowledge in new scenarios?
- When using a visual model and a predictive model as the components of a model-based reinforcement learning algorithm, how does each component impact the RL algorithms ability to learn?

1.3 Outline of thesis

In Chapter 2, background theory relevant to the thesis will be presented, giving the reader a summary of the fundamental principles and methods used in this thesis. In chapter 3 related research is presented. Here a description of research related to the field of model-free RL, model-based RL and collision avoidance using machine learning is summarized and discussed, giving an overview of reinforcement learning for collision avoidance and where research is lacking. Chapter 4 describes the simulation environment used in this thesis and gives a comprehensive description of how the reinforcement learning systems are set up and trained. In Chapter 5 the reinforcement learning systems are tested and compared. Training and test results are presented, and further experiments are performed to get a deeper insight into the results. The results are analyzed and discussed. Finally, Chapter 6 presents the conclusions that can be drawn from this research project, as well as where further research is necessary.

Chapter 2

Background

In this chapter background theory relevant to the thesis will be presented. Fundamental concepts like machine learning and artificial neural networks will be touched upon first, then various systems that use these fundamental concepts will be explained, like Convolutional Neural Networks, Autoencoders, Recurrent Neural Nets and Deep Reinforcement Learning.

2.1 Machine Learning

Many modern systems for collision detection rely on machine learning [4][5][7]. Machine learning is a form of artificial intelligence where computer systems learn and improve from previous experience, by automatically modeling statistical relations from empirical data. These computer systems are given large amounts of data known as training data and automatically learn to recognize complex patterns and make intelligent decisions based on the training data. This makes it possible for these computer systems to perform a specific task without using explicit instructions, instead relying on patterns and inference. There are three main categories of machine learning. In supervised learning, the algorithm builds a mathematical model from a set of data that contains both the inputs and the desired outputs. For example, if the task was to determine whether an image contained a specific object or not, like a cat, the training data for a supervised learning algorithm would include images with and without that cat as the input, and corresponding labels designating if the image contained the cat or not as the output. There is also unsupervised learning, where the algorithm builds a mathematical model from a dataset with only inputs but no output labels. Unsupervised learning can be used to find structure in the data, group inputs into categories and discover patterns [16]. Lastly there is reinforcement learning, where an agent interacts with an environment and learns by receiving rewards for performing actions, constantly trying to get as much reward as possible. The agent learns how to behave based on past experiences (exploitation) and by taking new actions (exploration), slowly learning trough trial and error [17]. It is this form of machine learning that will be used to tackle the problem of collision avoidance in this project. This is because its

difficult to provide explicit supervision to sequential decision making and control problems, so supervised learning is not suitable for our problem. In supervised learning, algorithms try to make their output mimic the labels that are given in the training data, where there is a certain right answer for each of the inputs. For our problem, we don't have the exact right answer and labels on how to solve it. Therefore reinforcement learning is better suited to this problem, where we can instead provide our algorithm with a reward function, letting it know when it is doing something right or something wrong.

2.2 Artificial Neural Networks

Artifical neural networks are computational systems loosely inspired by the biological brain, its neurons and the neural networks in the brain. These systems learn to solve problems without humans explicitly specifying how these problems should be solved. Artificial neural networks usually consists of an input layer, hidden layers and an output layer. Each layer has multiple neurons which imitate biological neurons. These neurons are connected to the neurons in the next layer. The neurons take in input data and perform simple operations on the data, and the result is then passed on to the neurons in the next layer. Each connection between the neurons has a weight that represents its relative importance. Figure 2.1 shows how a simple neural network may look like.

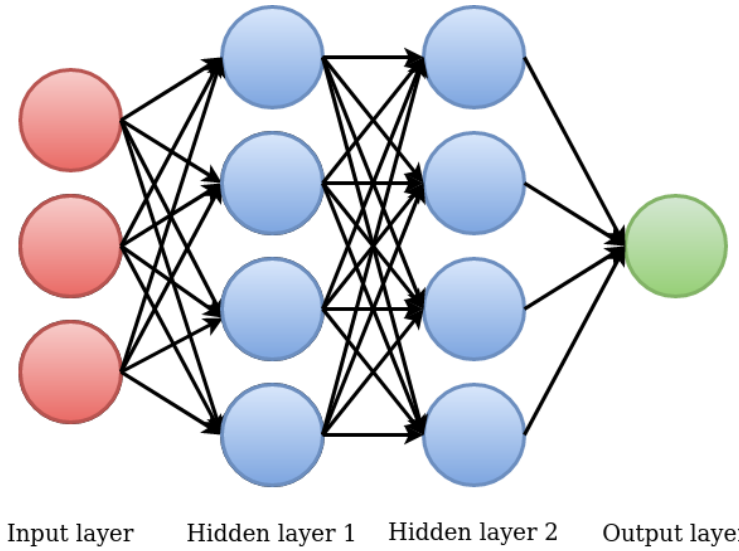

Figure 2.1: Example of a fully connected artificial neural network

The neural network learns by first feeding data through the network. Input data is sent into the input layer, this data is then sent through the hidden layers, where each layer learns to extract some low-level feature from the data. For example, if the task is to recognize an object in image

data, the first layer may look at the color of the pixels, and the next layer may identify any edges in the data based on lines of similar pixels. From there the third layer may extract shapes. Eventually the output layer will be reached, where the network will make a guess whether the object is present in the image data or not. This is called forward propagation. After it has made its guess, the result will propagate backwards through the network, adjusting the weights accordingly, making the network learn. This technique is called backpropagation. The backpropagation algorithm decides how much the weights in the network should be updated by comparing the predicted output (the networks guess) with the desired output. To update the weights, the error between the predicted output and the desired output must be calculated. This error is then used to update the weights backwards throughout all the layers in the neural network. This cycle of forward propagation and backpropagation is repeated over and over again, adjusting the values in the neural network, slowly learning from experience as it is fed data. In this way, the neural network will create a complex feature detector able to for example recognize an object in image data [16].

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a class of artificial neural networks that use the convolution operator. CNNs are highly efficient on image data, and commonly applied to image processing tasks such as image recognition and classification. The convolution operator works by having a filter slide over the pixel values of an image. This filter is a small square matrix with some numbers, usually with a size of 3x3 or 5x5. The dot product of the filter and the current pixel values in the image it is sliding over is calculated, resulting in a feature map of that area. The filter continues to slide over the image, calculating the dot product with the pixel values it covers, creating feature maps for each section it covers until it has moved over the entire image. Each of these feature maps forms a new image of features. The same convolution operation can then be performed on the new image that was formed by the last convolution operation. This operation can accentuate or dampen various features in an image, eventually extracting important features from the image. Equation 1 shows the convolution operation.

$$z[p,q] = w \cdot x = \sum_{r=-K}^{K} \sum_{s=-K}^{K} w[r,s] * x[p-r, q-s] \tag{2.1}$$

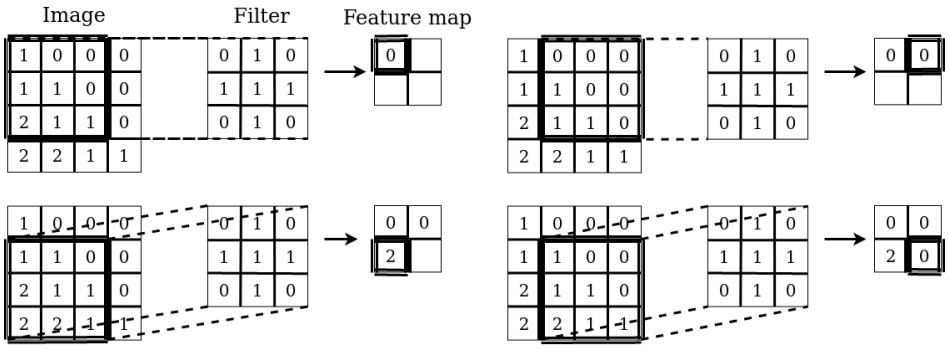

Figure 2.2: Example figure showing a 3x3 filter sliding over a 4x4 image, calculating the dot product for each step, resulting in a 2x2 feature map.

As the CNN learns, the values in the filters slowly change to extract different features, where some filters learn to extract vertical lines, other filters learn to extract corners and so on. As we move trough the layers, the filters learns to extract higher and higher level features, like shapes and objects. When working with RGB images its necessary to use three filters, one for each color channel. In the convolutional layers an activation function is added, which applies an element-wise function, such as the RELU function, that thresholds at zero. A pooling layer can also be added, which performs a down-sampling operation along the width and height of the image or feature map, reducing the size and parameters in the network. At the end of a CNN a fully connected layer is applied, which reduces the final feature map to for example a vector with three elements, where each of the 3 final numbers corresponds to a class, giving us a classification score for an image. With this network architecture, the CNN transforms the input image and its pixel values layer by layer to finally produce a class score. See figure 2.3 for an example of a CNN architecture [18].

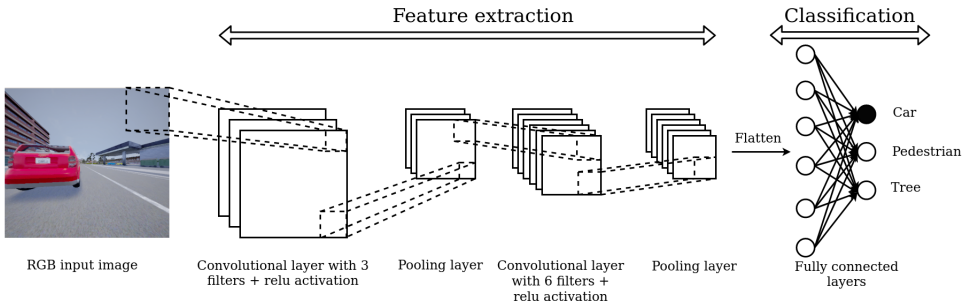

Figure 2.3: Example architecture of a CNN.

2.4 Autoencoders

Autoencoders are a type of neural network where the goal is to encode an input into a lower-dimensional code and then reconstruct this code back

to the original input. This lower-dimensional code is called a latent-space representation. Autoencoders use one neural network to encode the input to a latent space representation, then they use another neural network that takes this latent space representation as an input to reconstruct the original input, which should give a similar result as the input. Autoencoders are a form of unsupervised learning, since they don't need labeled data to train on, however they generate their own labels from the training data, so can be considered to be self-supervised. Although Autoencoders compress data, they are not like regular compression algorithms. They learn features specific for the given training data, and are thus only able to compress or encode data that is similar to the data they have been trained on, having learned to generalize on the data. Autoencoders are trained by feeding the encoder-decoder with data, and then the result of the encoded-decoded output is compared with the initial data. The error, which is the difference between the output and the initial data, is backpropagated trough the Autoencoder network, which then updates the weights of the encoder network and the decoder network. This process is repeated until the encoder is able to create a good encoding of the input data, and the decoder is able to use that code to reconstruct the input data. However Autoencoders are not good at generating new data. This is because the latent space they encode the inputs to may not be continuous and doesn't allow easy interpolation [19].

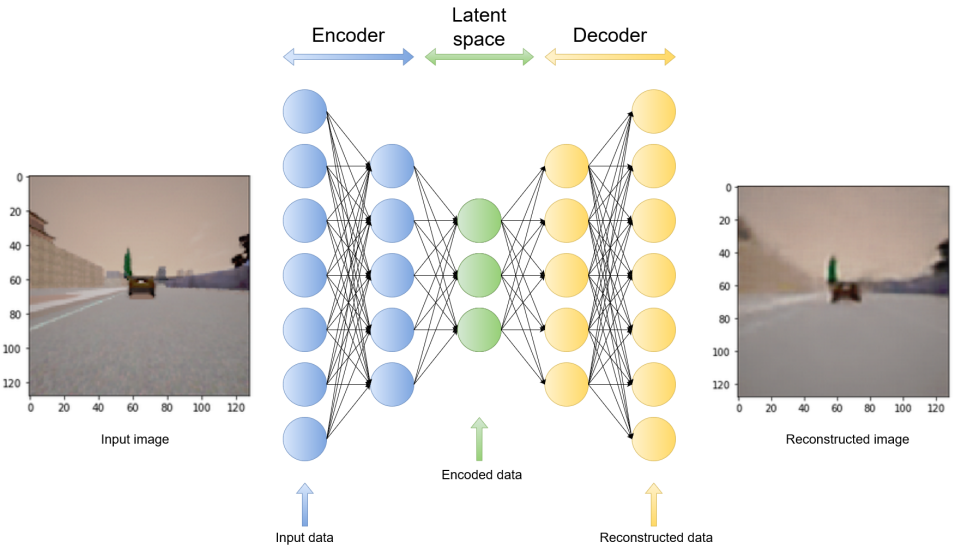

Figure 2.4: Autoencoder

2.4.1 Variational Autoencoder

Variational Autoencoders is a variation of Autoencoders where the latent space is continuous by design, which makes it easy to randomly sample and interpolate from the latent space. This is achieved by making the encoder output two vectors, a vector of means (μ) and a vector of standard

deviations (σ). These two vectors are then randomly sampled to obtain the encoded vector which we can use as input to the decoder network. From this the decoder network can reconstruct the original image. This means that even for the same input, the mean and the standard deviations will remain the same, but the actual encoding will vary due to the sampling. Preferably the encodings should be as close as possible to each other, while still remaining distinct, which enables smooth interpolation, thus making it possible to construct new samples. To encourage this, the Kullback-Liebler divergence is introduced into the reconstruction loss function. The KL-divergence measures how much two probability distributions diverge from each other. Minimizing KL-divergence encourages the encoder to distribute all encodings evenly around the center of the global latent space, while the reconstruction loss cluster similar encodings locally. This creates features that are clustered close together, with no sudden gaps in between these clusters, giving a smooth and better connected mix of features in the latent space that a decoder can understand. It is this clustering that allows easy interpolation in VAEs, which means we can generate new data [20][21].

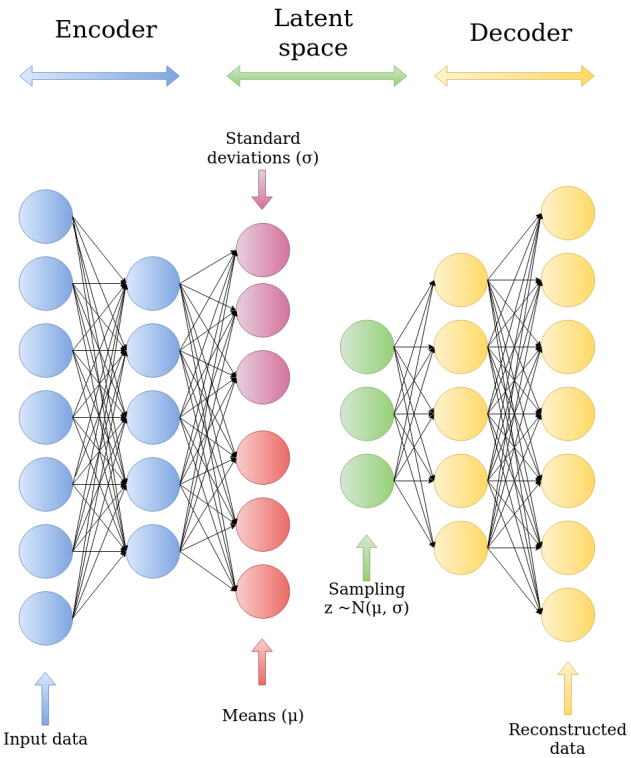

Figure 2.5: Variational Autoencoder

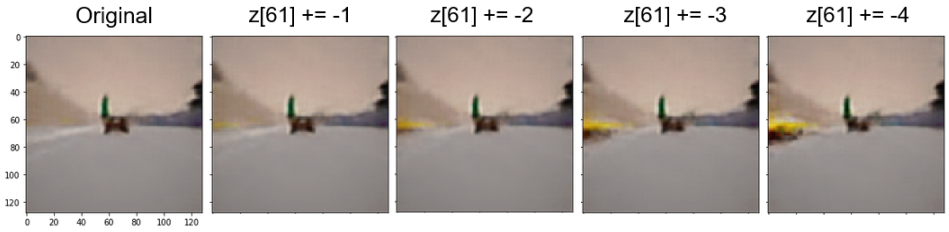

Figure 2.6: This figure shows an example of how one can generate new data with a VAE. This VAE have been trained on image data from the Carla traffic simulator. The original image shows a 128x128x3 image of a car on a road, which has been encoded to a latent vector z with size 64 and then decoded, using the trained VAE. This gives a reconstructed 128x128x3 image, as seen in the far left image. Because of how VAEs encode images, its possible to manipulate the encoded vector to generate new data. By subtracting values at index 61 in the latent vector, a new yellow car passing by on the left side of the road is generated. This may indicate that index 61 is responsible for representing cars on the left.

2.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is a type of neural network that is designed for processing sequential data, like speech and language, where context matters. For example if you want to predict the next word in a sentence, the previous words need to be taken into account. RNNs are able to remember previous data in a sequence by using the hidden state. The hidden state h_t summarizes the past sequence of inputs, and at each time step t the state is updated by the general equation 2.2.

$$h_t = fw(h_{t-1}, x_t) \tag{2.2}$$

Where h_t is the new state, fw is a function, h_{t-1} is the old state and x_t is the input at time t. In a Vanilla RNN output y_t is given by the equations

$$h_t = tanh(W_{hh}h_{t-1} + W_{hx}x_t + b) \tag{2.3}$$

$$y_t = W_{hy}h_t \tag{2.4}$$

Where W_{hh}, W_{hx}, W_{hy} are weight matrices and tanh is an activation function that regulate the values flowing trough the network, making sure they are between -1 and 1. Often a softmax layer is used at the end of an RNN, giving a probability distribution for a set of categorical predictions as output, for example which words is most likely to come next in a sentence. However RNNs suffer from short-term memory, which makes it hard to carry information from the beginning of a longer sequence. Therefore it is common to use a form of RNNs called Long Short-Term Memory (LSTM), which are capable of learning long-term dependencies [22].

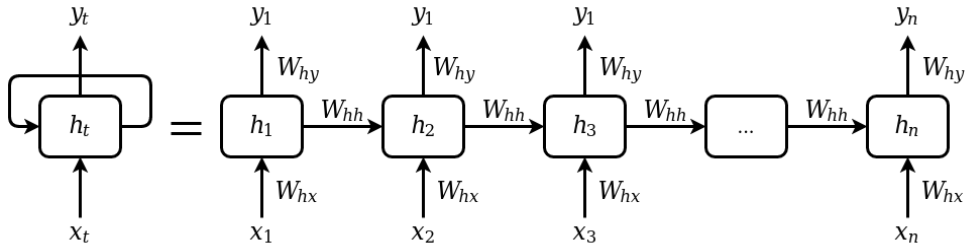

Figure 2.7: Recurrent Neural Network unrolled in time

2.5.1 LSTM RNN

The LSTM uses the cell state C_t, which acts as a conveyor belt passing previous information trough the network, functioning as the LSTMs memory. Instead of having only one neural network per module, LSTMs have four neural networks, where three of them act as gates, deciding what information should be remembered and what shouldn't. So if the LSTM detects an important feature at an early stage in an input sequence, it will carry this information over a longer distance and capture potential long-distance dependencies. The first step is to decide what information should be thrown away from the cell state. This decision is made by the forget gate f_t using a sigmoid layer to decide what should be kept (output = 1) and what should be forgotten (output = 0). The next step is to decide what new information should be stored in the cell state, which is decided by the input gate i_t. The input gate uses a sigmoid layer to decide which values should be updated, and then a tanh layer is used to create new candidate values that should be added. To update C_t, we multiply the old state C_{t-1} with f_t, thus forgetting worthless information. This is then added to $i_t * C_t$.

$$C_t = f_t * C_{t-1} + i_t * C_t \tag{2.5}$$

Finally we decide what to output using the output gate o_t. Here a tanh layer that pushes values between -1 and 1 is multiplied with a sigmoid layer σ which decides what parts of the cell state should be given as an output.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.6}$$

$$h_t = o_t * tanh(C_t) \tag{2.7}$$

LSTM-RNNs are trained in a supervised fashion on data consisting of training sequences, using gradient descent and backpropagation trough time to compute gradients used to update the weights in the LSTM network. The gates in the LSTM learn from training sequences when error values are back-propagated from the output layer. This error remains in the LSTM units cell, in what is called a Constant Error Carousel, which is continuously fed back to each of the LSTM unit gates, until they learn what to forget, remember or output [23].

2.5.2 Mixture-Density RNN

While the regular RNN can predict what is most likely to happen as a single probability distribution when using the softmax activation function, the Mixture Density RNN predict a distribution of several things that are likely to happen. A mixture density network alters the outputs of a neural network into the parameters of a gaussian mixture model. The model "mixes" several gaussian distributions with weights correlating to the likelihood of each components, to form a complex distribution. This enables the model to represent predictions that appear from multiple distributions. The mixture density models output parameters are composed of the centers (μ) and scales (σ) for each component distribution, and a weight (π) for each component. (μ) defines the location of each component, (σ) defines the width of each component, (π) defines the height of each curve. In a mixture density network, function L (see equation 2.8) measures the likelihood of t being drawn from a mixture parametrized by (μ), (σ) and (π), which are generated by the network inputs x. The loss function in a mixture density network is the negative log of the likelihood function L.

$$L = \sum_{i=1}^{K} \pi_i(x) N(\mu_i(x), \sigma_i^2(x); t) \tag{2.8}$$

This mixture density network can be applied to the outputs of an RNN to create a Mixture Density RNN. So instead of a model that only predicts one output value for each input, we get a model that has the capacity to predict a range of different output values for each input. [24] [25].

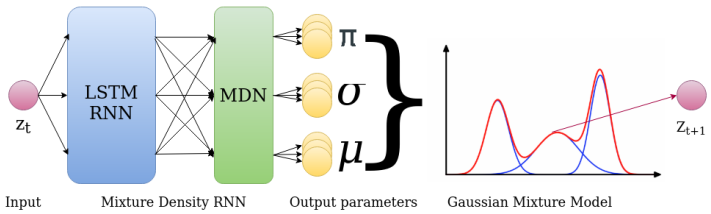

Figure 2.8: Mixture-Density RNN

2.6 Reinforcement learning

Learning by interacting with our environment is probably the first and most natural way we learn. Whether it is controlling a car or riding a bike, we receive immediate feedback from the environment, giving us a wealth of information about cause and effect. Reinforcement learning is the computational approach to learning from interaction. Reinforcement learning is a sub-field of machine learning, where an agent interacts with

the environment and learns by receiving rewards for performing actions. The goal of the agent is to maximize the expected cumulative reward, and as a result, it learns a good policy for behavior by learning from past experiences (exploitation) and by new actions (exploration), in a trial and error process. The agent must discover which actions gives the most reward on its own, with no human supervision or correct answer given to it. Reinforcement learning is a closed-loop problem where the agent's actions influence its later inputs. Actions taken in the present affects the immediate reward, but also what happens in the future and all subsequent rewards. These three characteristics of reinforcement learning, where the system is in a closed-loop, where it doesn't have direct instructions and how rewards play out over a continued time period, are the three distinguishing characteristics of reinforcement learning [17].

2.6.1 Intelligent Agent

As mentioned earlier, reinforcement learning usually has an agent that interacts with an environment. The term agent can be defined as a hardware system or more usually a software-based computer system that has been given a few specific properties. An agent has autonomy, meaning that it is able to operate and have a form of control over its actions without the direct intervention of humans or others. As well as being autonomous an agent also possess reactivity, meaning that the agent can perceive their environment and react to changes that happen in this environment in a timely fashion. An environment might be something as basic as a chessboard, or something more complex like a simulated environment of a city full of cars and pedestrians. Agents tend to be used in environments that are challenging, dynamic, unpredictable and unreliable. The environment may change rapidly, so the agent can't assume that the environment will remain static, and its unpredictable making it difficult to predict future states of the environment, often because it is not possible for an agent to have perfect and complete information about the environment. The environment being unreliable means that the actions an agent takes may fail for reasons beyond its control. Consequently, the agent must be able to abandon its plans and adapt if the environment changes in a significant way. It should not simply act in response to the environment, but also have goal-directed behavior. It will continue to attempt to achieve its goal despite failed attempts. In a game of chess for example, an agent must attempt to take actions that reach its goal of winning, by observing the environment and reacting to its changes, without human intervention [26][27].

2.6.2 Exploration vs Exploitation

To achieve a high reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to result in large amounts of reward (exploitation). But to first discover those actions, it has to try actions that it hasn't tried before. It has to explore a variety of actions and

progressively favor those that appear to be best. This problem of balancing exploration and exploitation is a typical problem in reinforcement learning. Exploitation means that the agent is making the best possible decision (by maximizing future reward), while exploration means that the agent is taking a sub-optimal actions to gain new information. Taking the sub-optimal action will mean a lower reward in the immediate future, but it may learn better strategies that enables an improved policy in the long term. There are several approaches to balancing exploration and exploitation. $\epsilon - greedy$ is one of the simplest approaches to this problem. With this approach ϵ decides what fraction of actions made is spent exploring and what fraction is spent exploiting. If ϵ is set to 0.2, then 20% of actions taken will be exploration, typically choosing a random action, and 80% will be exploitation, where the action that gives the best reward is chosen. A similar approach is $\epsilon - greedy$ with decay. Here the ϵ parameter is usually set to 1, and then for each episode of training ϵ is slowly decayed by multiplying ϵ with a decay parameter less than 1. With this approach there is a very high probability of exploring in the beginning of training, which is usually when its important to learn about the environment. Over time the probability of exploration decreases, until only actions that gives the best reward is taken [28]. See equation 2.9. Figure 2.9 shows how exploration and exploitation would be balanced using $\epsilon - greedy$ with decay if $\epsilon = 1$ and $decay = 0.9995$.

$$a_t = \begin{cases} a_t & with\ probability\ 1 - \epsilon * decay \\ random\ action & with\ probability\ \epsilon * decay \end{cases} \tag{2.9}$$

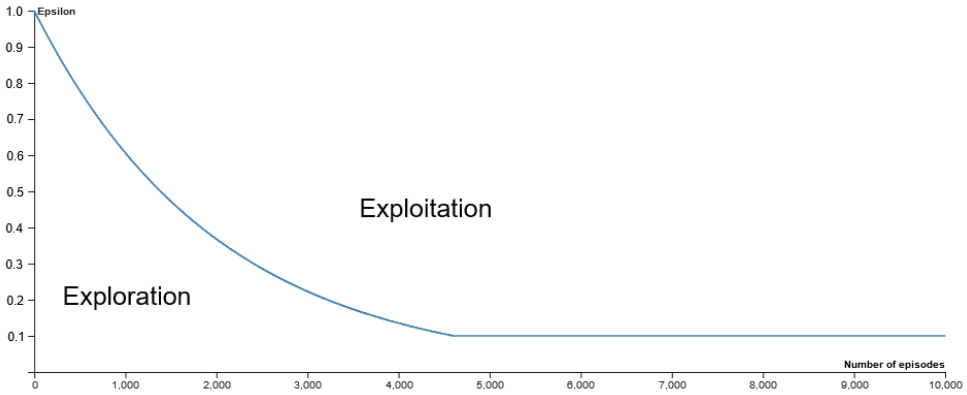

Figure 2.9: Example of epsilon greedy with decay. Epsilon = 1, decay = 0.9995, minimum epsilon = 0.1.

2.6.3 Credit Assignment Problem

The credit assignment problem is the problem of turning feedback into strategy improvements. When doing online learning, which typically is the case for reinforcement learning, an agent repeatedly makes actions of some kind, while repeatedly getting some kind of reward or feedback. But its

difficult to associate specific rewards with specific actions or combinations of actions. A critical action might be taken at time step 10, while the payoff will not come until time step 20. For example an agent playing a game of chess, the feedback is either win, lose or draw. It might have taken 100 different moves to eventually win the game, but it is difficult to assign how each of those 100 moves contributed to the result of the game. For an agent to maximize the reward in the long run, the agent needs to determine which actions will lead to a good reward [29].

2.7 Model based reinforcement learning

As mentioned earlier, in reinforcement learning we have an environment that an agent interacts with. The agent tries to maximize the expected reward by performing actions in that environment. In model-based reinforcement learning the agent is given a model of the environment and its possible actions. This model is a representation of the reality that the agent resides in. If the environment was on a road, the model would consist of representations of the various objects and obstacles on the road, like surrounding cars, signs, pedestrians, the sidewalk and buildings. While model-free methods learn by directly interacting with the environment, model-based methods is given a representation of reality, which might help it learn how to solve its task. In some forms of model-based reinforcement learning, the agent looks at possible actions it can take in the environment to figure out what are good moves and bad moves before actually executing those actions. Thus the agent creates a model of the environment and its possible actions.

Model-based reinforcement learning saves on training time when in complex environments, since making a reduced set of actions to create a model, then using this model to simulate episodes is much more efficient. There is no need to wait for the environment to respond nor to reset the environment to some state to resume learning. An internal predictive model also makes the transfer of knowledge to new tasks in the same environment easier. Meanwhile model-free methods need far more sets of actions in the real environment to learn what the optimal actions are. However model-based RL is likely to produce bad policies if the learned predictive model is imperfect, and so far model-free RL has been more successful for complex environments. The model-based method also uses more assumptions and approximations, and may therefore be limited to specific types of tasks. Below is an example of one way to do model-based RL, which can be summarized by 5 main steps [30][12].

- The agent performs actions in the real environment and gain experience (states and rewards)
- Then a model is derived and used to generate samples of possible actions that can be taken (planning)
- Value functions and policies are updated from samples

- Value functions and policies are used to choose actions to carry out in the real environment
- These steps are then looped through again, thereby gaining new experience and improving the model, the policies and the value functions

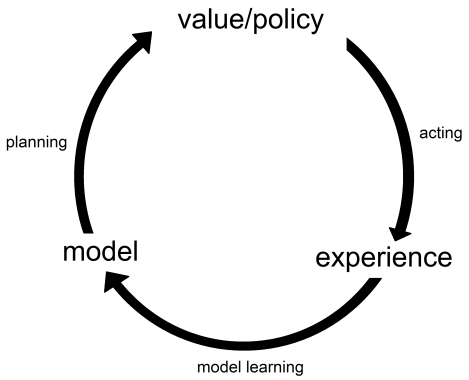

Figure 2.10: The looping steps of model based reinforcement learning.

2.8 Model free reinforcement learning

In model free reinforcement learning the focus is on figuring out the value functions directly from interactions with the environment. The agent does not have access to a model of the environment. So we attempt to solve the problem without forming an explicit model of the environment, and instead rely on learning a mapping from observations to values or actions. The agent does not try to understand the environment, and instead just get experience and try to figure out a policy of how to behave optimally to get the best possible rewards. There are several approaches for solving these problems. Two of these are Monte Carlo and Temporal Difference.

The Monte Carlo approach learns its value functions directly from episodes of experience, and the reward is given at the end of an episode. Monte Carlo learns from complete episodes, where the goal is, given a policy, learn a wanted value for the policy from episodes of experience. Monte Carlo uses empirical mean return instead of expected return for policy evaluation. Two common methods for policy evaluation is first visit Monte Carlo and every visit Monte Carlo. With first visit Monte Carlo, returns are averaged only for the first time state s is visited.

1. Initialize the policy and value function
2. Begin by generating an episode according to the current policy

2.1. Keep track of the states encountered through that episode

3. Select a state in step 2.1

   3.1. Add to a list the return received after first occurrence of this state

   3.2. Average over all returns

   3.3. Set the value of the state as that computed average

4. Repeat step 3

5. Repeat 2-4 until satisfied

In every visit Monte Carlo, returns are averaged for every time state s is visited in an episode. So in step 3.1 we add to a list the return received after every occurrence of this state, instead of just adding the return received after the first occurrence of this state.

In the Temporal Difference approach learning also happens directly from experience with the environment. Temporal Difference learns from incomplete episodes, by bootstrapping. The value function is updated immediately, which allows it to learn before knowing the final outcome after every step, unlike Monte Carlo which must wait until the end of an episode before the return is known. Temporal Difference works in continuous (non-terminated) environments, while Monte Carlo only works in episodic (terminating) environments.

The benefit of model free RL is that it needs no accurate representation of the environment (the model), which can be difficult to create [30]. However the model free approach requires enormous amounts of training data, and it is difficult to transfer a learned policy to a new task in the same environment [12].

2.9 Deep Q Learning

A further derivative of the Temporal Difference approach is Q-learning, which have achieved ground breaking results. Q-learning is an off policy RL algorithm that seeks to find the best actions to take given the current state. Its considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions. Q-learning seeks to learn a policy that maximizes the total reward. The q in Q-learning stands for quality, which represents how useful a given action is in gaining some reward. Q-learning uses a Q-table where Q-values are updated and stored after an episode. This Q-table becomes a reference table for the agent to select the best action based on the Q-value. The updates occur after each step or action and ends when an episode is done. The three basic steps for updating are as follows:

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)

2. Agent selects action by referencing Q-table with highest value (max) or by random selection (epsilon)

3. Update q-values

However creating and updating a Q-table can quickly become ineffective in large state space environments. To solve this we can use a neural network to approximate the Q-value function, which is called deep Q-learning. With deep Q-learning a neural network is given the state as input, and then a Q-value of all possible actions is given as output, and the next action is given by the maximum output of the neural network [31].

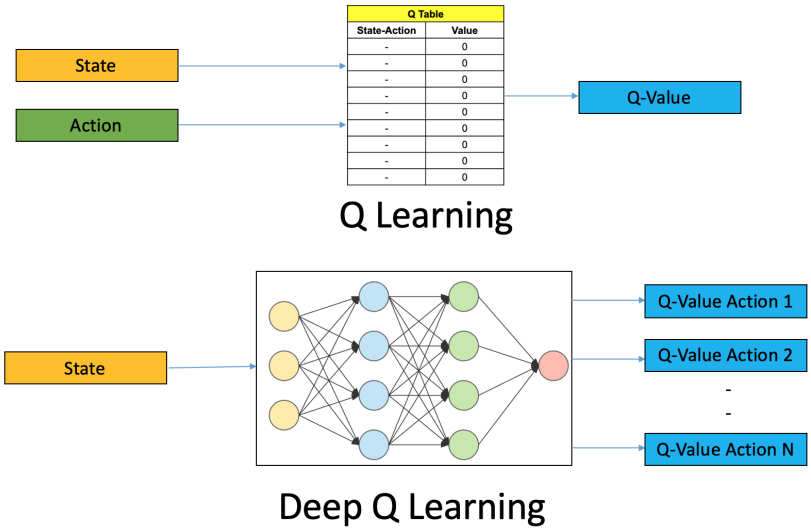

Figure 2.11: Figure showing the difference between Q-learning and deep Q-learning. Figure from Ankit Choudary, 2019[32]

2.9.1 Double Deep Q Learning

Q learning struggles with overestimations of action values. These overestimations are a result of using maximum action value as an approximation for the maximum expected action value. For example, if we have a Markov Decision Process with four states, where two of the states are terminal states, and we start at State A (see figure 2.12), with two actions, move left or right. The right action gives zero reward and lands in terminal state C. While moving to the left gives zero reward and lands in state B. From state B a number of actions can be taken, all moving to terminal state D. The reward of each actions from B to D has a random value that follows a normal distribution with mean -0.5 and a variance of 1.0. Meaning that over a large number of experiments the average reward of moving from B to D is -0.5. Based on this assumption, the best action to take in state A is to move right to terminal state C, which gives a higher expected reward, since 0 > -0.5. However because some of the rewards are positive when you move from B to terminal state D, Q-learning will think that moving from state A to B to D is the optimal actions to take. It may give positive rewards for some episodes, but in the long run its guaranteed to give a negative reward.

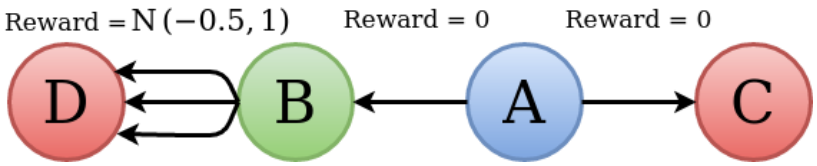

Figure 2.12: Example of a Markov Decision Process

This problem leads to the agent choosing the non-optimal action in any given state. To handle this problem, Double Q learning was introduced. Double Q learning uses two different action value functions, Q and Q', as estimators. The Q function is used to select the best action with maximum Q-value for the next state, while the Q' function is used to calculate the expected Q-value by using the action selected by the Q function. In this way overestimations are reduced by decomposing the max operation in the target into action selection and action evaluation [33]. This concept can also be applied to Deep Q Networks, giving us Double Deep Q Networks (DDQN). DDQN uses two different neural networks to learn and predict what action to take, a Q Network and a Target Network. The Q network is used to choose the best action to take for the next state (the action with the highest Q value), and the Target network is used to calculate the target Q value of taking that action in the next state. This method reduces the overestimation of Q values, helps the network train faster and results in more stable learning [34].

Chapter 3

Related work

This chapter intends to give an overview of research relevant to the thesis. Related research on model-free and model-based reinforcement learning will be presented, as well as research on collision avoidance.

3.1 Research on model free reinforcement learning

Model free reinforcement learning have shown groundbreaking results in solving a number of tasks. See section 2.8 for a description of model-free RL. There have been many recent successes in scaling reinforcement learning to complex sequential decision making, which was kickstarted by the Deep Q-networks algorithm [35]. Its combination of Q-learning with convolutional neural networks and experience replay enabled it to learn from raw pixels how to play many Atari games at human level performance. With experience replay the agents experiences is stored in a large table, and these experiences is sampled from later for the agent to learn from. Since then many extensions have been proposed that enhance its speed or stability. Researchers have experimented on various combinations of these improvements and enhancements, and integrated the components of various improvements into a single integrated agent, which they call Rainbow. They showed that combining these improvements provides state of the art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance [35][36]. Researchers have also shown that reinforcement learning is able to perform better than humans in highly complex environments, like for example in the game of StarCraft 2. From the perspective of reinforcement learning, StarCraft is a very difficult problem. Firstly, it is an imperfect information game. Players can only see a small area of the map through a local camera and there is a fog of war in the game. Secondly the state space and action space in StarCraft is huge. There are hundreds of units and buildings, and each of them have unique operations, making action space extremely large. A full length game also lasts from 30 minutes to more than an hour, and thousands of decisions have to be made. The combination of all these issues makes it a big challenge for reinforcement learning. Researchers at DeepMind recently unveiled AlphaStar, which

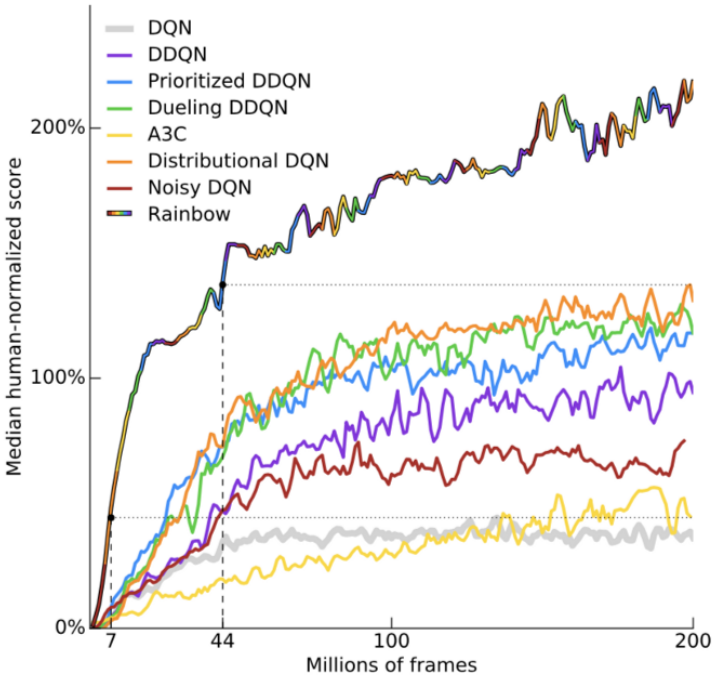

Figure 3.1: Median human-normalized performance across 57 Atari games. Comparing the integrated agent (rainbow-colored) to DQN (grey) and six published baselines

was able to beat professional players in a series of matches of StarCraft 2 [37], showing that reinforcement learning can be used in highly complex environments as well, such as StarCraft 2 or a crowded city or highway with multiple agent cars and pedestrians.

3.2 Research on model based reinforcement learning

In this section research on model-based reinforcement learning is presented. See section 2.7 for a description of model-based RL. Researchers at the University of California, Berkley have managed to use a model-based neural network to create an agent equipped with an internal model of the dynamics of the external world. The agent was trained trough random interactions with a collection of different environments, and the resulting model was then used to plan goal-directed actions in environments that were previously never encountered. The researchers demonstrated that the agent can plan actions for a game of billiards, and predict how the balls on the pool table will roll [14].

Model-based RL has also shown competitive results when it comes to solving several Atari games. Researchers at Google Brain introduced SimPle, a complete model-based deep RL algorithm that utilizes video prediction techniques, that can train a policy to play a game within the learned model. SimPle outperforms model-free methods in terms of learning speed in nearly all Atari games. While the best model-free RL

algorithms require tens or hundreds of millions of time steps for learning, SimPle is able to achieve competitive results with only 100K interactions between the agent and the environment. However SimPle was not able to beat the best state of the art model-free methods when looking at the final scores in the Atari Games. This is mainly because model-based RL excels more in learning efficiency than in final performance. The model also makes guesses when it extrapolates the rules of the game under a new policy, which results in large differences in performance between different runs on the same game [38]. The research shows that a lot of training time can be saved when using a model and that it is also possible to get competitive results compared to model-free RL. However SimPle's final scores were lower than the best state of the art model-free RL, and when it comes to avoiding collisions its important that the performance is as accurate as possible, since the result of a collision can be catastrophic.

PlaNet is another promising model-based RL algorithm. With PlaNet the environment dynamics are learned from images, and actions are chosen using fast online planning in latent space. To enable the dynamics model to accurately predict rewards for multiple time steps, a dynamics model with both deterministic and stochastic transition components where used. The agent is able to solve various continuous control tasks using only pixel observations. PlaNet was compared to strong model-free algorithms and was able to reach competitive and sometimes better results while using significantly fewer episodes. PlaNet was evaluated against model-free RL on five tasks with various challenges, and the agent was only given image observations and rewards. One task featured a pole that had to be balanced on a moving cart, while another task had the agent control the legs of a cheeta-like model and make it run. Another task was to control the legs of a bipedal walker and make it stand up and walk. PlaNet uses a latent dynamics model, which is a sequence of hidden or latent states. To make predictions PlaNet doesn't predict from one image to the next image but instead makes predictions in the latent states. By compressing the images in this way, the agent can automatically learn more abstract representations, like for example positions and velocities of objects. This makes it easier to make predictions without having to generate images along the way, which can be useful when trying to avoid collisions in traffic since there is usually very little time to react and make decisions [39].

3.2.1 World Models

In 2017, David Ha and Jorgen Schmidhuber released their paper called World Models, in which they used model based learning to reach breaktrough scores in two popular reinforcement learning environments, Car Racing and Doom. Their world model could quickly learn a compressed spatial and temporal representation of the environment, which then could be used by a simple agent to learn a policy that solved the required task. To create a model of the environment, they used a Variational Autoencoder (VAE) and a Mixture-Density RNN (MD-RNN). See section 2.4.1 and section 2.5.2 for more details on Variational Autoencoders and

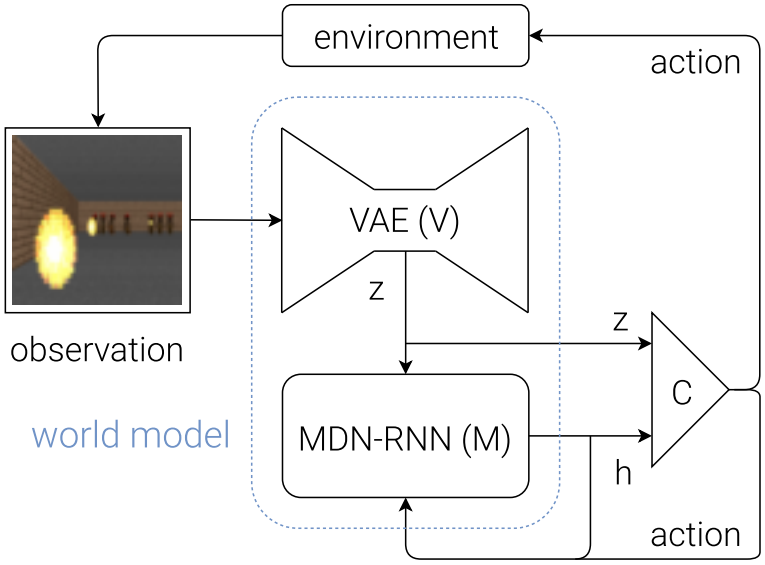

Figure 3.2: Schematic showing how the VAE, MDN-RNN and controller (C) work together

Mixture-Density RNNs. The VAE encodes an image of the current state of the environment into a latent vector, to generate a compressed spatial representation of the world. The MD-RNN takes this latent vector as well as the action of the agent as input to create a temporal representation of the world. The latent vector from the VAE is then concatenated with the hidden states of the MD-RNN and sent as input to a controller. The controller was deliberately made as simple and small as possible, so that most of the agents complexity was found in the world model (VAE and MD-RNN). To optimize the controller they used the Covariance-Matrix Adaptation Evolution Strategy (CMA-ES). Using this technique a score of 906±21 over 100 random trials was achieved, solving the task and obtaining new state of the art results in the carracing environment. See figure 3.2 for a schematic of their model. This model was also tested in the Doom environment. The world model is able to generate the environment on its own, which made it possible to train the agent without ever having to play the actual game. Since the world model was trained to mimic the Doom environment, it could simulate the essential aspects of the game, like game logic, enemy behaviour, physics and 3d graphics rendering. After training in the environment generated by the world model, the agent could be deployed in the actual game environment, and here it reached a score of 1100 time steps, well beyond the required score of 750 to consider the game solved. This approach removes the need to render images and physics with computationally heavy game engines during training of the agent [40].

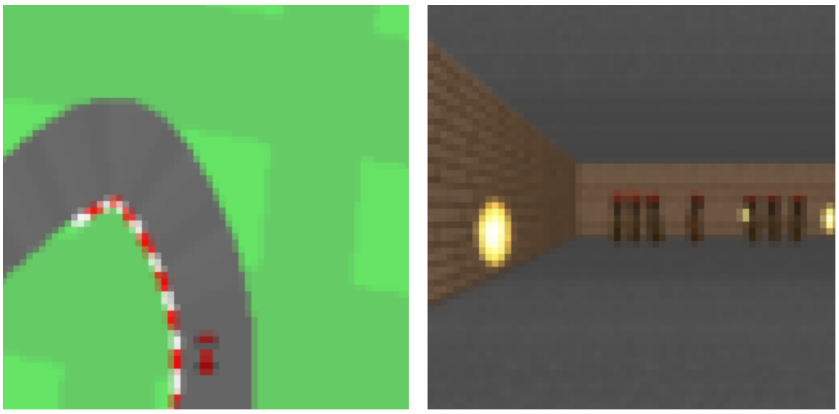

Figure 3.3: Example images from the environment that the World Models algorithm was trained and tested on. The left image shows the carracing environment, where the red car needs to be controlled so it stays on its track. The right image shows the doom take cover environment, where an agent can go either left or right, and has to dodge incoming fireballs.

3.3 Research on collision avoidance

Researchers at Nanyang Technological University proposed an extension on the DDQN architecture called two-stage noisy double deep Q-network to address the problem of collision avoidance. DDQN (Double Deep Q-Network) is a derivative of deep Q-learning that reduces the problem of overestimation of Q values, speeds up training and results in more stable learning (see section 2.9.1). The two-stage noisy DDQN uses a noisy network where parametric noise is added to the weights to achieve better exploration in the first stage, and then the noisy network is removed in the second stage where the model is further tuned. The network was given depth images which are sent trough convolutional layers to obtain feature representation, which is then mapped to Q-values using fully connected layers, and then the optimal action is determined based on the Q-values. The network was trained and evaluated in a simulation environment where they used a mobile robot trying to navigate in a room. The two stage noisy DDQN was compared to a DDQN and a noisy DDQN. Noisy DDQN was better than DDQN, but it was difficult to optimize towards the later phase. With the two stage approach where the noise was removed towards the end the network got the benefit of higher training and exploration efficiency introduced by the noisy network, while being able to better optimize at the second stage where the noise was removed [4].

In the paper "Safe Driving Mechanism: Detection, Recognition and Avoidance of Road Obstacles" the authors used various supervised learning methods to detect obstacles in an environment simulating cars on a road with traffic and objects on the road. Among the supervised learning techniques used for obstacle detection were decision trees, K-nearest neighbors, random forest and multilayer perceptron. With decision trees they were

able to reach 97% accuracy with the test set, while with random forest and multilayer perceptron they reached 99.4% accuracy on the test set. When an object was detected they used reinforcement learning for collision avoidance. Here the action set consisted of acceleration (positive and negative) and steering. For rewards they used a positive reward if the car stayed on the road and there was no collision, and a negative reward if the car went off the road and a collision was detected. With this system they were able to avoid colliding with obstacles, return to the correct lane and reach the intended destination [5].

Researchers at the University of California, Berkeley developed a collision avoidance system using model-based reinforcement learning with a real world mobile robot equipped with a camera, where the robot had to experience collisions at training time. However high speed collisions at training time can damage the robot, so they added uncertainty awareness, where the robot will slow down and be more cautious in environments that are unfamiliar. In familiar environments where it has more confidence, it will increase its speed while avoiding obstacles. Their method predicts the probability of a collision based on raw input from a camera and a sequence of actions, using deep neural networks. This predictor was used with model-predictive control to choose actions that avoids collisions with obstacles. The uncertainty awareness was mainly used during training time, however it can be useful in a fully finished collision avoidance system as well, especially in crowded city centers, where there are a lot of obstacles and unpredictable human activity [6].

In Chae et al 2017 [7] an autonomous braking system to avoid crashing in crossing pedestrians was developed using a Deep Q Network and a simulated car equipped with a camera sensor. The action space of the car consisted of 1) no braking 2) weak 3) mid 4) strong. They used trauma memory where previous collision experiences where stored. This trauma memory was used alongside replay memory, reminding the agent of previous collisions, thus allowing the agent to learn to avoid collisions consistently. The agent was able to successfully avoid collisions when TTC (time to collision) was 1.5 seconds or higher. With lower TTC some collisions where observed. They found that with the trauma memory the value function converged after 2000 episodes, and a high total reward was attained after convergence. While without the trauma memory, the policy didn't converge and continued to fluctuate.

The paper "Imminent Collision Mitigation with Reinforcement Learning and Vision" sought to reduce the severity of on-road collisions by controlling both steering and velocity in situations where collisions are imminent. Using only camera images as input, they constructed a model that is capable of learning and predicting the motion of pedestrians, obstacles, and cars. Two models that were able to both steer and brake were compared to a baseline model that was only able to brake. Their model consisted of a Variational Autoencoder (VAE), to reduce dimensionality and compress obser-

vations to a latent representation. Then a Recurrent Neural Network(RNN) is used to predict the next latent representation. Finally, a Deep Deterministic Policy Gradient (DDPG) is used as a controller, where the DDPG learns to take actions based on the predicted latent representations. Figure 5 shows an overview of their model.

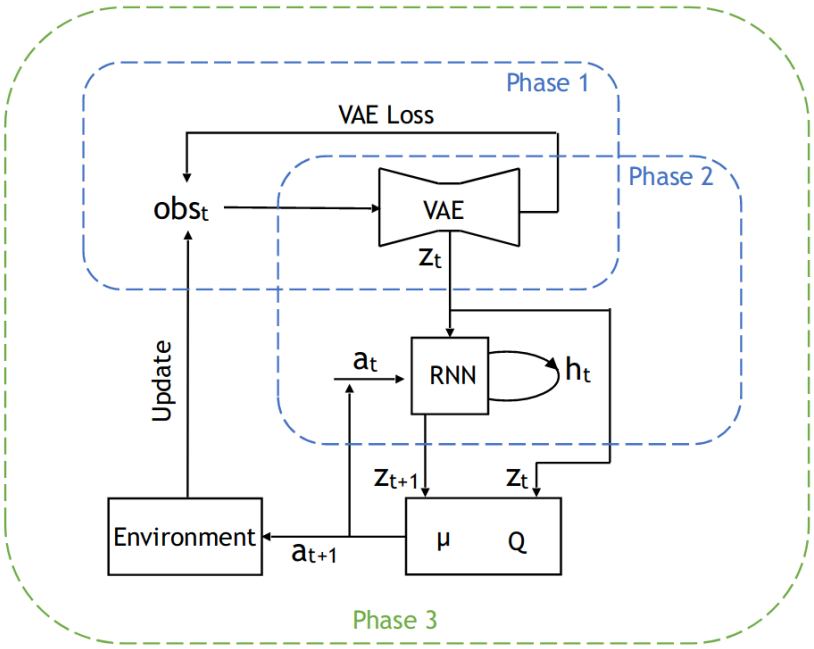

Figure 3.4: Overview of the model showing how the VAE, RNN and DDPG interact [8]

Their system was trained and tested in the CARLA simulation environment. For camera-input, they used the built-in semantic segmentation in CARLA. They tested two different reward structures, in the first reward structure, they simply counted the number of people involved in the collision. While in the second they used empirical models of injury equations that had been developed and tested by other researchers, thus accounting for accident severity. Both the first and the second reward structure showed a significant increase in the number of collisions avoided, and a lower incidence of severe injury, when using the steer and brake system compared to the baseline where only straight braking was allowed. The second reward structure where accident severity was taken into account showed an improvement in all areas and scenarios and in some cases they saw an improvement by 60% compared to the baseline [8].

In summary, both model-based and model-free reinforcement learning for collision avoidance have been researched and shown to be useful. However it is difficult to tell which method has been the most successful at collision avoidance, and what the benefits and drawbacks of these methods are. Most of these studies have been done in widely different environments, with different tests and challenges, making it hard to compare the two methods. Literature where model-based and model-free

RL for collision avoidance is compared seem to be lacking.

3.4 Current state of the art in autonomous driving

The main reason it is important to study collision avoidance, is to solve the overall problem of autonomous driving. Therefore a short description of state of the art in autonomous driving is given here. Companies like Tesla are getting closer and closer to making autonomous driving possible. All the cars they produce comes with full self-driving hardware, and the software that will control the car will be released and enabled in an update for all drivers once Teslas autonomous driving system has been fully calibrated and deemed safe. Currently the Tesla Autopilot system already has lane centering, adaptive cruise control, self-parking, the ability to automatically change lanes, and the ability to summon the car to and from a garage or parking spot. Using Radar, an array of cameras around their cars and neural networks, they believe they will be able to make a fully self driving car [1]. There are several car companies that are trying to develop self driving cars, among them is BMW, Uber, Ford, Volvo and Waymo. Waymo, which originated as a google project, is one of the companies that are advancing very fast, and already offers a commercial self-driving taxi service in Phoenix, Arizona [2].

Waymo uses a deep recurrent neural network (RNN) named ChauffeurNet, that is trained to emit a driving trajectory by observing a mid-level representation of the scene as an input. A mid-level representation does not directly use raw sensor data, thereby factoring out the perception task, and allows them to combine real and simulated data for easier transfer learning. In addition they employ a separate PerceptionRNN that iteratively predicts the future of other moving objects in the environment, and this network shares features with the RNN that predicts the driving [3]. Researchers behind ChaffeurNet believes that thorough exploration of rare and difficult scenarios in simulation, within a reinforcement learning framework, will be the key to improving performance of their models, especially for highly interactive scenarios.

Urban autonomous driving is challenging due to complex road geometry and multiagent interactions. Current decision making methods are mostly manually designing the driving policy, which might result in sub-optimal solutions and is expensive to develop, generalize and maintain at scale. However with reinforcement learning, a policy can be learned and improved automatically without any manual designs. Recently researchers proposed a framework to enable the use of model free deep reinforcement learning in challenging urban autonomous driving scenarios. They designed a specific input representation and used visual encoding to capture the low dimensional latent states. They then applied state of the art model free deep RL algorithms (DDQN, TD3, SAC) into their framework. Their method was evaluated in a challenging roundabout task with dense surrounding vehicles in a high definition driving simulator, and the results showed that their method was able to solve the given task of driving in a

roundabout [41].

All of this research into autonomous driving from large companies shows that a future where autonomous cars are a reality is getting closer and closer. However there are a lot of justified skepticism in letting a computer control your car, therefore its important to focus on collision avoidance and safety if we are ever going to make a society with autonomous cars a reality.

Chapter 4

Method

In this chapter two reinforcement learning algorithms for predicting and avoiding collisions in traffic will be adapted using the neural network frameworks Keras and Tensorflow. One will be a model-based reinforcement learning algorithm, where an explicit internal model of the physical system of cars is learned. And one will be a model-free reinforcement learning algorithm, where learning an explicit model is avoided and only mappings from video images to actions are learned. They will be trained, tested and evaluated in a simulation environment, where performance, training time and various costs and benefits will be compared.

4.1 Scope and Assumptions

To maintain a manageable scope, this thesis will only focus on learning to avoid collisions in traffic situations. Other parts of autonomous driving, like learning to follow traffic rules, reading signs and staying in lanes will not be taken into account when setting up this experiment. The experiments will be done in a simulation environment as using real cars and real traffic situations would be very dangerous and very expensive. Collecting available driving data online, like dashboard footage, is possible but very tedious, and the quality of the data is varied. At the same time there are lots of useful simulation environments with consistent data quality which are easy to use, for example the Carla simulation environment. Therefore using a simulation environment for learning collision avoidance is chosen for this experiment. The neural network algorithms used in the model-based and model-free systems have been chosen because they have a fairly large amount of research behind them, and can be considered established neural network algorithms. A lot of this research has been done in simpler environments, such as 2D Atari games. By adapting, testing and comparing them in a more complex environment, new insights into model-based and model-free RL can be gained and may help give an indication of where further research in this field is necessary.

4.2 Simulation environment

To train and test the reinforcement learning algorithms that will be used for predicting collisions, a simulation environment is necessary. In this simulation environment the vehicles should be equipped with a dashboard camera, and the environment should be as realistic as possible so that the algorithms are applicable to the real world as well. CARLA is a good candidate for simulation. CARLA is an open source car driving simulator built upon Unreal Engine 4. It has a fairly realistic urban driving environment, with diverse weather and lighting conditions, user defined camera poses, fairly large user base and a well documented API [9]. Carla simulator have also been used in several research papers [8][42][43][44]. For this reason the Carla simulation environment was chosen for this experiment.

Figure 4.1: An image from the Carla simulator showing a complex and diverse junction

4.3 Setting up the agent and environment

The Carla simulator has several maps to choose from in which the experiment can take place. For this experiment Town03 was chosen. Town03 is a fairly good representation of a city, and provides complex driving scenarios with various junctions, roundabouts, neighborhoods and other interesting driving situations that requires a good collision avoidance policy. As well as having complex roads, there is also a diverse set of buildings and obstacles, like light-poles, trees, fences and other things one may find in a city. The agent controls a car of type Tesla Model S in this map, and this car is equipped with a camera sensor attached at the front bumper (see figure 4.2), that provides real time RGB images of what is happening in front of the car. It is also given a collision sensor, which signals every time the car experiences a collision of any type, as well as what it collided with. If the agent car takes a hard turn, the simulated suspension system of the car can make the chassis of the car hit the road, which sometimes

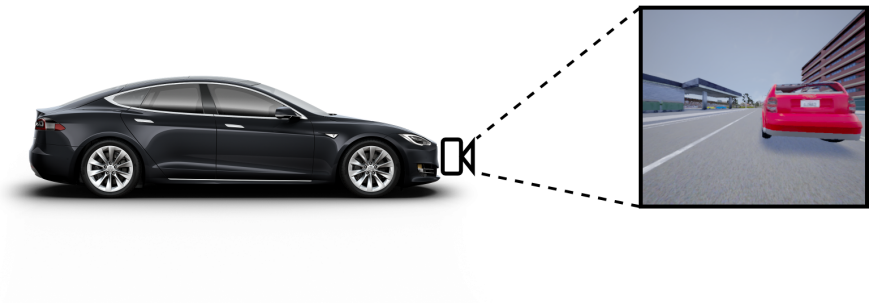

Figure 4.2: Camera attachment and car's viewpoint in the Carla simulator

triggers the collision sensor. This can also happen if the agent car takes a turn onto a sidewalk, making the chassis of the car hit the curb of the sidewalk. Since these collisions don't give any good visual indications of a collision, and since the sidewalk may often be necessary to drive on to avoid a collision, these types of collisions were filtered out using a simple if-test so they don't register as collisions. The agent is given an action space of 6 discrete actions, see table 4.1. In reinforcement learning the action space is typically either continuous or discrete depending on which reinforcement learning algorithm is used. When using Deep Q Learning its typically discrete, and making a DQN continuous would require significant modifications to the algorithm. OpenAI (2018) [45] found that using a discrete action space to control a hand with reinforcement learning worked much better than using a continuous action space. They theorized that a discrete probability distribution is more expressive than a multivariate Gaussian, or because discretization of actions makes learning a good policy potentially simpler. In Tang et al (2019) [46] researchers looked at the impact of discretizing continous control tasks using baseline reinforcement learning algorithms. They found that discretizing continuous control tasks improved the performance of baseline reinforcement learning algorithms, especially on high-dimensional tasks with complex dynamics. Therefore a discrete action space is chosen for this experiment. With a throttle and steering amount set at 1.0, the agent car would flip over when turning at high speeds, so throttling and steering was set to 0.75. When braking was set to 1.0, a brake meant an instant stop, so the braking amount was set to 0.5 to encourage the agent to use the brake as a way to lower its speed when necessary, not just instantly stop.

Action	Action ID	Throttle Amt	Brake Amt	Steer Amt
Turn left	0	0.75	0	0.75
Go forward	1	0.75	0	0
Turn right	2	0.75	0	0.75
Brake	3	0.0	0.5	0.0
Turn left and brake	4	0.0	0.5	0.75
Turn right and brake	5	0.0	0.5	0.75

Table 4.1: Agent action space

4.3.1 Training episodes

During training, the agent is spawned at random locations throughout the map. A large number of NPC vehicles (Non-Player Cars, meaning cars that are controlled by the simulator) are also randomly spawned in the map. These are controlled by the simulator, and drive around, simulating a populated city full of cars, motorcyclists and cyclists. There is also a random chance that a car will be spawned 25 meters in front of the agent cars spawnpoint. Before an actual training episode begins, the agent car will accelerate for 10 meters, then the episode will begin, and there will be a 50 percent chance that the agent will be on a collision course with a car spawned 15 meters in front of it. Manually spawning a car in front of it was done to ensure that the agent experienced enough collisions, since just spawning NPCs throughout the map didn't provide enough collisions for the car. Another solution to ensure that the car experienced enough collisions was to spawn a huge amount of NPC cars, however simulating several hundred cars driving around the simulator is computationally expensive and uses a lot of RAM. All of these cars potentially occupy a spawn point that the agent car can't use, which can result in the agent car not finding spawn points or spawning on top of NPC cars. Because of this a hybrid version of both spawning a moderate amount of NPC cars as well as manually spawning a car in front of the agents spawn point is used. To avoid the agent from simply learning that it always has to turn or brake when the episode starts, the car in front of it is randomly spawned with a 50 percent chance. If the agent learns to simply drive straight when no car has been spawned ahead of it, while also learning to make some maneuver to avoid a car when it gets spawned in front it, it may be a good indicator that the agent is able to see and understand when there is an obstacle in front of it and when there isn't. At the same time it also needs to learn to not crash into NPC cars and other obstacles, like buildings and trees, while trying to maneuver away from the car that was spawned in front it.

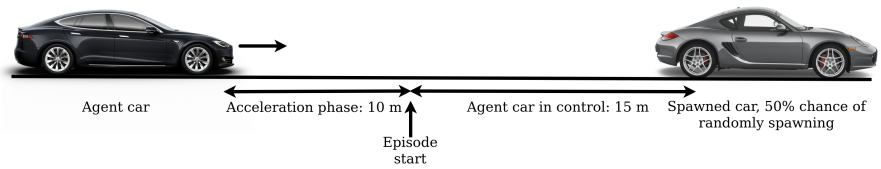

Figure 4.3: Diagram showing how a training episode begins. The agent car accelerates for 10 meters, then the episode starts with the agent car on a collision course with a spawned car.

An episode will last for about 10 seconds (acceleration phase not included), giving the agent enough time to experience some driving, where it has to make a few maneuvers to avoid collisions with objects or cars in the environment. The more time spent driving, the higher the likelihood of crashing into something, and the more objects the agent needs to maneuver around, often ending up in difficult situations. Since a collision is a terminal state, meaning that the episode ends, an agent may never experience an episode without a collision if the episodes are long. The agent may have made several good maneuvers throughout an episode, while still getting punished heavily for inevitably experiencing a collision. In these long episodes that always leads to collisions, it may be more difficult to know which actions led to a good reward and which led to a collision, since any and all actions it took during all episodes it has experienced has led to a collision. The agent is repeatedly taking actions, and repeatedly getting rewards for its actions. But it is difficult to associate particular actions (or combinations of actions) with particular rewards. This is known as the credit assignment problem, see section 2.6.3. Therefore episodes were kept fairly short, where it is likely that it will have to perform a few collision avoidance maneuvers while still being able to finish an episode without ending up in a collision.

4.3.2 Reward function

The reward function describes when an agent should get rewards. By getting positive or negative rewards for taking certain actions that completes or fails at a task in the environment, the RL agent should learn a policy resulting in good actions by trying to maximize its reward. To learn a good collision avoidance policy, there should be a significant negative reward (punishment) for being involved in a collision. A collision of any type will be a terminal state, ending the episode. However if that is the only reward, the agent can simply learn to instantly brake and never actually drive around, thus solving the problem of avoiding collisions. Therefore a positive reward for driving should be given, which should encourage the agent to actually drive and not just brake. Initial testing showed that the agent still preferred to just brake and stand still, even when it had a clear path with no obstacles in front of it. Therefore, to further encourage driving, a negative reward for standing still is given. With these considerations,

a reward of -1 is given for a collision and episode ends, a reward of +0.01 is given for driving 1 meter, and a reward of -0.01 is given for standing still. See pseudocode below of the reward function.

```
if collision == True:
    reward = -1
    end_episode = True
if distance >= 1 meter:
    reward = 0.01
if speed == 0:
    reward = -0.01
if time == length_of_episode:
    end_episode = True
```

10

4.3.3 Exploration vs Exploitation

During training its important to balance exploration and exploitation. For this, epsilon greedy with decay is used, where a decaying epsilon parameter is used to give the probability of taking a random action or an optimal action (according to the network). With this policy there is a very high probability of taking a random action in the beginning, so that the agent gets to explore its action space and the environment early on, and then focus on exploitation later, see section 2.6.2. During training epsilon was set to 1, and the decay rate was set to 0.9975, giving a fairly heavy focus on exploitation. While a systematic parameter tuning study was out of the scope of this work, initial experiments showed that these parameters gave a reasonably good balance between exploration and exploitation. See equation 4.1.

$$a_t = \begin{cases} a_t & with\ probability\ 1 - \epsilon * 0.9975 \\ random\ action & with\ probability\ \epsilon * 0.9975 \end{cases} \tag{4.1}$$

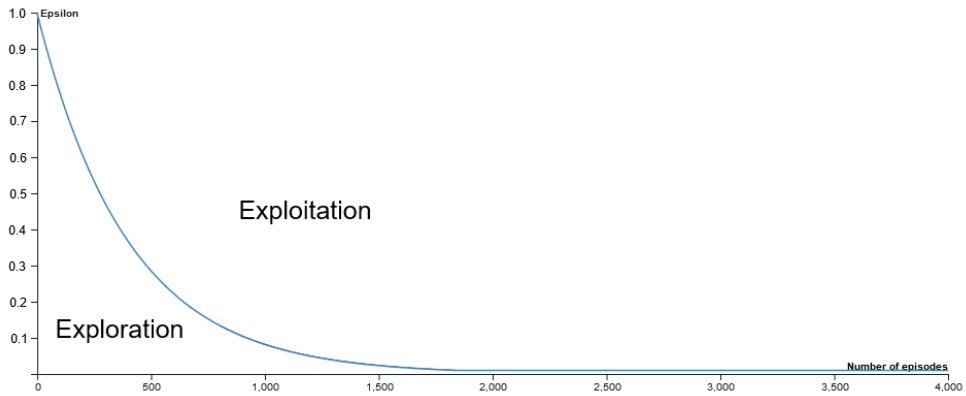

Figure 4.4: Epsilon greedy with decay. Epsilon = 1, decay = 0.9975, minimum epsilon = 0.001.

4.4 Model-Free Reinforcement Learning

4.4.1 DDQN

The first reinforcement learning method for collision avoidance to be tested is model free. So there is no explicit model of the environment, and instead a direct relation between RGB input images and what actions to take is learned. For this a Double Deep Q Network (DDQN) with experience replay is used. DDQN is a derivative of Deep Q Networks (see section 2.9 and 2.9.1) that uses two neural networks to decouple action selection and action evaluation. This reduces overestimation's of action values, helps the network train faster and leads to more stable learning [34].

4.4.2 Experience Replay

With experience replay, the agent's experiences at each time step are stored, where the agents experience at time t is represented as e_t.

$$e_t = (s_t, a_t, r_t, s_{t+1}) \tag{4.2}$$

Where s_t is the state of the environment, a_t is the action taken from state s_t, r_t is the reward and s_{t+1} is the next state of the environment. In this way the agents experience is summarized in e_t. These experiences are stored in replay memory, which is usually set to some finite size limit. These stored experiences are randomly sampled to train the network. The advantage of using experience replay is that each step of experience can potentially be used in many weight updates which results in greater data efficiency. It also helps against unwanted feedback loops. For example if the maximizing action is to turn to the left, then the next training samples will be dominated by that behavior, and if the maximizing action switches to the right, then the training distribution will also switch. By randomly sampling past experiences, the behavior distribution is averaged and not dominated by the current behavior. Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples. Randomly sampling from the replay memory bank breaks these correlations and therefore reduces the variance of the updates [35].

4.4.3 Parameters and Training

To learn a relation between RGB input images and what actions to take, convolutional layers will be used for the Double Deep Q Network. In the model-based section of this chapter, the paper 'World Models' by David Ha and Jurgen Schmidhuber [40](see section 3.2.1 for a description of their method) will be adapted to the problem of collision avoidance with model-based reinforcement learning. Here a Variational Autoendoer (VAE, 2.4.1) in combination with a Mixture-Density Recurrent Neural Network (MD-RNN, 2.5.2) is used to create a model of the environment. The VAE uses a CNN for the encoder network to change the RGB images to a latent vector, giving a compressed representation of the world. Since this

thesis is interested in comparing model-based RL with model-free RL, a similar CNN to the one used in the model-based section of this chapter, will be used for the model-free RL method. This may give a more fair comparison of the two techniques, as they will share similar parameters and preconditions for learning representations of the world. Figure 4.5 shows the network parameters and architecture used in the model-free RL. This architecture is similar to OpenAIs breakthrough paper Mnih et al. (2013) [35], where convolutional layers with the relu activation function is used to process input images, which is then sent to fully connected layers, with a final output layer in which there is a separate output unit for each possible action. Using this network, the agent is trained for 4000 episodes, with an episode length of 10 seconds and a decay rate of 0.9975, making the agents first 1000 episodes focused on exploration. The remaining 3000 episodes are focused on exploitation, giving a heavy focus on exploitation during training. See 4.2 for a list of hyperparams used during training.

Hyperparameter	Value
Episodes	4000
Image width	128
Image height	128
Episode length	10 seconds
Replay memory size	4000
Minibatch size	32
Update target every	10 episodes
Discount	0.99
Epsilon decay	0.9975
Min epsilon	0.001
Optimizer	Adam
Learning rate	0.0001
Loss	SGD

Table 4.2: Hyperparameters used during training of the model-free RL algorithm

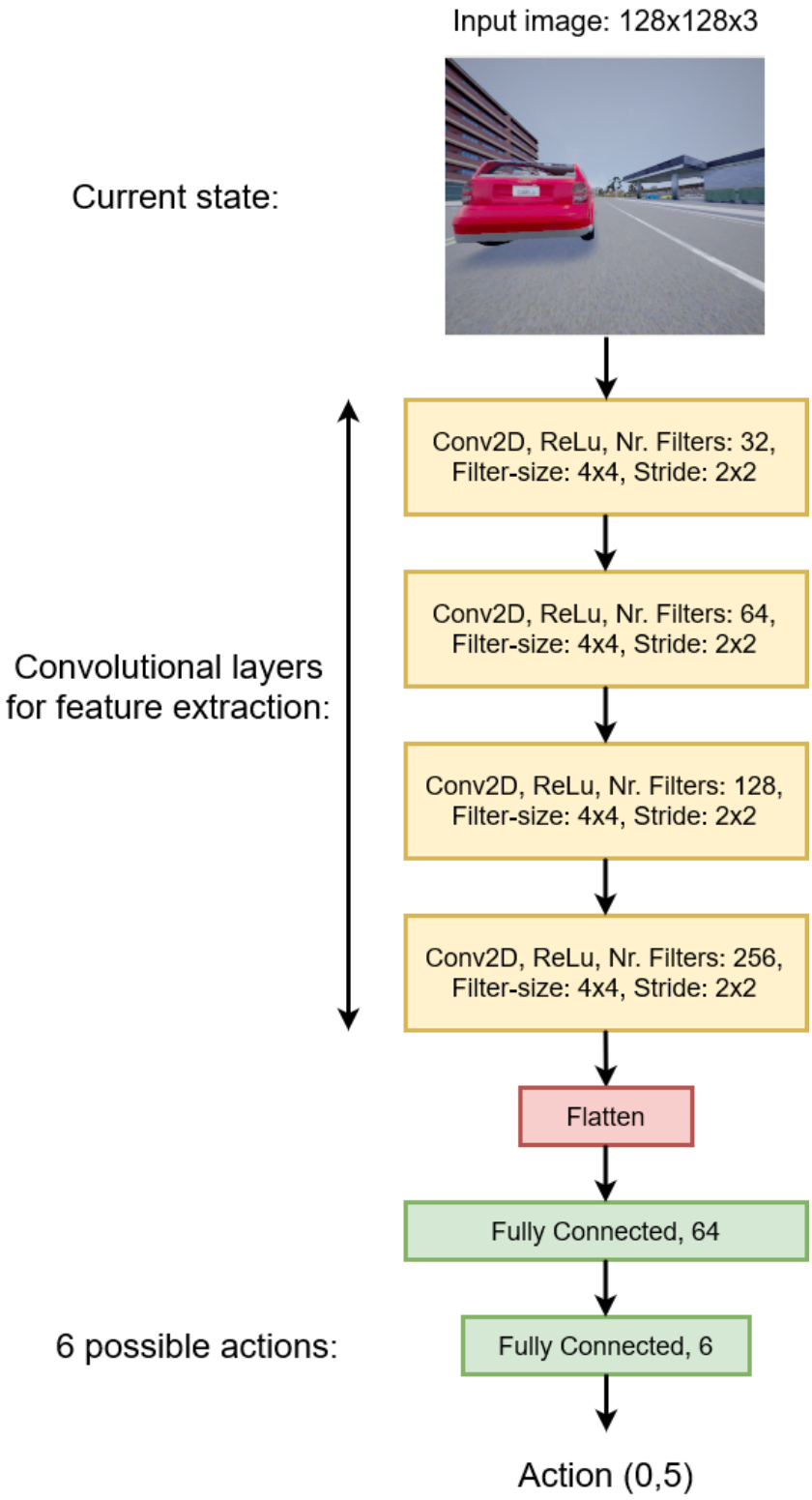

Figure 4.5: Model-Free network architecture

4.5 Model-Based Reinforcement Learning

In the model-based section of this thesis, the paper 'World Models' by David Ha and Jurgen Schmidhuber [40] is adapted to the problem of collision avoidance with model-based reinforcement learning in the Carla environment. Their method achieved breaktrough scores in both a 2D and 3D environment. In the 2D environment the task was to steer a car along a randomly generated path from a birds-eye view. In the 3D environment the task was to dodge incoming fireballs, from a first-person view. Showing that the method used is good at learning to steer along quickly changing paths, and also dodging incoming objects, which are both fairly similar tasks to the problem of collision avoidance. Therefore this method may be well suited to the problem of collision avoidance. However the Carla environment is far more complex than the environments used in the original paper. Here there is a huge diversity of obstacles that needs to be learned, and the roads and surroundings are always changing throughout the city. Steering is also more complex, as movement is not instant, since the agent car has momentum and complex physics that are meant to be realistic. In this method, the RL agent has a visual component that compresses what it sees into a small representative code, using a VAE trained on image data from the Carla environment. The RL agent also has a memory component that makes predictions about future codes using an MD-RNN trained on sequences of driving in the Carla environment. Lastly there is a decision making component consisting of a DDQN that decides what actions to take based only on the representations created by its vision and memory components.

4.5.1 Variational Autoencoder (Vision)

The role of the VAE is to learn an abstract compressed representation of the input images from the agents camera, acting as the agents real time vision. See section 2.4.1 for a description of VAEs. The VAE is given 128x128x3 images from the agents camera sensor, which is then encoded, and the resulting latent vector z of size 64 is given to the agent as its vision in real time.

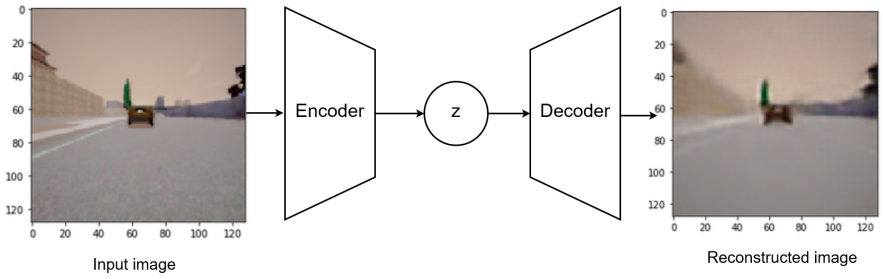

Figure 4.6: Flowchart of a VAE, where a 128x128x3 image is encoded to a latent vector z of size 64, and then reconstructed back to a 128x128x3 image using the decoder.

This latent vector z may make it easier for the agent to understand the world, since various objects, roads, buildings etc will have a simple representation in the vector of only a few values. While in an RGB image of size 128x128x3 there are 49152 parameters representing the world, which should be more difficult to learn. However during the encoding we risk losing some data, which may be important for collision avoidance. For example smaller details like a light pole may be lost, which in a 128x128 image may only be represented by a thin line of 2-3 pixels, depending on the distance from the light pole. Parameters for the VAE will be based on the VAE used in World Models by David Ha and Jurgen Schmidhuber [40]. Figure 4.7 shows the parameters and architecture of the VAE that is used.

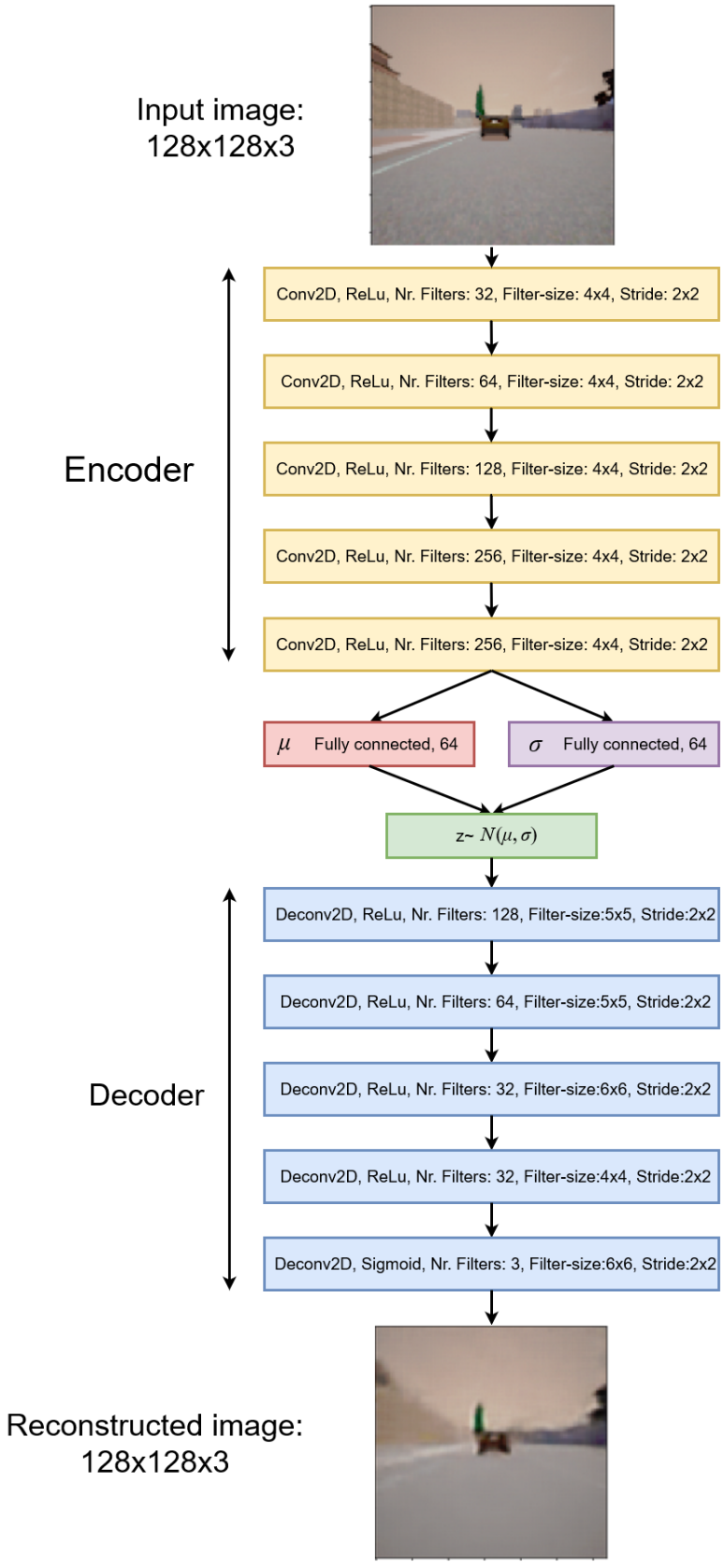

Figure 4.7: VAE network architecture

Dataset and Training

When training the model-free RL algorithm its not necessary to create a dataset because the network trains and learns to interpret images while the agent interacts with and observes the environment. In principle VAE and MDN-RNN could be trained in an end-to end manner, learning while the agent interacts with the environment. However the original writers of World Models [40] recommends against it, as they found that training each module separately is more practical, achieves satisfactory results and does not require exhaustive hyper parameter tuning. Therefore it was chosen to train each module separately making a dataset for the VAE necessary.

Image data of the Carla environment is collected by having the agent car controlled by the simulators inbuilt autopilot collect images of the environment as it drives around the environment. A total of 400 000 images was collected. After significant training and testing, the VAE seemed to struggle with reconstructing smaller objects, like cars, light poles and signs. At the same time it got really good at reconstructing parts of the environment that is always present, like roads, the sky and buildings, even being able to reconstruct the windows of buildings. To solve this problem the maximum amount of simulated cars was spawned in the environment, to make sure that cars are well represented in the dataset. However this did not help. The reason the VAE struggles with reconstructing cars may be because cars and smaller objects take up a very small portion of the image, while 90% or more of the image consists of the road, the sky and surrounding buildings. So during training as long as it gets good at reconstructing these parts, a good loss is still achieved, since its already able to reconstruct 90% of the image.

To solve this the loss function is weighted to encourage reconstructions of cars and other objects. This is done by collecting a segmented example of every RGB image, creating a dataset that contained both RGB images and segmented images. Segmented images is acquired by using a built in segmentation camera sensor in the Carla simulator environment. The segmented image is an image where each object in the image is represented by a value between 0-13. This segmented image is used to create a weight mask, where each object is given a weight. See figure 4.5.1. This weight mask is then multiplied with the reconstruction loss in the VAE during training. After training the weight mask is not used when reconstructing images. This gave a significant improvement when it came to reconstructing cars. See example figure 4.5.1.

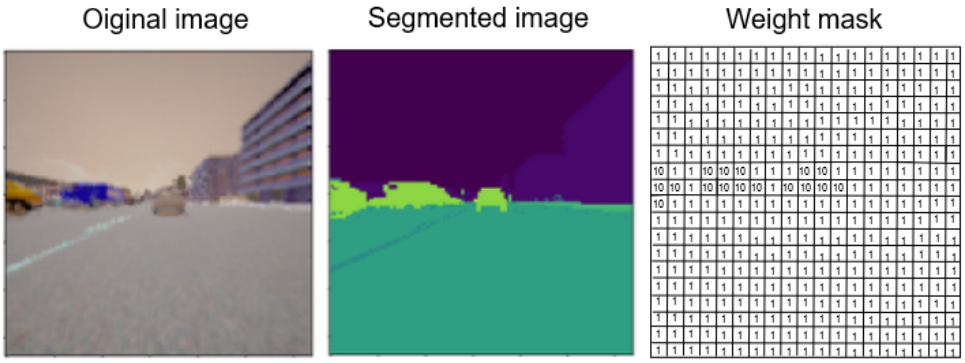

Figure 4.8: Example of a weight mask. Cars are given a weight of 10, while the road, buildings and the sky is given a weight of 1.

Figure 4.9: Left shows the original RGB image, middle shows a reconstructed image using a VAE model that is trained without a weight mask, and right shows a reconstructed image using a VAE model that is trained with a weight mask.

Hyperparameter	Value
z-size	64
batch size	100
Nr. Epochs	50
Learning rate	0.0001
KL-tolerance	0.5
Loss	MSE

Table 4.3: Hyperparameters used during training of the VAE

4.5.2 Mixture Density RNN (Memory)

While the VAE compresses what the agent sees in real time, the role of the MD-RNN is to predict future z-vectors that the VAE is expected to produce. Here a LSTM Recurrent Neural Network combined with a Mixture-Density network is used. See section 2.7 and 2.5.2. P(z) is approximated as a mixture of Gaussian distributions, and the MD-RNN is trained to output the probability distribution of the next latent vector z_{t+1}. The MD-RNN models $P(z_{t+1}|a_t, z_t, h_t)$ where a_t is the action taken at time t, z_t is the encoded latent vector of what the agent sees at time t, and h_t is the hidden state of the MD-RNN at time t. The hidden states of the MD-RNN are sent to the agent, giving the agent a compressed model of the temporal dynamics of the world, a form of understanding of what will happen in the future.

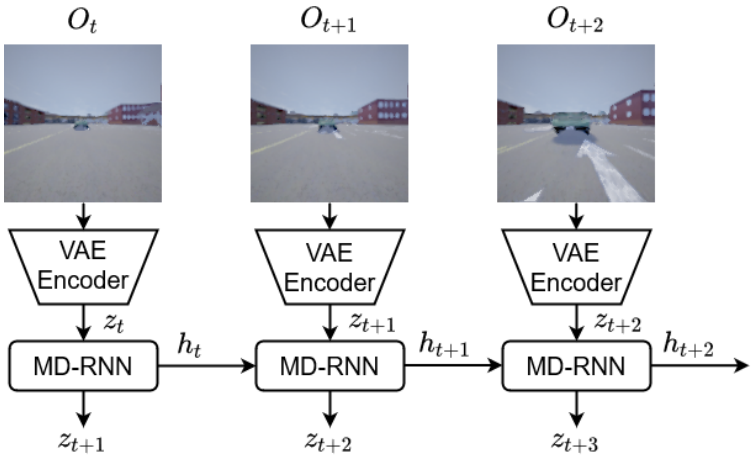

Figure 4.10: Image showing how MD-RNN predict future latent vectors

Dataset and training

To train the MD-RNN a dataset is needed. The dataset needs to contain both image sequences of the agent driving in the environment, as well as the actions used during driving. This means that Carlas autopilot can't be used, since its action space is continuous and doesn't follow the discrete action space that the RL agent uses, see action space here 4.1. Therefore the trained model-free RL agent was used to create a dataset for the MD-RNN. This agent does not drive as well as Carlas autopilot, but it does have a driving policy that is good enough to show what movement and driving looks like, while also using the same action space that the model-based agent will use. When creating the dataset, a collision is no longer considered a terminal state. The agent is allowed to drive for 20 seconds, where it collects 200 images and 200 actions taken at that image, giving each driving sequence an FPS of 10. The FPS is low because its unnecessary to train on sequences of images where there is very little difference or change happening. 2500 episodes is collected giving a total of 500 000 images and

actions. The MD-RNN learns to predict future latent vectors, therefore the images where encoded to latent vectors, creating a dataset of 500 000 latent vectors and related actions.

Hyperparameter	Value	Description
Training steps	40000	
Sequence length	200	Length of sequences to train on
Input sequence width	65	Latent-vector + action
Output sequence width	64	
RNN size	512	Number of RNN cells
Batch size	200	Size of batches to train on
Gradient clipping	1	Use gradient clipping
Nr. Mixtures	8	Nr. of mixtures in MD
Learning rate	0.001	
Layer norm	1	Use layer normalisation

Table 4.4: Hyperparameters used during training of the MD-RNN.

4.5.3 Reinforcement Learning (Control)

The role of the decision making component is to determine the best course of actions to take in order to maximize the expected cumulative reward of the agent during an episode. For the decision making component of the model based reinforcement learning system, a DDQN is used. In the original paper, 'World Models' by David Ha and Jurgen Schmidhuber[40], they used Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) as their controller. This technique requires a lot of CPU cores, which wasn't an available resource for this thesis, however there was access to a strong GPU. Therefore a DDQN was used instead of CMA-ES, since DDQNs are more well suited to training on GPUs. This also gives a fair and similar comparison with the model-free RL algorithm, which also used a DDQN. The VAE encodes real time images into latent vectors, which is concatenated with the hidden states of the MD-RNN, which makes real time predictions on future latent vectors. This concatenated vector of encoded latent vectors and hidden states is given to the DDQN as input. The DDQN uses these components to learn a collision avoidance policy. Both the VAE and the MD-RNN is pre-trained in the Carla environment. The encoder in the VAE outputs a vector of size 64, and the MD-RNN has 512 hidden states. These are concatenated to form a vector of size 576, which are then normalized between 0 and 1. Most of the complexity should lie in the Vision and Memory component. Therefore the DDQN is given a simple two layer fully connected network. One input layer of size 576 and one output layer of size 6. This is similar to the controller component in World Models[40], where they used a simple single-layer linear model that maps z_t and h_t directly to action a_t at each time step. Table 4.5 shows the parameters used during training of the full model-based RL algorithm.

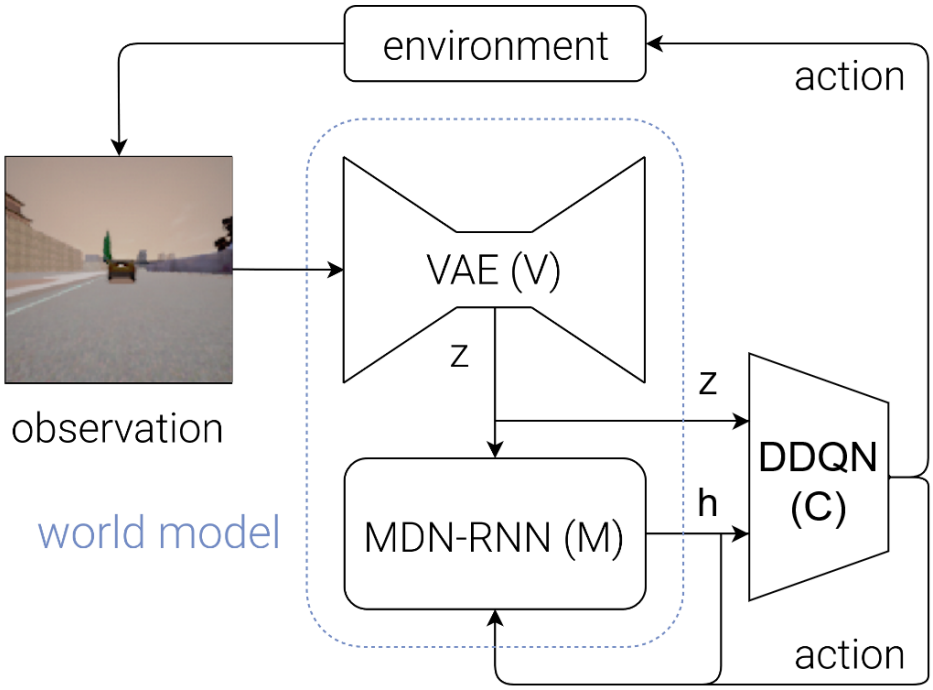

Figure 4.11: Flow diagram of the model-based agent adapted from World Models, [40]. Real time image observations are encoded by V at each time step t to produce z_t. The latent vector z_t is concatenated with the hidden state of M, h_t at each time step. C outputs an action a_t to control the car. M takes the current a_t and z_t to update its hidden state, producing h_{t+1}, used at $t+1$

Hyperparameter	Value
Episodes	4000
Image width	128
Image height	128
Episode length	10 seconds
Replay memory size	4000
Minibatch size	32
Update target every	10 episodes
Discount	0.99
Epsilon decay	0.9975
Min epsilon	0.001
Optimizer	Adam
Learning rate	0.0001
Loss	SGD

Table 4.5: Hyperparameters used during training of the model-based RL algorithm

Chapter 5

Results and Analysis

In this chapter the trained model-free RL system and the trained model-based RL system is tested and compared. We begin by looking at training results, and do a qualitative visual analysis of the systems ability to avoid collisions. Afterwards, the main experiment will quantativly test the collision avoidance policy of the various models. Then the components of the model-based RL system, the Variational Autoencoder (VAE) and the Mixture-Density Recurrent Neural Network (MD-RNN) are looked at to get a better understanding of their impact on the model-based RL system. To gain further insights into the results a variation of the two systems will also be tested, where the memory component in the model-based system is used to predict future images, which is then used by the model-free RL system. An additional experiment will be performed to test the models ability to adapt to a new task. Lastly the costs and benefits of training between the model-free and model-based RL agent is compared, looking at data efficiency and computational cost.

5.1 Training Results

5 models of each system has been trained to get more reliable results during testing. Figure 5.1 shows the median, max and min for reward and collision rate during training, for the 5 trained model-free systems and the 5 trained model-based systems. In the graphs showing rewards during training, higher rewards is better. In the graph showing collision rate during training, a lower collision rate is better, where a collision rate of 1 means that the agent car crashed 100% of the time. In the beginning of training, random actions is taken for exploration. This leads to the agent mostly standing still or barely driving in the beginning, since all actions are being selected at random. This results in a low collision rate early on, as well as negative rewards for standing still. Over time the agent is given more and more control over the car, which allows it to drive longer distances as it learns how to control the car, but this also leads to the collision rate going up, since its now driving longer distances and getting into more risky situations. This explains why the collision rate seem to increase over time, while rewards also are increasing. From the plots we

can see that the model-free systems reach a median collision rate of about 0.6-0.7 while the model-based systems reach a median collision rate of about 0.8-0.9 during training.

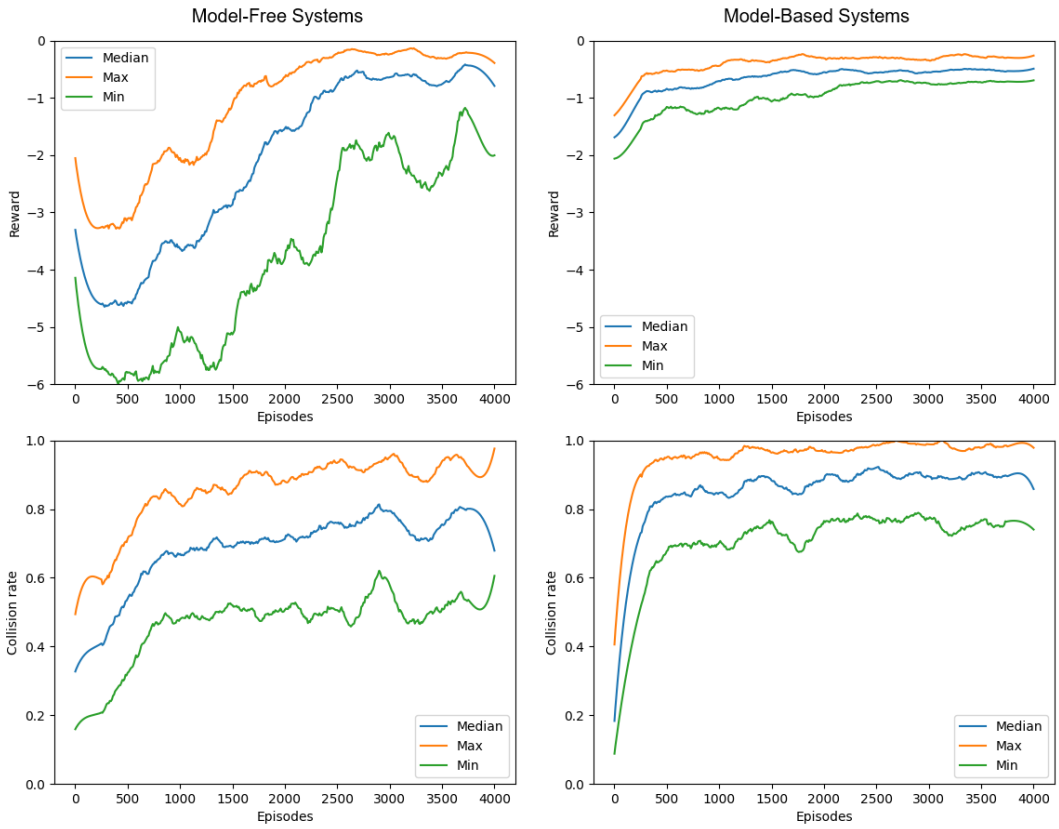

Figure 5.1: Plot showing median, max and min for reward and collision rate for the 5 trained model-free systems, and the 5 trained model-based systems. Plots have been smoothed with Savitsky Golay filter [47].

5.2 Visual Analysis

We can view the agents driving in the environment while they are training and after training. This allows us to do a qualitative analysis of their collision avoidance policy. This is not a replacement for quantitative data, which will be looked at in the next section, but is a useful addition to the measurements and the quantitative data that comes next. By viewing how all the trained models are driving, there are some patterns that can be observed. The trained model-free agents tend to learn a fairly good collision avoidance policy, where they regularly make clear avoidance maneuvers to avoid cars in their path. Although collisions still regularly happen. They also tend to learn a driving policy where they drive in a zigzag pattern until they eventually start driving straight again if the path is clear. This may be the agents thinking they are on a collision course with the surrounding buildings and walls, constantly making them correct their

path back and forth. Figure 5.2 shows images from a driving sequence performed by a model-free RL agent, showing its zig-zag driving pattern. In the first row we can see that the car is on a collision course with a white car. Then it turns to the left to avoid this car. In the second row we can see that the car is on a collision course with a wall, and here it turns to the right to avoid this wall, which puts it on a collision course with a black car. Then in the third row it turns to the left again to avoid the black car.

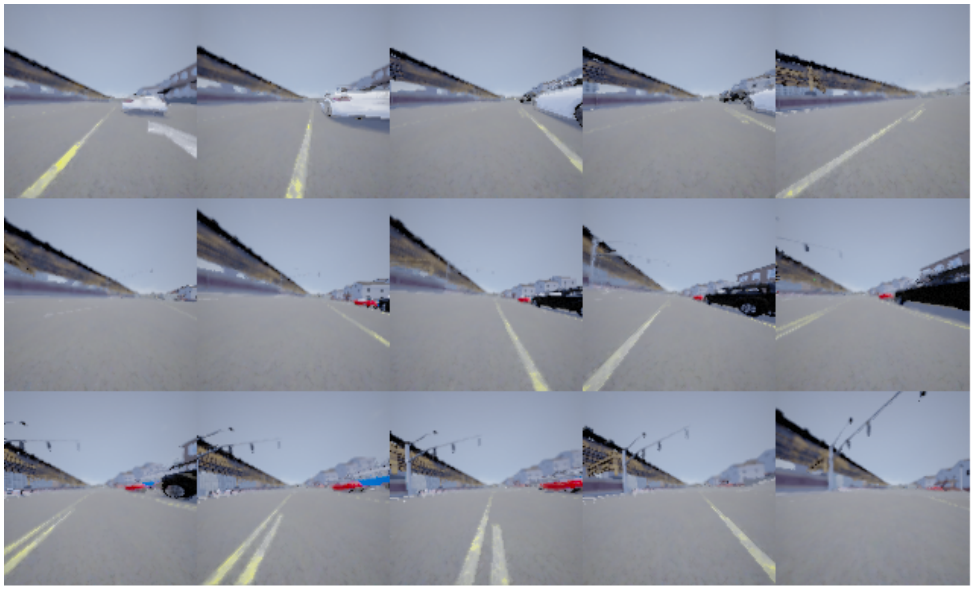

Figure 5.2: This sequence of images shows an example of a model-free RL agents driving in the environment.

Based on a visual analysis of the model-based agents, they did not seem to learn a good collision avoidance policy, and instead always learned to drive straight or constantly brake. When it attempts a collision avoidance maneuver by turning, its usually too late. Figure 5.3 shows an example of a model-based RL agent driving straight into another car without making any effort to avoid it. This might be the result of the network ending up in a local maximum. During training there was 75 cars spawned in the environment, as well as a 50% chance of a car getting manually spawned 25 meters in front of the agent car. This made it possible for the agent to sometimes drive completely straight for an entire episode without experiencing any collisions. If this happens, it gets the maximum amount of rewards possible. If its not able to interpret the model of the environment and not understand where obstacles are present, this might be the best policy, since it sometimes can result in the agent getting a high reward. This might also explain why the model-based systems seemed to have a high reward during training, and may point to problems with both the reward structure and the model of the environment. When attempting to spawn a larger amount of cars (100 cars, 75% of a car getting manually spawned 25 meters in front) it tended to learn to constantly break, thus avoiding the punishment for collisions.

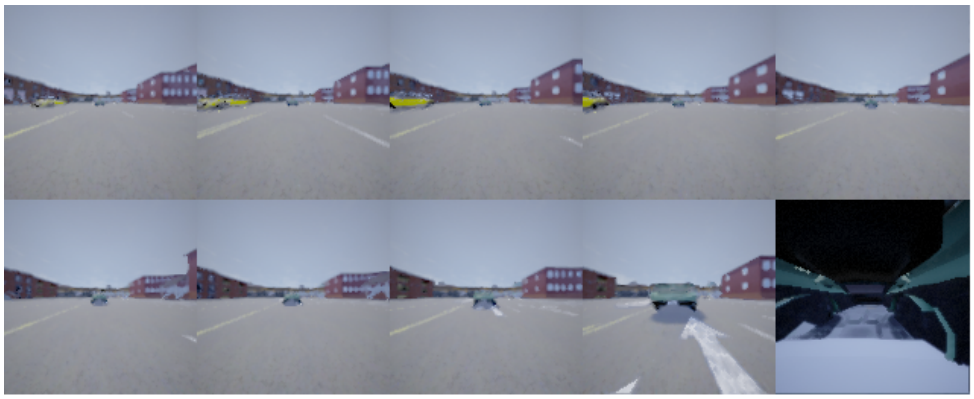

Figure 5.3: This sequence of images shows an example of a model-based RL agent driving in the environment.

Neither algorithms learned a collision avoidance policy that would ever be safe for use in the real world based on our visual analysis, however the model-free RL algorithm seemed to learn a significantly better collision avoidance policy. A video of their driving policies can be viewed, see footnote[1] for a video with examples of a model-free agent driving. In this video, we can see two 30 second episodes where the agent makes several collision avoidance maneuvers. See footnote[2] for a video with examples of a model-based agent driving. In the first episode, the agent is able to turn and avoid a building, but crashes in a small buss-stop shelter. In the next episodes, we can see examples of the agent car driving straight into other cars with no attempts to avoid them.

5.3 Testing the Collision Avoidance Policy

A visual analysis may give some insights into the collision avoidance policies of the two techniques. But to get statistically significant data, we need to do an experiment where we can collect and compare quantitative data. In this experiment we will test and compare the collision avoidance policy of the model-based RL algorithm and the model-free RL algorithm. To test their collision avoidance policy, they will get to control a car which is spawned at random locations in the Carla environment. They will be tested in town 03, which is the city they have been trained in. The city will be populated with vehicles that are controlled by the simulator. During training, cars were manually spawned 25 meters in front of the agent car. This will not be done during testing, which may result in the models having a lower collision rate than what we have seen during training. They will get to freely drive around in the environment for a given amount of time, where the goal is to drive the longest distance without crashing. The test episode ends if a collision happens or if the time runs out. During this time, the distance traveled, how long a test episode lasted and if a collision

[1] https://youtu.be/qb8wHsr5Yoc
[2] https://youtu.be/n72PD4t0q6k

happened or not will be measured. This experiment is repeated for 100 episodes for each trained model.

Deep reinforcement learning algorithms are sensitive to random seeds, the random nature of weight initialization in the network and stochasticity in the environment, which can lead to reproducibility issues, and differences in algorithm performance even when training the same model with the same hyperparameters [48]. Therefore training just one model of the model-free and model-based RL algorithms aren't enough to get conclusive results about the two techniques. The model-free RL algorithm and the model-based RL algorithm is trained and tested 5 times each to get more conclusive results about which technique reaches a better collision avoidance policy. All models are trained with the same hyperparameters as described in the Method chapter. 5 models of each technique isn't necessarily enough to reach conclusive results and for the results to be statistically significant, but because of the time consuming process of training and testing each model and there only being access to one machine learning computer, we decided to train 5 models of each, which is possible to do within a reasonable time frame. Training neural networks also has a significant environmental impact, and large industry standard training of a single AI network can emit as much CO2 as five cars in their lifetime [49]. Several highly cited papers on reinforcement learning have also used 5 trials when evaluating their models [50][51][52][53].

There is no simple way to measure collision avoidance maneuvers, as there is no sensor or input from the simulator that can specifically measure collision avoidance maneuvers and how good they are. One could use a human visual analysis of the collision avoidance policy, but that would be time consuming, tedious and prone to bias. Another way could be to use the simulators inbuilt radar and measure if the distance between the agent car and an object is decreasing, and then see if it manages to avoid it by checking if the distance between the object and the agent car increased or stopped decreasing. However this would give a lot of false positives whenever the agent car drove next to another car without being on a collision course with it, or if a car drove past it in the opposite lane, or in any other situation where it is not necessarily on a collision course, but just driving close to or nearby an object. By measuring distance, time spent driving and if a collision happened or not, we can get a fairly good indication of how good the models are at avoiding collisions. For example if the given test episode time is 30 seconds, then an ideal model would get an average time spent per episode of 30 seconds with 0% collisions and an average distance of 200-300 meters over 100 test episodes. Since a large number of vehicles will be spawned in the environment, there is a small chance that an agent car might accidentally get spawned on top of a simulated car during testing, resulting in an instant collision. If this happens, the test episode will be filtered out by checking if the episode length is below 2 seconds.

The models are tested in environments with increasing difficulty, defined as easy, medium and hard. This is because a too difficult environment may lead to all models reaching a collision rate of 100%,

making it difficult to gain any insight into which technique reaches the best collision avoidance policy. In the easy test, the environment is populated with 50 cars and the agents have to drive for 10 seconds before the episode ends. In the medium test, the environment is populated with 75 cars, and the agents have to drive for 20 seconds before the episode ends. In the hard test, the environment is populated with 100 cars, and the agents have to drive for 30 seconds before the episode ends. The results of these tests can be seen in the box-plots below, which compare distance and time for the models. The results comparing distance and time for the easy test can be seen in box-plot 5.4. The results for the medium test can seen in box-plot 5.5, and the results for the hard test can be seen in box-plot 5.6. Box-plot 5.7 compares the median collision rate between the 5 trained model-free systems, and the 5 trained model-based systems, for all difficulty settings. Tables showing additional results for each trained model for these tests can be found in Appendix, in section 7.1.

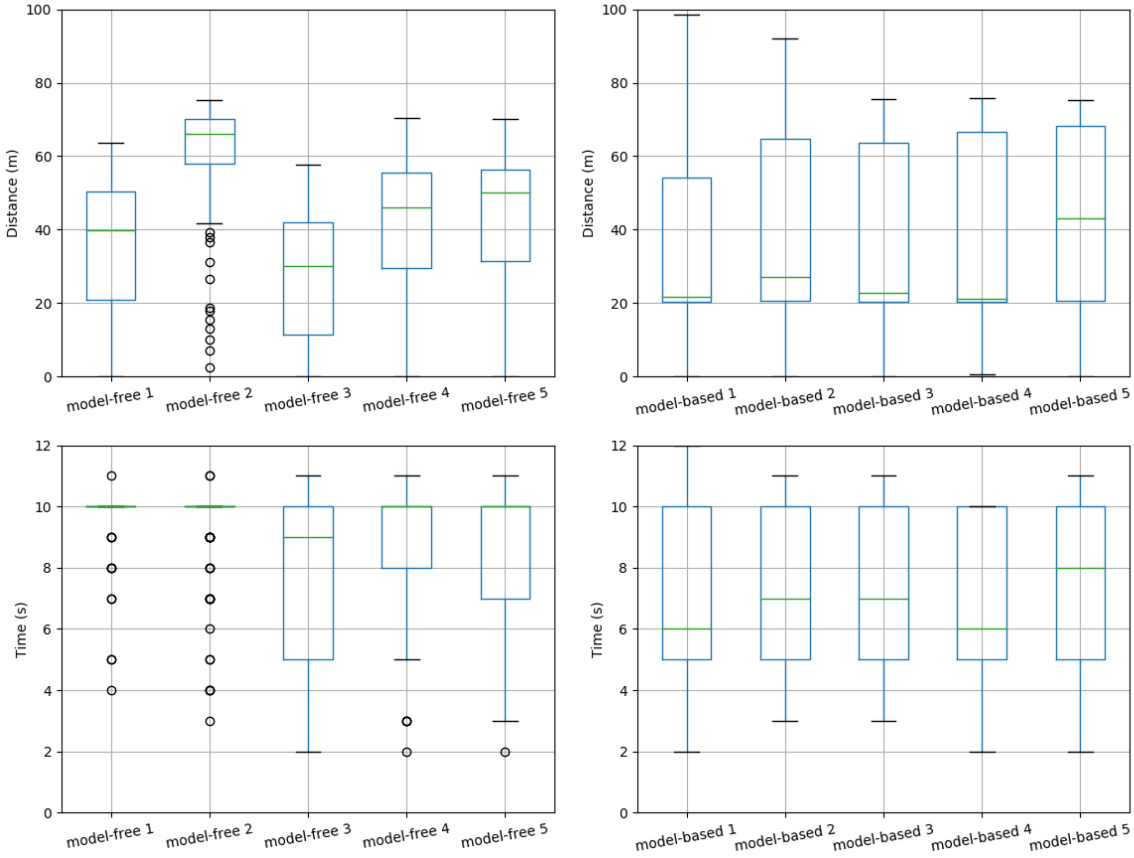

Figure 5.4: Box-plots comparing distance and time for the model-free systems and the model-based systems, tested in the easy environment for 100 episodes each. Green line is median, circles are outliers and the blue box represents the interquartile range (25th to the 75th percentile).

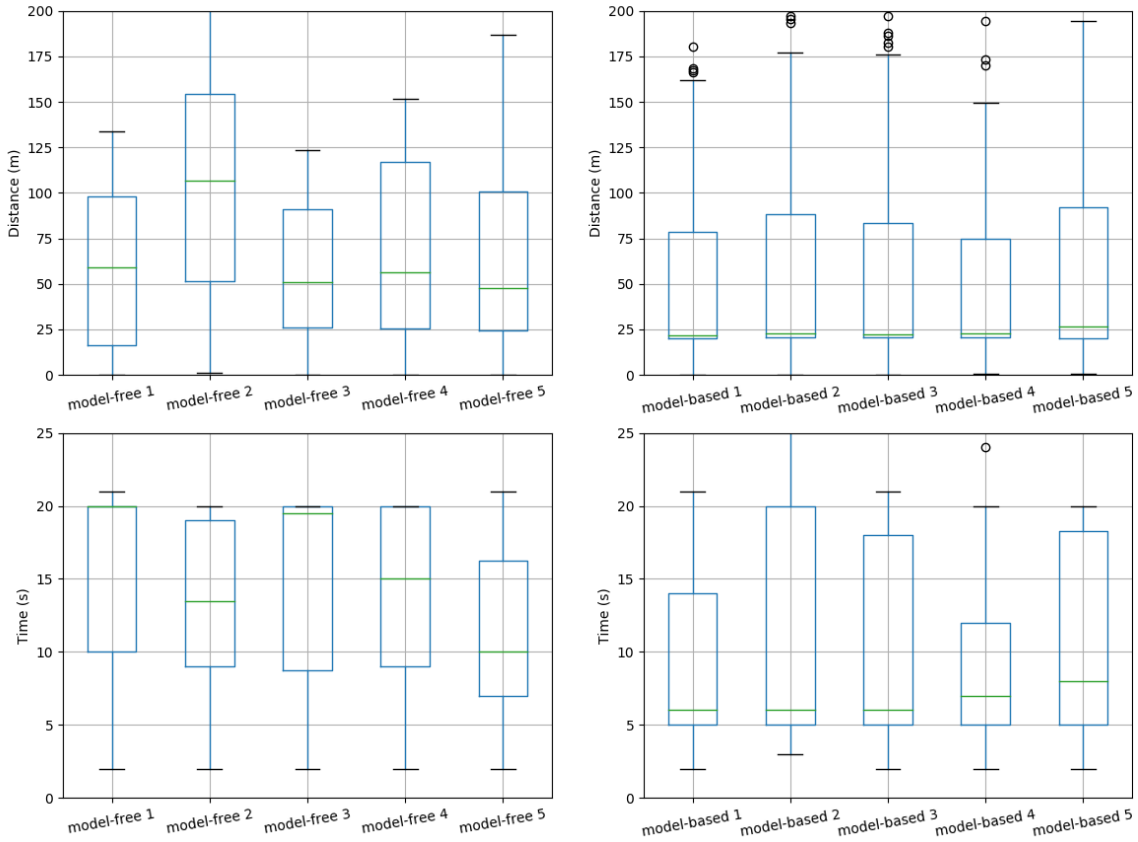

Figure 5.5: Box-plots comparing distance and time for the model-free systems and the model-based systems, tested in the medium environment for 100 episodes each.

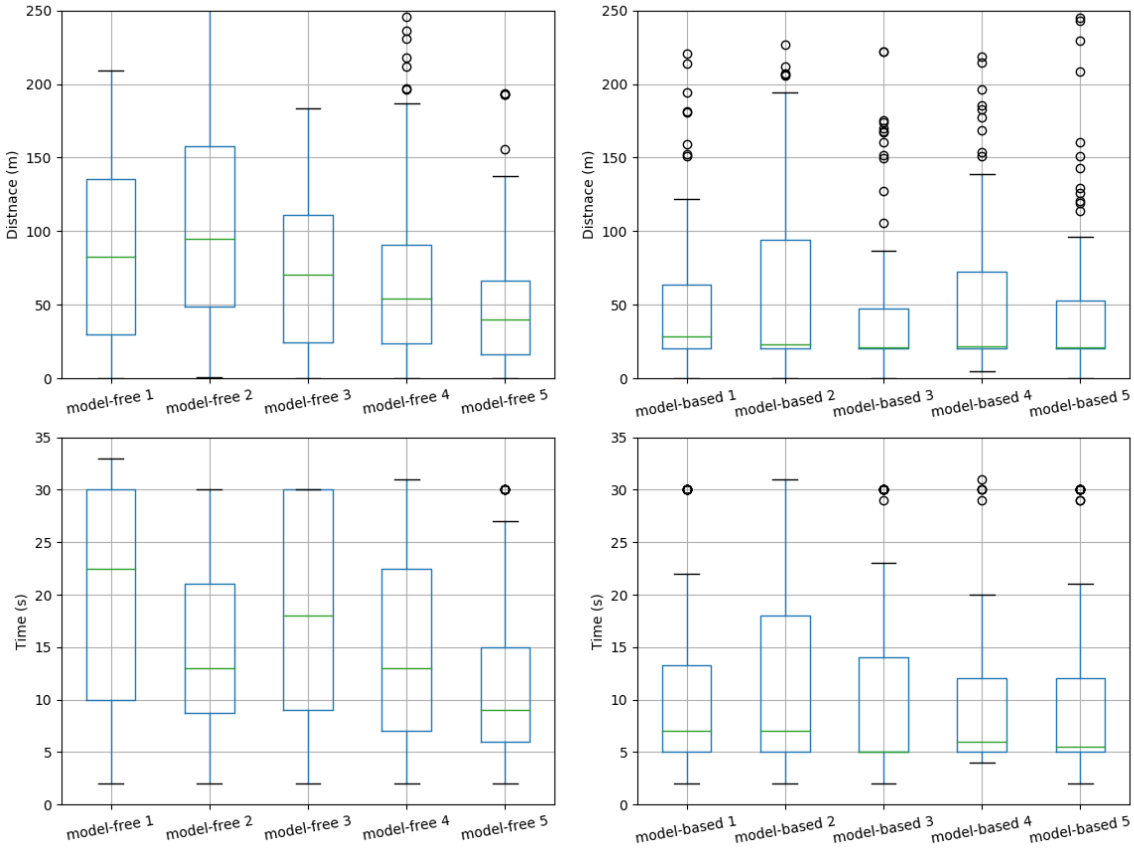

Figure 5.6: Box-plots comparing distance and time for the model-free systems and the model-based systems, tested in the hard environment for 100 episodes each.

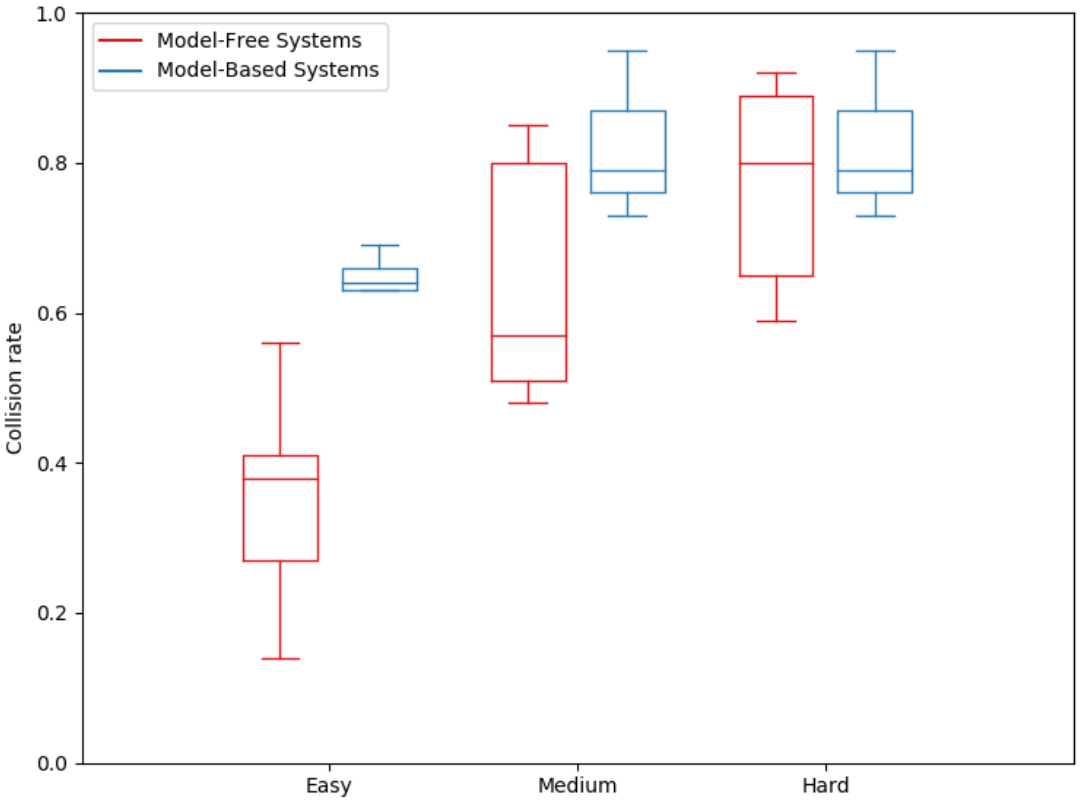

Figure 5.7: Box-plot comparing the median collision rate for all difficulties. Lower collision rate is better.

Based on these tests it seems that the model-free reinforcement learning algorithm learned a significantly better collision avoidance policy than the model-based reinforcement learning algorithm. In the easy test environment, the model-free systems got a mean collision rate of 35.2%, while the model-based systems got a mean collision rate of 63.6%. The best model-free system got a collision rate of only 14%, while the best model-based system got a collision rate of 56%, see table 7.1. The model-free systems also managed to drive for longer distances, and in 4 out of 5 models got a median distance double that of the model-based system, see box-plot 5.4. Looking at table 7.2, which shows the distribution of objects the cars crashed into, the model free systems seem to hit walls and buildings more often than vehicles. This suggests that they are good at avoiding vehicles in the environment, while possibly crashing into walls and buildings in their attempt to avoid cars. Meanwhile the model-based agents seem to mostly crash in vehicles. To make sure that the difference in collisions are statistically significant we can do a Mann-Whitney U test. This test is a non-parametric statistical significance test for determining whether two independent samples were drawn from a population with the same distribution [54]. When doing a Mann Whitney U test on the collisions between model-free systems and model-based systems for the

easy test environment, we get a p-value less than 0.001, which means that the Mann-Whitney U test found the difference in collision rate between the model-free and model-based RL algorithms to be statistically significant.

In the medium test environment, the model-free systems reached a mean collision rate of 64.2% while the model-based systems got a mean collision rate of 82.0%. Here the difference in collision rate is not as large as in the first test, but the model-free systems was still able to drive a longer distance, with a mean distance of 70.3 meters, and a mean time of 13.6 seconds compared to the model-based systems, which has a mean distance of 57.1 meters and a mean time of 10 seconds, see table 7.3. The box-plots in 5.5 shows that the median distance and the median time is significantly higher for the model-free systems compared to the model-based systems. When doing a Mann Whitney U test on the collision rate between model-free systems and model-based systems for the medium test environment, we get a p-value less than 0.001, indicating that the difference is statistically significant.

In the hard test environment, the difference in mean collision rate between the model-free systems and the model-based systems is smaller, but the model-free systems still reached a lower collision rate at 77.6% versus 88% for the model-based systems. The model-free systems were able to drive a longer distance, with a mean distance of 77.8 meters versus 51.5 meters for the model-based systems. Mean time spent in the environment before a collision is also significantly longer for the model-free systems, with a mean time of 16.2 seconds, while the model-based systems got a mean time of 10.7 seconds, see table 7.5. When doing a Mann Whitney U test on the difference in collision rate between model-free systems and model-based systems for the hard test environment, we get a p-value less than 0.001. To understand why the model-based system wasn't able to learn a good collision avoidance policy, we need to look at the components of the model in the model-based system, and how they perform.

5.4 VAE and MD-RNN

To understand why the model-based system didn't learn a good collision avoidance policy, we need to look at the components of the model-based system. The problem with the system might lie in the VAE, the MD-RNN or in the RL algorithm, and a deeper dive into these components is necessary to understand why the model-based system didn't work. In the model-based RL system, the VAE encodes input images to a latent vector, and then the RL algorithm uses this encoded vector as a visual input. One way to measure the performance of the VAE can be done by looking at how good it is at reconstructing images from latent vectors, by comparing input images to reconstructed images. To compare these images we can use Mean Squared Error (MSE), which the VAE used as loss during training. We can also use a measurement called Structural Similarity Index (SSIM), which is a perceptual metric that quantifies the image quality degradation that is caused by data compression or other image processing, like passing an

image trough a VAE [55]. To measure the MSE and SSIM of reconstructed images, we give the VAE 100 random images from the Carla environment from a dataset it has not been trained on. Then we compare the input image to the reconstructed image using MSE and SSIM as a measurement. This test gives our VAE a mean MSE score of 0.011±0.008 over 100 random images. With MSE a value close to 0 is better. The VAE got a mean SSIM score of 0.877±0.056, where a value closer to 1 is better. We can compare these scores to other papers. In Lu et al. [56] several variations of autoencoders were trained on the CelebA-64 dataset. This dataset is quite different from ours but can give some indication of how our VAE performs in relation to other papers. Table 5.1 shows a comparison between our VAE and the autoencdoers used in Lu et al. Here we can see that our VAE got a better SSIM score than the comparable autoencoders in Lu et al. Our MSE score is however significantly worse than the vanilla autoencoders, but better than the VAEs in Lu et al.

Models	MSE	SSIM
Our VAE	0.011	0.877
Vanilla Autoencoder, MSE loss (Lu. et al)	0.0042	0.800
Vanilla Autoencoder, SSIM loss (Lu. et al)	0.0057	0.86
VAE, MSE loss (Lu. et al)	0.0235	0.53
VAE, SSIM loss (Lu. et al)	0.0244	0.60

Table 5.1: Comparison showing MSE and SSIM score for our VAE and autoencoders in Lu et al.

Since the quantitative analysis above doesn't reveal which details the VAE is able to capture, looking at a few examples of reconstructed images may help give us an impression of which details are captured. See figure 5.8. Here we can see that the resulting reconstruction is fairly blurry, but it also seems to be able to capture the necessary details for collision avoidance. Where there are cars, there are blurry colored blobs and the road is still somewhat distinct from surrounding buildings.

Figure 5.8: This figure shows some examples of images from the Carla environment that has been encoded and reconstructed by the VAE, with belonging MSE and SSIM scores for each pair.

Since the model-based RL algorithm uses latent vectors and not reconstructed images, its also interesting to take a look at how these vectors are structured. One way to evaluate the structure of the latent vectors, is to see if we can intentionally manipulate latent vectors, and make the decoder reconstruct cars and other objects that aren't present in the original image. Figure 5.9 shows an example of how the VAE can generate new data. The original image shows a 128x128 image of a car on a road, which has been encoded to a latent vector then decoded, using the trained VAE. Because of how VAEs encode images, its possible to manipulate the encoded vector to generate new data. By subtracting values at index 61 in the latent vector, a new yellow car passing by on the left side of the road is generated. This may indicate that index 61 is responsible for representing cars on the left. Figure 5.10 shows a similar example. In the original reconstructed image (no vector manipulation) we can barely see a car in the distance. By subtracting values at index 0 in the latent vector, we can make the car appear closer. This may indicate that index 0 is responsible for distance from cars in the middle of the road. Based on these experiments, it appears that the VAE has some sort of structure that the RL algorithm should be able to take advantage of. For example, given a latent vector with a negative value at index 0, it should indicate that the agent is close to a car in front of it, and should either brake or turn to avoid a collision.

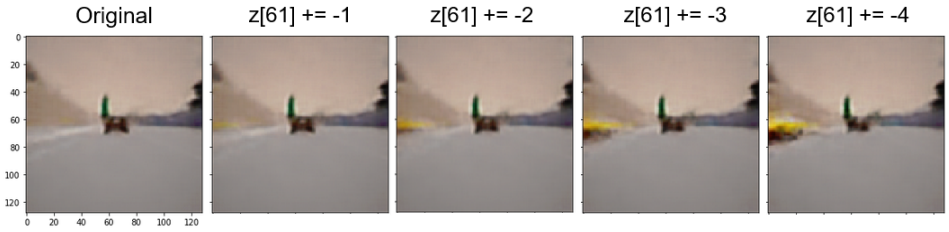

Figure 5.9: Figure showing latent vector manipulation for generating new data. Here a latent vector has been manually manipulated to generate a car on the left, which originally wasn't present in the original input image.

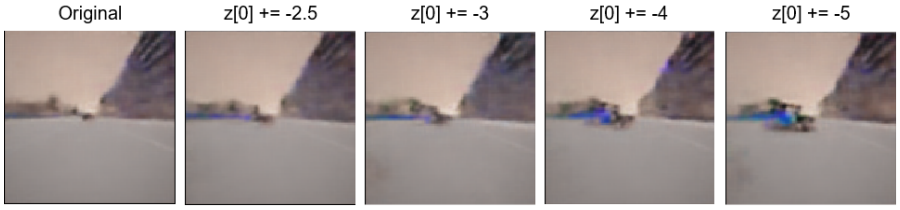

Figure 5.10: Figure showing latent vector manipulation for generating new data. Here a latent vector has been manually manipulated to make a car in the distance appear closer.

To quantitatively evaluate the predictive capabilities of the MD-RNN, we can use SSIM to compare predicted latent vectors that have been reconstructed to images by the decoder, with images of what actually happened (ground truth) at that time step. Then we can compare these scores to similar predictive networks. In Luc et al. [57] predictive networks was developed and tested on the Cityscapes dataset. The Cityscapes dataset is a dataset containing sequences of traffic in a city, making this paper a good candidate for comparison, as their data is fairly similar to the data our VAE and MD-RNN has been trained on. Our SSIM scores are based on the mean SSIM score for 100 random sequences, comparing the ground-truth frame in that sequence to the predicted frame from the MD-RNN. Table 5.2 shows a comparison of SSIM scores for our VAE+MD-RNN and the predictive networks in Luc et al. In this test our VAE+MD-RNN got a higher SSIM score than the predictive networks in Luc et al. The citiscapes dataset uses real world images, which is significantly more complex compared to images from the Carla simulator. This may have a noteable impact on the results.

Model	14 frames SSIM	20 frames SSIM
Our VAE+MD-RNN	0.85	0.83
X2X, AR(Luc et al.)	0.76	0.61
X2X, batch(Luc et al.)	0.76	0.65
XS2XS, batch(Luc et al.)	0.76	0.64

Table 5.2: SSIM comparison between our VAE+MD-RNN and the predictive networks used in Luc et al.

Figure 5.11 shows some examples of predictions 10 frames forward, which is the equivalent of a 1 second prediction. We can see that the predictions are fairly blurry, but does show some ability to predict what happens. In episode A we can see that the MD-RNN was able to predict that the gray car in the input frame would be closer 10 frames into the future, but it has been reconstructed as a blue car. In episode B it seems that it was able to correctly predict that the red car would be a little bit further away 10 frames into the future. In episode D it has predicted that the black car has moved forward, and that a new red car is starting to appear, which isn't present in the ground truth image.

Figure 5.12 shows some examples of predictions 20 frames forward, which is the equivalent of a 2 second prediction. In episode A the input frame shows a blue car appearing to the left, and the MD-RNN was able to correctly predict that the car moved further down the road. In episode B a barely visible red car appears in the distance. The MD-RNN is able to correctly predict a red object to be closer 20 frames into the future. In episode C it seems that the MD-RNN falsely predicted that the red car would be further away, while the ground truth shows that it should be closer. In episode D the VAE seems to have reconstructed the yellow car as a red car, while the flower pot has been changed to a blurry blue car. The cars are little bit further away, but not as much as in the ground truth, which could potentially lead to a collision. In episode E we can see a red car in the distance, which correctly appears closer in the prediction.

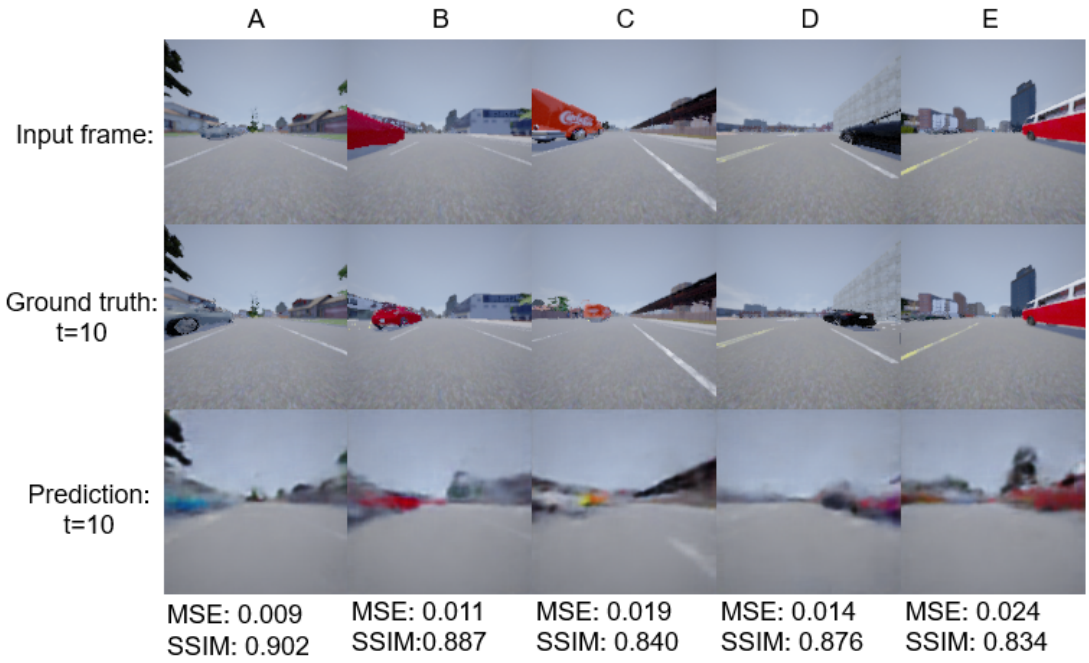

Figure 5.11: This figure shows some example predictions from the MD-RNN, where it has been given an input frame from a driving episode in the Carla environment and related actions taken during that episode. The 128x128 input frame is encoded by the VAE to a latent vector and given to the MD-RNN, then the resulting prediction from the MD-RNN is decoded by the VAE into a 128x128 image. These examples show a prediction 10 frames forward. Since the MD-RNN was trained on 10 fps, this represents 1 second. Ground truth shows what actually happened 10 frames forward in this episode. MSE and SSIM represents the difference between Ground truth and Prediction.

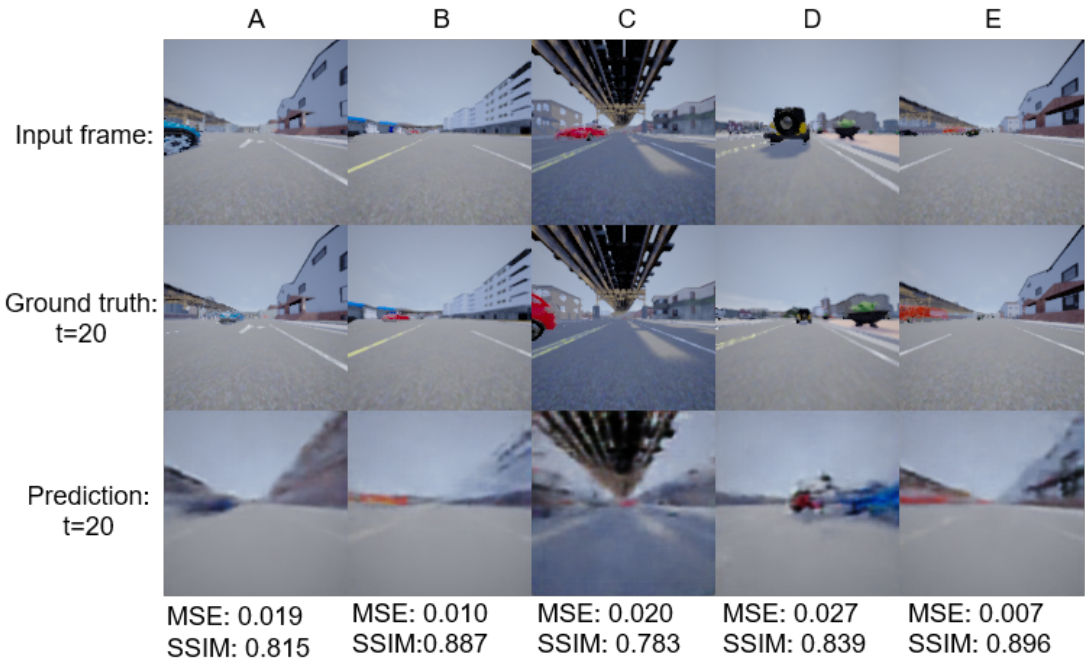

Figure 5.12: This figure shows some example predictions from the MD-RNN. This examples shows a prediction 20 frames forward, representing 2 seconds into the future. Ground truth shows what actually happened 20 frames forward in this episode. MSE and SSIM represents the difference between Ground truth and Prediction.

These examples, as well as the comparison between similar predictive networks, see figure 5.2, show that the MD-RNN is capable of making mostly reasonable predictions, although some details are lost and sometimes the predictions are faulty. In some examples, cars that should be predicted to be further away are predicted to be closer, which can quickly lead to a collision. We can also see that completely false predictions get a higher SSIM score than the predictive networks we have compared our MD-RNN with. Perhaps these faulty predictions makes it difficult for the DDQN to interpret its input vector, as the data it is given doesn't always correspond with what is actually happening in the environment. Although these faulty predictions should be remedied by the vision component (VAE), which provides encoded vectors of real time images. When looking at the architecture of the VAE, it should be noted that the convolutional network in the VAE is very similar to the convolutional network used in the model-free system. See figure 5.4. The main difference is that conv-layers of the encoder outputs a vector of means (μ) and a vector of standard deviations (σ). These two vectors are then randomly sampled to obtain the encoded vector. The Encoder is also trained differently, using a pre-made dataset and trained in conjunction with the decoder. Meanwhile in the model-free system, the output of the conv-layers is simply flattened, and its trained online. Overall they should have the same power for visual recognition.

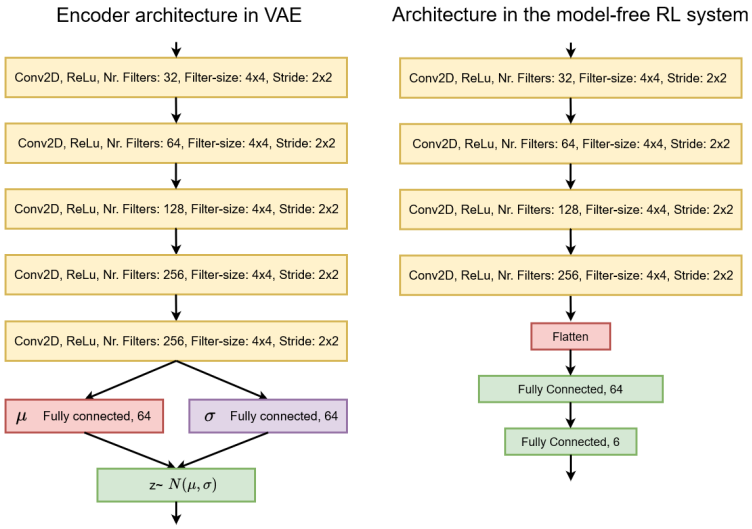

Figure 5.13: Comparison of architecture in encoder and architecture in the model-free RL system.

It seems like the Carla environment may have been too complex to be represented by the VAE+MD-RNN that was proposed by the World Models paper[8]. They saw groundbreaking success in simple game environments, examples of which can be seen in figure 3.3, but does not seem to work as well in the Carla environment. We have seen that the VAE and MD-RNN performs as good or better than comparable papers when comparing SSIM and MSE, see tables 5.1 and 5.2. However upon further inspection, faulty predictions by the MD-RNN is also present, as seen in figure 5.12. Here we can see that even faulty predictions are able to score fairly high when using SSIM as a measurement. One paper that saw success with model-based reinforcement learning in the Carla environment is Porav et al, 2019 [8]. A description of their paper can be found in section 3.3. Their architecture is similar to ours, which is based on the World Models architecture[40], where they use an Autoencoder and an RNN to encode images to the latent space and make latent predictions. With our architecture, we concatenate latent vectors that have been encoded by the VAE together with the hidden states of the RNN. While in Porav et al, they encode input images to a latent vector, and then use an RNN to make a predicted latent vector one timestep forward, and concatenate these two vectors and use that as input to the RL-algorithm. However their collision avoidance scenarios was much more controlled, focusing on specific, orchestrated scenarios, and they used segmented images as camera input. By using segmented images as input, the complexity of the environment is significantly reduced. This could have been a good option to mitigate our issues, and with the recent performance of real-time semantic segmentation approaches [58][59], we can assume that such a system would be readily available as an input processor.

5.5 Testing a variation of the model-based algorithm

The model-based RL algorithm uses a VAE (see 2.4.1) to encode images into latent vectors, and a MD-RNN (see 2.5.2) to predict future latent vectors. The latent vector from the VAE is concatenated with the hidden states of the MD-RNN, and given to the RL algorithm as input. These components can easily be used to predict future images while an agent car is driving. By using the decoder in the VAE to decode a predicted latent vector from the MD-RNN, we can get images several time steps into the future. By using these predicted future images as input to the model-free RL algorithm, instead of real time images, the model-free RL algorithm may potentially learn a better collision avoidance policy by taking advantage of the predictive capabilities of the MD-RNN. This technique gives us an intermediate method between the model-free and the model-based RL algorithms, where we get the predictive capabilities of the model-based RL algorithm, and the visual and control capabilities of the model-free RL algorithm. The model-free systems were able to learn a fairly good collision avoidance policy using RGB images, while it seems that the model-based systems weren't able to interpret or learn anything from the latent space representations of the environment. We already know that the DDQN is able to learn a collision avoidance policy using RGB images, so by using the decoder in the VAE to bring the predicted latent space representations back to the RGB image domain, it might tell us whether the problem lies in the environment being represented in the latent space, or if the problem was that the VAE and MD-RNN wasn't able to create correct predictions in the first place. See figure 5.14 for a flowchart of this system. This system is trained using the same parameters and training scheme as in the model-free system, see section 4.4 and table 4.2 for hyperparameters used during training. This method will be tested using the same testing scheme as in section 5.3 Testing the Collision Avoidance Policy. See results below.

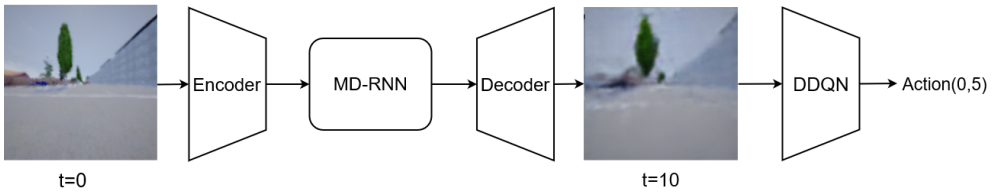

Figure 5.14: Architecture of Predicted-Model-Free RL algorithm. Image at t=0 is input. Image at t=10 is the predicted output which is used as input to the DDQN. MD-RNN is trained on image sequences with 10 FPS, so predicting 10 frames represent 1 second into the future.

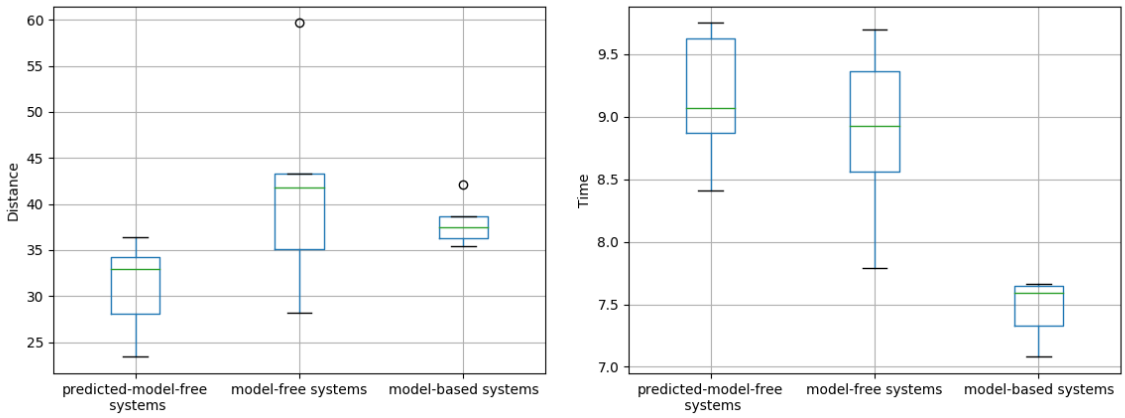

Figure 5.15: Box-plot comparing median distance and time for all systems, tested for 100 episodes in the easy environment. Higher values is better.

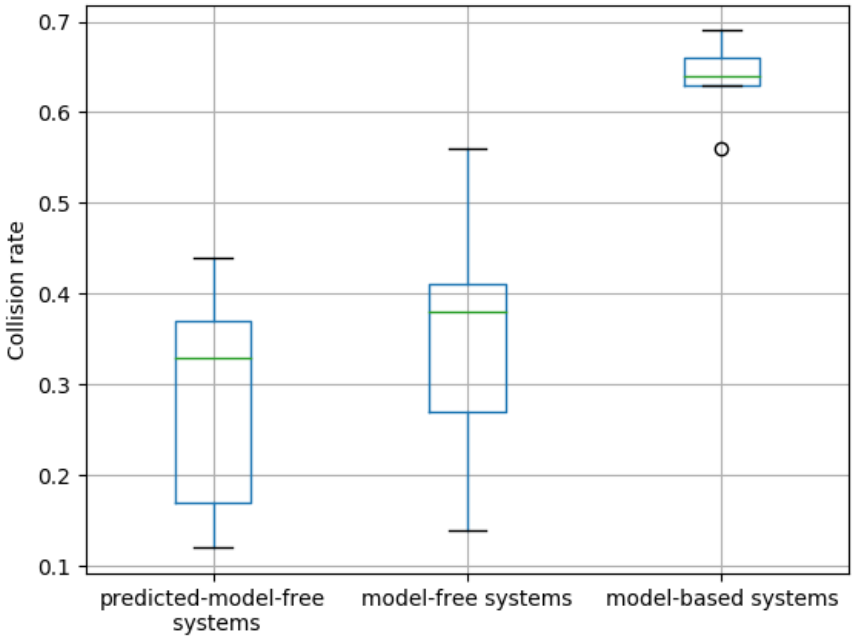

Figure 5.16: Box plot comparing median collision rate for all systems, tested for 100 episodes in the easy environment. Lower values is better.

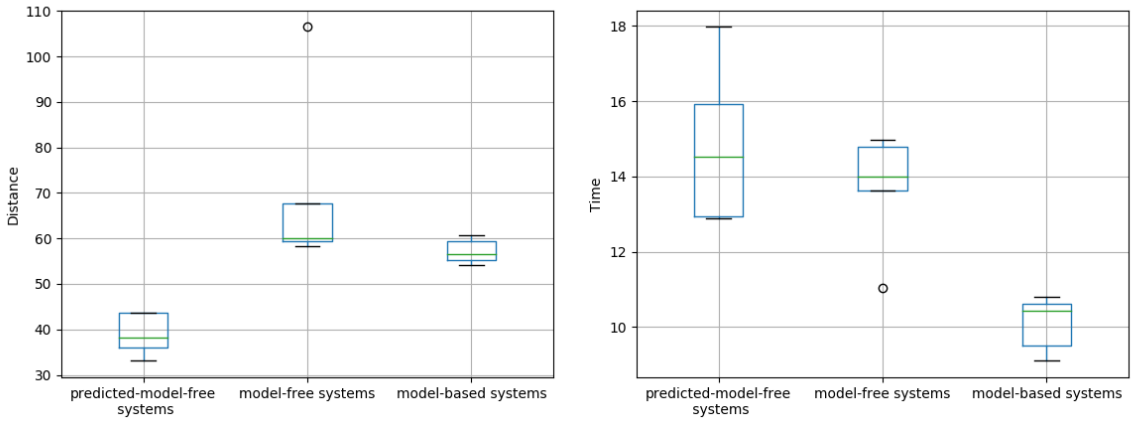

Figure 5.17: Box plot comparing median distance and time for all systems, tested for 100 episodes in the medium environment.

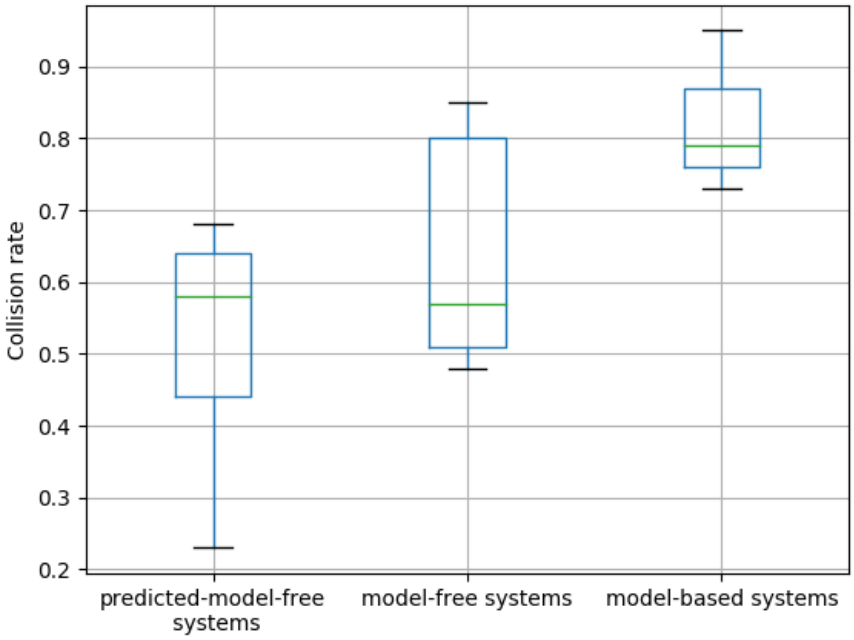

Figure 5.18: Box plot comparing median collision rate for all systems, tested for 100 episodes in the medium environment.

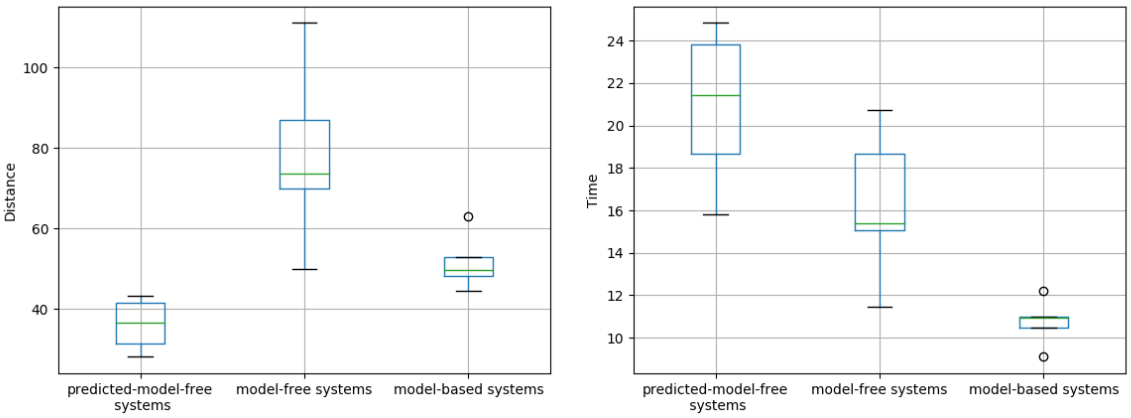

Figure 5.19: Box plot comparing median distance and time for all systems, tested for 100 episodes in the hard environment.

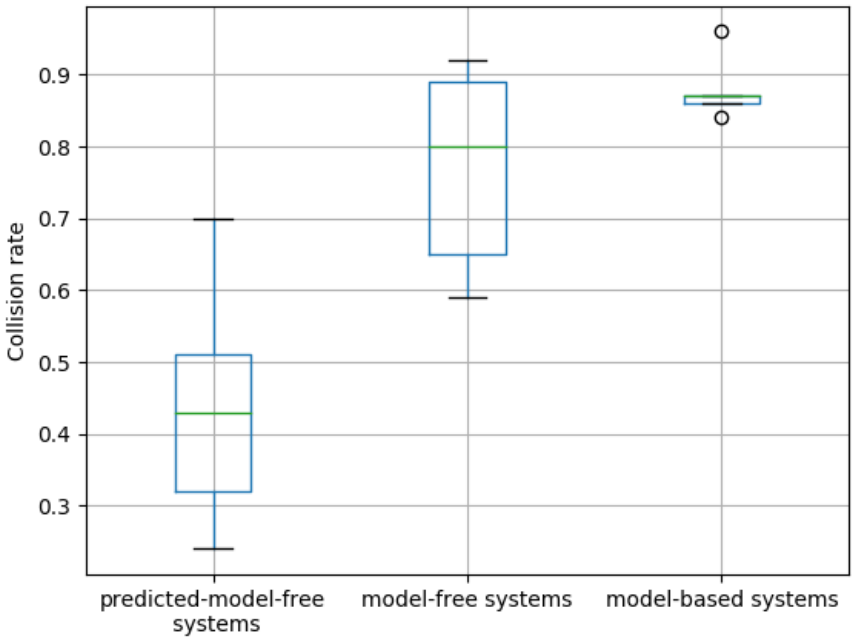

Figure 5.20: Box plot comparing median collision rate for all systems, tested for 100 episodes in the hard environment.

Based on these tests, the predicted-model-free systems regularly has a much lower collision rate than the model-based systems, and a slightly lower collision rate than the model-free systems. However it also drives shorter distances than the other models. Tables showing results for each individual trained model can be found in Appendix, section 7.2. In the first test using the easy environment setting, the predicted-model-free systems

got a mean collision rate of 28.6%. In the same test the model-free systems got a mean collision rate of 35.2% and the model-based systems got a mean collision rate of 63.6%. In this test the predicted-model-free systems got a slightly smaller mean distance than the model-free and model-based systems. To check that the difference in collision rate is statistically significant, we can do a Mann Whitney U test, comparing the collisions of the predicted-model-free systems to the model-free systems. This gives a p-value of 0.013, indicating that the lower collision rate for the predicted-model-free systems is statistically significant.

In the second test with the medium environment setting, the predicted-model-free systems got a mean collision rate of 51.4%, which is lower than the model-free and model-based systems, which got a mean collision rate of 64.2% and 82.0% respectively. However when looking at the median collision rate, which can be seen in box-plot 5.18, the predicted-model-free systems got a fairly similar median collision rate to the model-free systems. The predicted-model-free systems also got a significantly lower mean distance, at 38.9 m, compared to the model-free systems and model-based systems, which got a mean distance of 70.3 m and 57.1 m respectively. Checking for statistical significance we get a p-value less than 0.001, indicating that the predicted-model-free has a statistically lower collision rate than the model-free systems.

In the third test, with the hard environment setting, the predicted-model-free systems got a significantly lower collision rate than the other systems, with a mean collision rate of 44%. In this test the model-free systems got a mean collision rate of 77%, and the model-based systems got a mean collision rate of 88%. Checking for statistical significance between the collision rate for the predicted-model-free systems and the model-free systems we get a p-value less than 0.001. However the predicted-model-free systems also drives significantly lower distances, which we can see in boxplot 5.19, which naturally leads to a lower collision rate. Overall the predicted-model-free system drove much shorter distances than the other systems, which is possibly a result of the MD-RNN sometimes generating false predictions, making it difficult for the agent to learn. Considering its short distances, it did not seem to learn a better police for driving and avoiding collisions compared to the model-free RL system.

We know that the DDQN is able to learn a fairly good collision avoidance policy that is able to drive longer distances than other systems without crashing, which we saw with the model-free system. When adding the MD-RNNs predictions to this model-free system, performance seems to go down. This indicates that the problem lies with the predictive abilities of the MD-RNN.

5.6 Transferability

In this experiment we will test the model-free and model-based RL algorithms ability to reuse past knowledge and learn a new task. When it comes to the model-based RL system, once a model of the environment

have been developed, it should be possible to use this model for any task that is relevant to that environment. And since the entire system is split up into components, its easy to simply re-train the control component separately, making transfer learning and learning new tasks easy. In the model-free system, the convolutional layers have been trained together with the decision layers, forming an internal visual model of the world. These layers can be frozen and reused when learning a new task, training only the last fully connected decision layers.

In this experiment, the new task will be to follow a car in front of it. The car it is following is controlled by Carlas autopilot, and follows the speed limits and rules of the city we are training in. This task is fairly different from the collision avoidance task, where the agent wants to stay away from cars. In this task, it has to stay close to the car, following its path, while also not crashing in the car. Although there are some similarities with the collision avoidance task, as it has to identify cars or a car, and it still has to make sure it doesn't crash into the car or surrounding objects in the environment, making this task a good fit for a transfer learning experiment.

5.6.1 Training

Because training many models is very time consuming and resource demanding, we will not train 5 models for each system in this experiment, as done previously. Instead we will choose the best performing trained model from each system, and use those in our transfer-learning experiment. So these results may not be conclusive, but may give us an indication of each systems ability to learn a new task. We will also train a model-based system and model-free system from scratch with the same training parameters, so that we can compare with and without transfer learning, to see if transfer learning actually gave a benefit, and which system benefited the most from transfer learning.

During training, a car controlled by Carlas autopilot will be spawned 20 meters in front of the agent car, and then it has to follow that car within a given range. Each system is trained for 1000 episodes, with a 15 second episode length. The reward function is changed to reflect its new task of following a car. See reward function below. In this reward function, the car gets a punishment of -1 and the episode ends if the agent car crashes with the pilot car or any other object. When the agent car is in range of the pilot car, it gets a reward of 1 divided by the distance from the pilot car. This gives a higher reward when the car is close to the pilot car, but there are still small rewards to be gained at longer distances as well. Being within range of the pilot car is defined as having a distance from the pilot car between 2 and 30 meters. If the agent car falls out of range from the pilot car, the episode ends. The two systems were trained with the same parameters used previously, as defined in the Method chapter. See table 4.2 for parameters used during training of the model-free system in this experiment, and table 4.5 for parameters used during training of the model-based system. For the systems that are trained from scratch, epsilon is set to 1 and the decay parameter is set to 0.99 for exploration. Since the model-

based system wasn't able to learn using its model of the environment, it is unexpected that it will learn much in this experiment. However transfer learning may still show some benefits.

```
if collision == True:
    reward = -1
    end_episode = True
if distance_from_pilot > 2 and distance_from_pilot <= 30:
    reward = (1 / distance_from_pilot)
    end_episode = False
if self.pilot_distance > 30:
    end_episode = True
```

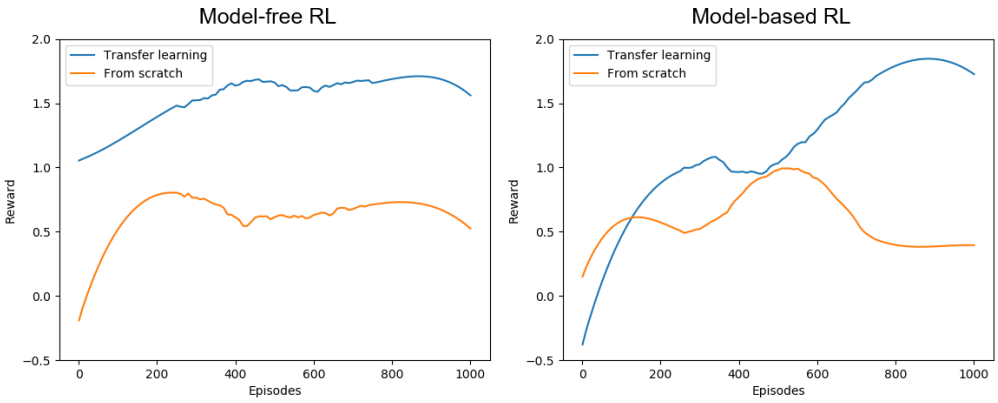

Figure 5.21: Reward during training, smoothed with Savitsky Golay filter [47]

Figure 5.21 shows the resulting rewards during training, comparing transfer learning and training from scratch. As we can see, using transfer learning resulted in a significantly higher reward than training from scratch, both for the model-free RL system and the model-based RL system. The model-free RL system with transfer learning started out with a higher reward early on, which is probably a result of its good collision avoidance policy making sure it doesn't crash in the pilot vehicle. So here it seems like it was able to transfer some of its past knowledge to a new task. While the model-based RL system with transfer learning started out with lower reward than the system trained from scratch. This is a result of its learned driving policy from the collision avoidance training, were it learned to mostly drive straight. This results in a collision with the pilot vehicle, resulting in a punishment of -1. However it seems that it quickly adapted to the new reward function and did change its behavior. None of the systems learned an exceptionally good policy for following the pilot vehicle, however the RL systems did benefit from transfer learning, and it does seem like both gained a better starting point for learning, especially the model-free RL system, which already started with a much higher reward than the system trained from scratch.

5.6.2 Testing

A test was done to see which system ended up with a better policy for following a vehicle after training. During this test, the pilot car was spawned 15 meters in front of the agent car, and the agent was given 20 seconds to follow the agent car per episode. Each system was tested for 100 episodes. For each episode, mean distance from pilot vehicle over an episode and time spent within range of the pilot was recorded. Figure 5.22 shows a boxplot of the resulting mean distance for each system. Figure 5.23 shows a boxplot of the resulting time spent within range for each system.

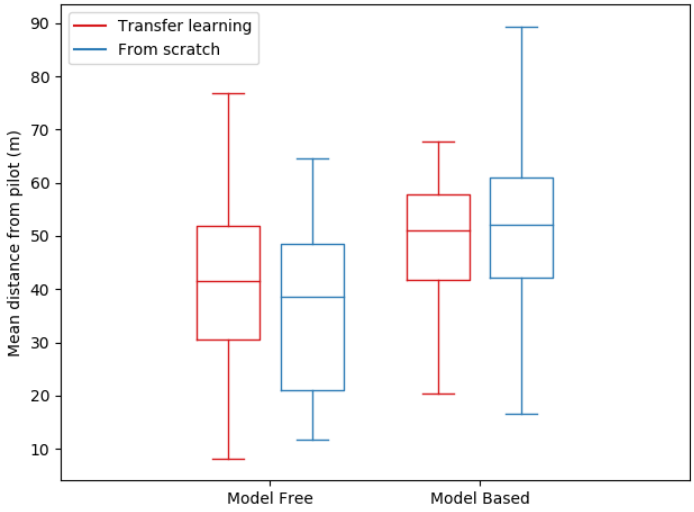

Figure 5.22: Boxplot showing mean distance from pilot vehicle over an episode, tested for 100 episode for each system. Lower values are better.

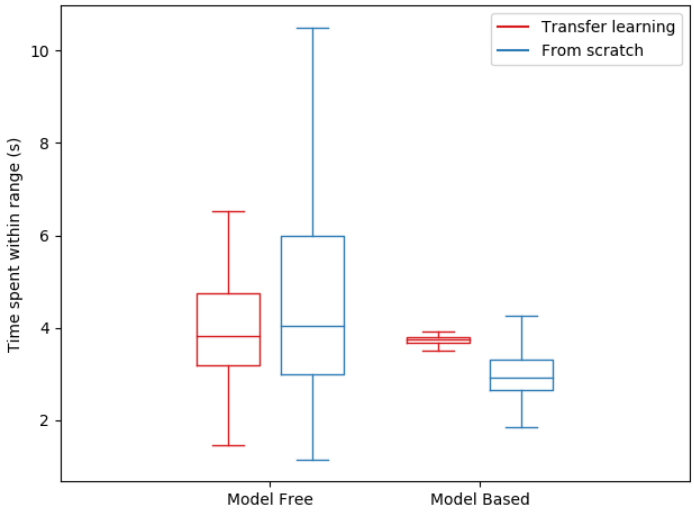

Figure 5.23: Boxplot showing mean time spent within range of pilot vehicle, tested for 100 episodes for each system. Higher values are better.

From the boxplots we can see that both the systems trained with transfer learning and the systems trained from scratch ended up with a fairly equal performance based on our test. Although rewards for the systems trained with transfer learning was significantly higher than the systems trained from scratch, this did not translate to a notably better performance during testing. This is probably because rewards are already so low, for both the systems trained with transfer learning and from scratch, that the higher rewards of the transfer learned systems didn't have that big of an impact on performance. We can see that the model-based system spends somewhat more time within range of the pilot vehicle with a median time of about 4 seconds spent within range. While the model-based system trained from scratch has a median of about 3 seconds. A Mann Whitney U test on these distributions gave a p-value less than 0.05. The model-based system trained with transfer learning also has a slightly lower mean distance from the pilot vehicle ($p = 0.04$). This indicates that the model-based system trained with transfer learning is slightly better. For the model-free systems, median time spent within range is slightly higher for the system trained from scratch than the system trained with transfer learning, however a Mann Whitney U test gives a p-value of 0.48, higher than 0.05, so this difference isn't statistically significant. The model-free system trained from scratch also has a lower mean distance from pilot vehicle than the system trained with transfer learning ($p = 0.02$). So performance wise the model-free system did not benefit much from transfer learning, while the model-based system saw a slight performance wise benefit. While this test does indicate some benefits with transfer learning for the model-based system, the sample size is too small for it to be conclusive results, since only one trained model of each system was used. There was also some issues with the experiment, since the gain in rewards from transfer learning didn't translate to a significantly better performance.

5.7 Training and Data Efficiency

A problem with reinforcement learning is that it requires a lot of samples in the environment. This makes it difficult to use reinforcement learning in the real world, and limits it to simulation environments. Model-based reinforcement learning attempts to mitigate this [60]. Model-based RL seeks to make training more efficient by reusing a learned internal model for several different tasks [12], thus requiring more data when this internal model is first trained. In this section we will discuss the sample efficiency and complexity of the model-free and model-based reinforcement learning algorithms used in this project.

The training of the model-based system was done in three phases. First the VAE was trained, which required a dataset of the Carla environment. To create this dataset a car was set to drive in the Carla environment, collecting 400 000 RGB images and 400 000 equivalent segmented images which was used as a weight mask in the loss function. The VAE was trained on these images for 50 epochs with a batch size of 100. When training the MD-

RNN, an additional dataset was made, which consisted of image sequences of driving that had been encoded to latent vectors by the VAE, and the associated actions taken during that sequence. However the dataset made for the VAE could be reused in this case. The MD-RNN was trained on this dataset for 200 epochs with a batch size of 200. After training the VAE and MD-RNN, the RL algorithm was trained for 4000 episodes with an episode length of 10 seconds.

In comparison, the model-free system was trained for only 4000 episodes, and achieved a better collision avoidance policy than the model-based system. No dataset was needed, as the RL algorithm learned online, directly from the environment. This makes the model-free RL system the most data efficient system, compared to the model-based system. This shows that developing and training a model of the environment for model-based RL can be a more complex and time consuming process compared to model-free RL, especially in complex environments, and sometimes the model doesn't work. When the model doesn't work, which happened in our case, it can hinder learning or cause the wrong policy to be learned. However when the model does work, one major benefit that other papers have found is that they can quickly learn new tasks [12], allowing an already trained model to be reused. In the context of autonomous driving, this can be quite useful, as there are many sub-tasks, for example collision avoidance, following other cars, staying in lanes or overtaking a car

Chapter 6

Conclusion

This project aimed at comparing model-free reinforcement learning with model-based reinforcement learning for collision avoidance in a complex car simulator. A model-free RL algorithm was implemented using a Double Deep Q Network, while a model-based RL algorithm was implemented by combining the World Models architecture with a Double Deep Q Network. The model-free RL algorithm learns directly from RGB images, while the model-based RL algorithms learn from a latent space representation of the environment. These systems were trained in the Carla simulation environment, where the goal was to learn to avoid collisions in traffic situations. The two systems were tested and compared. From these tests, it was found that the model-based RL algorithm was not able to learn a good collision avoidance policy using the World Models architecture, while the model-free RL algorithm achieved a significantly better collision avoidance policy.

Further experiments were done to gain an understanding of where the issues with the model-based RL algorithm lie. Training an internal model is difficult, especially for complex environments. The World Models architecture saw state of the art results in two simple game environments [40], but based on our findings it is not powerful enough for the complex 3D car simulator environment used in this project. The components of the model-based RL algorithm were compared to similar research and their results. The visual component, consisting of a Variational Autoencoder, was found to perform better than similar networks, using SSIM and MSE as a measurement for performance. The memory component, consisting of a Mixture-Density Recurrent Neural Network (MD-RNN) was also found to perform better than similar predictive networks when using SSIM as a measurement. However, the compared networks were trained on the Citiscapes dataset, while ours is trained on the Carla simulation environment. This difference may have a notable impact on the results, as the Carla simulation environment is less detailed and simpler than the real world images in the Citiscapes dataset. Upon further inspection of the MD-RNN, some examples of faulty predictions were found. Since the model-free RL algorithm was able to learn a fairly good collision avoidance policy using RGB images, an additional experiment was performed were

we decoded the predicted latent space representation used by the model-based RL algorithm back to the image domain, and tested it with the model-free RL algorithm. It was unable to learn a good collision avoidance policy using these predicted images, indicating that the MD-RNN is not able to make accurate predictions. This disconnect between real-world events and predicted events may have hindered learning.

The model-based and model-free RL algorithm's ability to reuse past knowledge for learning a new task was tested. Here we found that both the model-based and model-free RL algorithms saw a slight benefit from transferring past knowledge to a new task during training, giving an increase in rewards. However, this did not translate to a notably better performance during testing. In all likelihood, this is because rewards are generally so low for both the systems trained from scratch and those trained with transfer-learning, that the increase in rewards wasn't enough to meaningfully impact performance. The model-based system saw a slight benefit in performance on the new task, while the model-free system trained with transfer learning performed similarly to the system trained from scratch. This indicates that reusing past knowledge is something that model-free systems struggle with. Model-based RL facilitates transfer learning, because it has a separate learned model of the environment, while in model-free RL there is no clear separation between policy and its understanding of the environment. Further research on model-based reinforcement learning systems is necessary so that we can reap the benefits of transfer learning in RL.

Finally, training time and data efficiency between the two systems were compared. The model-based system took significantly more time to train, since components forming the internal model had to be trained, in addition to the RL algorithm. A dataset also had to be made to train these components. While in the model-free system only the RL algorithm had to be trained. Since the model-based system was unable to learn from its representation of the world, we did not see the benefits of model-based RL that other papers have found, like sample efficiency and easier transfer learning.

To summarize, this thesis demonstrates that implementing model-based RL in complex environments comes with several challenges. When the predictive model fails, the disconnect between real-world events and predicted events hinders learning. More research on the best way to develop and implement model-based RL for complex environments is necessary. It is hoped that the findings in this thesis shine a light on problems and pitfalls with model-based reinforcement learning in complex environments, and encourages further research on this topic so we can reap the benefits of model-based RL for all manner of different tasks, including collision avoidance.

6.1 Further work

Model-based RL is still not as well understood as model-free RL. More studies on the best way to develop models and representations of environments for model-based RL is needed. Looking at networks that exhibit dynamic temporal behavior, like recurrent neural networks, is a worthwhile endeavor in furthering the abilities of model-based RL. More research comparing model-based RL with well known model-free RL algorithms is also necessary so that we can better understand the costs and benefits of the two methods.

The World Models architecture [40] was in all likelihood not powerful enough to learn an accurate model of the complex 3D environment in Carla. Porav et al. [8] saw success with model-based RL in the Carla environment using a similar architecture to the World Models architecture. However, instead of using RGB images, they used segmented images. By using segmented images, the complexity of the environment is significantly reduced, bringing it to a similar complexity as the environments that the World Models papers saw success with. This could have been a solution to our problems, and with the recent performance of real-time semantic segmentation approaches [58][59], we can assume that such a system would be readily available as an input processor. The Carla simulator also has a built-in semantic segmentation, making it very easy to implement for this project.

Another aspect that should be looked at is longer training times for the reinforcement learning portion of the systems presented in this thesis. Both the model-free and model-based RL algorithms were only trained for 4000 episodes, which from a reinforcement learning perspective is a fairly short amount. Training reinforcement learning algorithms often require tens to hundreds of millions of samples [60]. Therefore looking at longer training times is a worthwhile experiment for this project. Initial tests with longer training time earlier in this project were made, however it is very time-consuming. 20000 episodes with a 10-second episode length were tested and took about 55 hours. These tests saw little success, but this was early in the project while hyperparameters and training were still being tweaked.

Lastly, the transfer-learning experiment should see further testing. The experiment should be set up in a similar way to the main experiment, where 5 models of each system are trained with transfer learning, and 5 models of each system are trained from scratch. Then they are all tested and compared, and in that way we would get more conclusive results regarding these architecture's ability to reuse knowledge.

Chapter 7

Appendix

7.1 Results: Testing the collision avoidance policy

Models	Collision %	Mean Dist	Max Dist	Mean Time
Model-Free 1	14%	35.1±19.2 m	63.6 m	9.7±1.0 s
Model-Free 2	27%	59.6±16.2 m	75.2 m	9.3±1.5 s
Model-Free 3	56%	28.2±17.0 m	57.6 m	7.7±2.6 s
Model-Free 4	38%	41.7±18.7 m	70.4 m	8.9±1.9 s
Model-Free 5	41%	43.2±18.6 m	70.2 m	8.5±2.3 s
Total Mean	35.2%	41.5 m	67.4 m	8.8 s
Model-Based 1	69%	35.4±22.3 m	98.4 m	7.3±2.3 s
Model-Based 2	64%	38.6±24.3 m	92.1 m	7.6±2.4 s
Model-Based 3	66%	37.4±22.6 m	75.4 m	7.5±2.3 s
Model-Based 4	63%	36.3±22 m	75.9 m	7.0±2 s
Model-Based 5	56%	42.1±22.9 m	75.3 m	7.6±2.3 s
Total Mean	63.6%	37.9 m	83.4 m	7.4 s

Table 7.1: Models tested for 100 episodes in the easy test environment, where episode length is set at 10 seconds and 50 cars are spawned in environment. Collision percentage measures how many of the 100 episodes resulted in a collision.

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	86	4	6	3	1
Model-Free 2	73	7	12	6	2
Model-Free 3	44	12	26	10	8
Model-Free 4	62	9	22	5	2
Model-Free 5	59	8	15	11	7
Total Mean	64.8	8	16.2	7	4
Model-Based 1	31	58	7	3	1
Model-Based 2	36	50	7	6	1
Model-Based 3	34	60	3	2	1
Model-Based 4	37	57	2	1	3
Model-Based 5	44	43	8	4	1
Total Mean	36.4	53.6	5.4	3.2	1.4

Table 7.2: Models tested with for 100 episodes in the easy test environment. Table showing distribution of objects that the models experienced a collision with. Vehicles represent cars, motorcycles and bikes controlled by the simulator. Walls represent buildings and walls in the environment. Poles represent trees, traffic lights and traffic signs. Misc represent various objects in the environment like large flowerpots, trashcans and similar objects.

Models	Collision %	Mean Dist	Max Dist	Mean Time
Model-Free 1	48%	59.3±42.1 m	133.7 m	14.9±6.1 s
Model-Free 2	80%	106.4±57.0 m	234.0 m	13.6±5.1 s
Model-Free 3	51%	58.3±37.2 m	123.6 m	14.8±6.1 s
Model-Free 4	57%	67.6±46.7 m	151.8 m	13.9±6.1 s
Model-Free 5	85%	60.1±44.1 m	186.8 m	11.0±5.6 s
Total Mean	64.2%	70.34 m	165.9 m	13.6 s
Model-Based 1	87%	56.5±59.1 m	231.2 m	9.5±5.9 s
Model-Based 2	73%	55.1±55.4 m	235.5 m	10.7±6.8 s
Model-Based 3	79%	59.3±58.9 m	212.9 m	10.6±6.6 s
Model-Based 4	95%	54.0±54.3 m	253.3 m	9.1±5.5 s
Model-Based 5	76%	60.6±58.3 m	234 m	10.4±6.5 s
Total Mean	82%	57.1 m	233.3 m	10.0 s

Table 7.3: Models tested for 100 episodes in the medium test environment, where episode length is set at 20 seconds and 75 cars are spawned in environment. Here the standard deviation is very high, and sometimes higher than the mean. This is because the agent car is spawned at random locations in the environment, and the distance it is spawned from obstacles and vehicles varies greatly as a result, which causes the time to collision and the distance it is able to drive to vary.

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	52	17	17	12	2
Model-Free 2	20	31	24	16	9
Model-Free 3	49	13	13	12	7
Model-Free 4	43	17	24	11	5
Model-Free 5	15	26	29	18	12
Total Mean	35.8	20.8	21.4	13.8	7.0
Model-Based 1	13	69	12	4	2
Model-Based 2	27	63	4	1	5
Model-Based 3	21	62	10	5	2
Model-Based 4	5	65	20	8	2
Model-Based	24	59	9	3	5
Total Mean	18	63.6	5.4	11	3.2

Table 7.4: Models tested with 100 episodes in the medium test environment. Table showing distribution of objects that the models experienced a collision with. The model-free systems seem to maintain a fairly even distribution of collision objects, even when there are more cars spawned in the environment, showing that they are better at avoiding vehicles compared to the model-based systems, which mainly crashes with vehicles.

Models	Collision %	Mean Dist	Max Dist	Mean Time
Model-Free 1	59%	86.9±63.2 m	208.8 m	20.7±9.8 s
Model-Free 2	89%	111.0±77.8 m	303.5 m	15.0±8.0 s
Model-Free 3	65%	73.6±55.2 m	183.3 m	18.6±10.0 s
Model-Free 4	80%	69.9±61.7 m	245.8 m	15.4±9.6 s
Model-Free 5	95%	49.8±45.3 m	193.8 m	11.4±7.7 s
Total Mean	77.6%	77.8 m	227.0 m	16.2 s
Model-Based 1	87%	49.6±48.3 m	220.4 m	11.0±8.3 s
Model-Based 2	86%	63.0±67.0 m	314.6 m	12.2±9.1 s
Model-Based 3	84%	44.5±50.3 m	222.3 m	10.9±9.3 s
Model-Based 4	96%	52.8±52.1 m	218.6 m	9.1±6.3 s
Model-Based 5	87%	48.0±56.0 m	254.8 m	10.4±8.9 s
Total Mean	88%	51.5 m	246.1 m	10.7 s

Table 7.5: Models tested for 100 episodes in the hard test environment, where episode length is set to 30 seconds and 100 cars are spawned in environment.

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	41	22	17	16	4
Model-Free 2	11	34	24	18	13
Model-Free 3	35	28	21	11	5
Model-Free 4	20	23	37	9	11
Model-Free 5	8	26	29	22	15
Total Mean	23.0	26.6	25.6	15.2	9.6
Model-Based 1	13	58	19	8	2
Model-Based 2	14	64	13	5	4
Model-Based 3	16	68	10	6	0
Model-Based 4	4	75	12	4	5
Model-Based 5	13	68	13	5	1
Total Mean	12.0	66.6	13.4	5.6	2.4

Table 7.6: Models tested for 100 episodes in the hard test environment. Table showing distribution of objects that the models experienced a collision with. Vehicles represent cars, motorcycles and bikes controlled by the simulator. Walls represent buildings and walls in the environment. Poles represent trees, traffic lights and traffic signs. Misc represent various objects in the environment like large flowerpots, trashcans and similar objects

7.2 Results: Testing a variation of the model-based algorithm

Models	Collision %	Avg Dist	Max Dist	Avg Time
Predicted-Model-Free 1	17%	28.0±21.6 m	73.2 m	9.6±1.3 s
Predicted-Model-Free 2	44%	23.4±19.1 m	71.1 m	8.4±2.3 s
Predicted-Model-Free 3	33%	34.2±22.5 m	71.6 m	8.8±1.9 s
Predicted-Model-Free 4	12%	36.3±27.2 m	80.3 m	9.7±1.4 s
Predicted-Model-Free 5	37%	32.9±23.2 m	94.7 m	9.0±1.8 s
Total Mean	28.6%	30.9 m	78.1 m	9.1 s

Table 7.7: Models tested for 100 episodes in the easy test environment

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	83	5	9	1	2
Model-Free 2	56	9	27	3	5
Model-Free 3	67	4	18	5	6
Model-Free 4	88	4	2	5	1
Model-Free 5	63	12	16	3	6
Total Mean	71.4	6.8	14.4	3.4	4

Table 7.8: Collision distribution for easy test environment

Models	Collision %	Avg Dist	Max Dist	Avg Time
Predicted-Model-Free 1	44%	36.0±40.4 m	205.3 m	15.9±6.2 s
Predicted-Model-Free 2	68%	33.0±32.5 m	185.0 m	12.9±6.4 s
Predicted-Model-Free 3	64%	38.2±36.2 m	162.3 m	12.8±6.6 s
Predicted-Model-Free 4	23%	43.7±44.6 m	223.0 m	17.9±5.1 s
Predicted-Model-Free 5	58%	43.6±34.2 m	153.3 m	14.5±6.2 s
Total Mean	51.4%	38.9 m	185.7 m	14.8 s

Table 7.9: Models tested for 100 episodes in the medium test environment.

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	56	14	18	8	4
Model-Free 2	32	11	32	13	12
Model-Free 3	36	20	29	6	9
Model-Free 4	77	11	7	3	2
Model-Free 5	42	14	31	8	5
Total Mean	48.6	14	23.4	7.6	6.4

Table 7.10: Collision distribution for medium test environment.

Models	Collision %	Avg Dist	Max Dist	Avg Time
Predicted-Model-Free 1	32%	36.5±40.7 m	219.9 m	23.8±9.6 s
Predicted-Model-Free 2	70%	28.0±28.6 m	173.5 m	15.8±10.7 s
Predicted-Model-Free 3	51%	31.3±30.9 m	196.8 m	18.6±11.6 s
Predicted-Model-Free 4	24%	43.0±42.6 m	195.2 m	24.8±9.6 s
Predicted-Model-Free 5	43%	41.3±40.4 m	222.6 m	21.4±10.5 s
Total Mean	44.0%	36.0 m	201.6 m	20.8 s

Table 7.11: Models tested for 100 episodes in the hard test environment

Models	No collision	Vehicles	Walls	Poles	Misc
Model-Free 1	68	15	8	2	7
Model-Free 2	30	17	39	4	10
Model-Free 3	49	20	16	9	6
Model-Free 4	76	18	3	3	0
Model-Free 5	57	12	21	6	4
Total Mean	56.0	16.4	17.4	4.8	5.4

Table 7.12: Collision distribution for hard test environment

Bibliography

[1] A. Ohnsman. *Tesla CEO Talking With Google About 'Autopilot' Systems*. https://www.bloomberg.com/news/articles/2013-05-07/tesla-ceo-talking-with-google-about-autopilot-systems. 2019.

[2] Jon Fingas. *Waymo launches its first commercial self-driving car service*. https://www.engadget.com/2018/12/05/waymo-one-launches/. 2018.

[3] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. "ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst". In: *CoRR* abs/1812.03079 (2018). arXiv: 1812.03079. URL: http://arxiv.org/abs/1812.03079.

[4] Keyu Wu et al. "Depth-Based Obstacle Avoidance through Deep Reinforcement Learning". In: *Proceedings of the 5th International Conference on Mechatronics and Robotics Engineering*. ICMRE'19. Rome, Italy: Association for Computing Machinery, 2019, pp. 102–106. ISBN: 9781450360951. DOI: 10.1145/3314493.3314495. URL: https://doi.org/10.1145/3314493.3314495.

[5] Andrea Ortalda et al. "Safe Driving Mechanism: Detection, Recognition and Avoidance of Road Obstacles". In: Jan. 2018, pp. 96–107. DOI: 10.5220/0006935400960107.

[6] Gregory Kahn et al. "Uncertainty-Aware Reinforcement Learning for Collision Avoidance". In: *CoRR* abs/1702.01182 (2017). arXiv: 1702.01182. URL: http://arxiv.org/abs/1702.01182.

[7] Hyunmin Chae et al. "Autonomous Braking System via Deep Reinforcement Learning". In: *CoRR* abs/1702.02302 (2017). arXiv: 1702.02302. URL: http://arxiv.org/abs/1702.02302.

[8] Horia Porav and Paul Newman. "Imminent Collision Mitigation with Reinforcement Learning and Vision". In: *CoRR* abs/1901.00898 (2019). arXiv: 1901.00898. URL: http://arxiv.org/abs/1901.00898.

[9] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *CoRR* abs/1711.03938 (2017). arXiv: 1711.03938. URL: http://arxiv.org/abs/1711.03938.

[10] ed WHO. *Global status report on road safety 2015*. https://www.who.int/violence_injury_prevention/road_safety_status/2015/en/. 2015.

[11] US Department of Transportation. *National Motor Vehicle Crash Causation Survey*. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/811059. 2018.

[12] Kai Olav Ellefsen and Jim Tørresen. "Self-Adapting Goals Allow Transfer of Predictive Models to New Tasks". In: *CoRR* abs/1904.02435 (2019). arXiv: 1904.02435. URL: http://arxiv.org/abs/1904.02435.

[13] Peter W. Battaglia et al. "Interaction Networks for Learning about Objects, Relations and Physics". In: *CoRR* abs/1612.00222 (2016). arXiv: 1612.00222. URL: http://arxiv.org/abs/1612.00222.

[14] Katerina Fragkiadaki et al. "Learning Visual Predictive Models of Physics for Playing Billiards". In: *CoRR* abs/1511.07404 (2015).

[15] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. *Learning Multimodal Transition Dynamics for Model-Based Reinforcement Learning*. 2017. arXiv: 1705.00470 [stat.ML].

[16] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[17] Martijn van Otterlo and Marco Wiering. "Reinforcement Learning and Markov Decision Processes". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: https://doi.org/10.1007/978-3-642-27645-3_1.

[18] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[19] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. "Transforming Auto-Encoders". In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Ed. by Timo Honkela et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 44–51.

[20] Diederik P. Kingma and Max Welling. "An Introduction to Variational Autoencoders". In: *CoRR* abs/1906.02691 (2019). arXiv: 1906.02691. URL: http://arxiv.org/abs/1906.02691.

[21] Carl Doersch. *Tutorial on Variational Autoencoders*. 2016. arXiv: 1606.05908 [stat.ML].

[22] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". In: *CoRR* abs/1808.03314 (2018). arXiv: 1808.03314. URL: http://arxiv.org/abs/1808.03314.

[23] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[24] Kai Olav Ellefsen, Charles Patrick Martin, and Jim Tørresen. "How do Mixture Density RNNs Predict the Future?" In: *CoRR* abs/1901.07859 (2019). arXiv: 1901.07859. URL: http://arxiv.org/abs/1901.07859.

[25] Christopher M. Bishop. "Mixture density networks". In: 1994.

[26] M. Wooldridge and N. Jennings. "Intelligent Agents: Theory and Practice". In: *Knowl. Eng. Rev.* 10.2 (1995), pp. 115–152. URL: http://www.dsl.uow.edu.au/~aditya/csci370/readings/Wooldridge.Jennings.95.pdf.

[27] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. USA: John Wiley Sons, Inc., 2004. ISBN: 0470861207.

[28] Anchit Gupta, James Harrison, and Emma Brunskill. "CS234 Notes - Lecture 11,12 Exploration and Exploitation". In: *Stanford* (2014). URL: https://web.stanford.edu/class/cs234/slides/lnotes11.pdf.

[29] Marvin Minsky. "Steps toward Artificial Intelligence". In: *Proceedings of the IRE* 49 (1961), pp. 8–30.

[30] Sergey Levine. *UC Berkeley Reinforcement Learning Class: Model-Based Reinforcement Learning*. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_9_model_based_rl.pdf. 2015.

[31] Sergey Levine. *UC Berkeley Reinforcement Learning Class: Deep RL with Q-Functions*. http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-8.pdf. 2015.

[32] Ankit Choudary. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python.* Ed. by medium.com. Apr. 2019. URL: https://medium.com/analytics-vidhya/a-hands-on-introduction-to-deep-q-learning-using-openai-gym-in-python-b15d7d8597d.

[33] Hado V. Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

[34] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[35] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[36] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *CoRR* abs/1710.02298 (2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.

[37] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

[38] Lukasz Kaiser et al. "Model-Based Reinforcement Learning for Atari". In: *CoRR* abs/1903.00374 (2019). arXiv: 1903.00374. URL: http://arxiv.org/abs/1903.00374.

[39] Danijar Hafner et al. "Learning Latent Dynamics for Planning from Pixels". In: *CoRR* abs/1811.04551 (2018). arXiv: 1811.04551. URL: http://arxiv.org/abs/1811.04551.

[40] David Ha and Jürgen Schmidhuber. "World Models". In: *CoRR* abs/1803.10122 (2018). arXiv: 1803.10122. URL: http://arxiv.org/abs/1803.10122.

[41] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. "Model-free Deep Reinforcement Learning for Urban Autonomous Driving". In: *CoRR* abs/1904.09503 (2019). arXiv: 1904.09503. URL: http://arxiv.org/abs/1904.09503.

[42] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. "Deep Imitation Learning for Autonomous Driving in Generic Urban Scenarios with Enhanced Safety". In: *CoRR* abs/1903.00640 (2019). arXiv: 1903.00640. URL: http://arxiv.org/abs/1903.00640.

[43] Axel Sauer, Nikolay Savinov, and Andreas Geiger. "Conditional Affordance Learning for Driving in Urban Environments". In: *CoRR* abs/1806.06498 (2018). arXiv: 1806.06498. URL: http://arxiv.org/abs/1806.06498.

[44] Felipe Codevilla et al. "End-to-end Driving via Conditional Imitation Learning". In: *CoRR* abs/1710.02410 (2017). arXiv: 1710.02410. URL: http://arxiv.org/abs/1710.02410.

[45] OpenAI et al. "Learning Dexterous In-Hand Manipulation". In: *CoRR* abs/1808.00177 (2018). arXiv: 1808.00177. URL: http://arxiv.org/abs/1808.00177.

[46] Yunhao Tang and Shipra Agrawal. "Discretizing Continuous Action Space for On-Policy Optimization". In: *CoRR* abs/1901.10500 (2019). arXiv: 1901.10500. URL: http://arxiv.org/abs/1901.10500.

[47] Abraham. Savitzky and M. J. E. Golay. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." In: *Analytical Chemistry* 36.8 (1964), pp. 1627–1639. DOI: 10.1021/ac60214a047. eprint: https://doi.org/10.1021/ac60214a047. URL: https://doi.org/10.1021/ac60214a047.

[48] Peter Henderson et al. "Deep Reinforcement Learning that Matters". In: *CoRR* abs/1709.06560 (2017). arXiv: 1709.06560. URL: http://arxiv.org/abs/1709.06560.

[49] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *CoRR* abs/1906.02243 (2019). arXiv: 1906.02243. URL: http://arxiv.org/abs/1906.02243.

[50] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[51] Yan Duan et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: *CoRR* abs/1604.06778 (2016). arXiv: 1604.06778. URL: http://arxiv.org/abs/1604.06778.

[52] Timothy Lillicrap et al. "Continuous control with deep reinforcement learning". In: *CoRR* (Sept. 2015).

[53] John Schulman et al. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[54] H. B. Mann and D. R. Whitney. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. In: *Ann. Math. Statist.* 18.1 (Mar. 1947), pp. 50–60. DOI: 10.1214/aoms/1177730491. URL: https://doi.org/10.1214/aoms/1177730491.

[55] Zhou Wang et al. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612.

[56] Yingjing Lu. “The Level Weighted Structural Similarity Loss: A Step Away from MSE”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 9989–9990. DOI: 10.1609/aaai.v33i01.33019989. URL: https://doi.org/10.1609/aaai.v33i01.33019989.

[57] P. Luc et al. “Predicting Deeper into the Future of Semantic Segmentation”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 648–657.

[58] Chao Peng et al. “Large Kernel Matters - Improve Semantic Segmentation by Global Convolutional Network”. In: *CoRR* abs/1703.02719 (2017). arXiv: 1703.02719. URL: http://arxiv.org/abs/1703.02719.

[59] Liang-Chieh Chen et al. “Rethinking Atrous Convolution for Semantic Image Segmentation”. In: *CoRR* abs/1706.05587 (2017). arXiv: 1706.05587. URL: http://arxiv.org/abs/1706.05587.

[60] Jacob Buckman et al. “Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 8224–8234. URL: http://papers.nips.cc/paper/8044-sample-efficient-reinforcement-learning-with-stochastic-ensemble-value-expansion.pdf.